TOPS-10/TOPS-20 COBOL-68 Language Manual

AA-5057B-TK

August 1981

This manual reflects the software of Version 12B of the COBOL-68 compiler, Version 12B of LIBOL, and Version 4C of SORT.

This manual replaces the document of the order numbers AA-5057A-TK, AD-5057A-T1, and AD-5057A-T2

OPERATING SYSTEM:	TOPS-10, Version 7.01 TOPS-20, Version 4
SOFTWARE VERSION:	COBOL-68, Version 12B LIBOL, Version 12B

digital equipment corporation • marlboro, massachusetts

First Printing, August 1969
Updated, May 1979
January 1980
Revised, August 1981

DEC	DECnet	IAS
DECUS	DECsystem-10	MASSBUS
Digital Logo	DECSYSTEM-20	PDT
PDP	DECwriter	RSTS
UNIBUS	DIBOL	RSX
VAX	EduSystem	VMS
		VT

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

FIGURES

FIGURES (Cont.)

FORWARD

This manual describes COBOL-68 as implemented on both TOPS-10 and TOPS-20. This manual is a complete manual containing reference material, user's guide material, and COBOL utilities. Chapter 1 discusses language elements, conventions used in this manual, and the structure of a COBOL-68 program. Chapters 2 through 5 describe the four major divisions of a COBOL-68 program. Chapters 6 through 13 provide the information necessary to use the COBOL-68 system, including performance improvements, utility programs, file formats, report writing, and various other useful features of COBOL-68.

Several Appendixes, A through E, plus a Glossary of COBOL terms are included in this manual. Appendix A contains the COBOL reserved words, Appendix B contains the character collating sequence, Appendix C describes how to define logical names under TOPS-20, Appendix D describes an alternate form of numeric test, and Appendix E describes Tape Handling.

It is assumed that the reader has a knowledge of the COBOL-68 language. This manual is intended primarily for reference and is not a tutorial guide for beginning COBOL programmers. Those wishing to learn the COBOL-68 language are referred to the following books:

Farina, Mario V., COBOL Simplified, New Jersey, Prentice Hall, Inc., 1968.

McCameron, Fritz A., COBOL Logic and Programming, Homewood, Illinois, Richard D. Irwin, Inc., 1966.

McCracken, Daniel D. and Garbassi, Umberto, A Guide to COBOL Programming, Second Edition, New York, John Wiley and Sons, Inc., 1970.

TOPS-10 users should read and be familiar with the following manuals:

- TOPS-10 Operating System Commands Manual
- TECO Programmer's Reference Manual
- TOPS-10 Monitor Calls Manual
- TOPS-10 Hardware Reference Manual
- TOPS-10 LINK Reference Manual
- TOPS-10 SORT/MERGE User's Guide

TOPS-20 user's should read and be familiar with the following manuals:

- TOPS-20 Commands Reference Manual
- TOPS-20 TV Reference Manual
- TOPS-20 EDIT Reference Manual
- TOPS-20 Monitor Calls Manual
- TOPS-20 Hardware Reference Manual
- TOPS-20 LINK Reference Manual
- TOPS-20 SORT/MERGE User's Guide

ACKNOWLEDGMENT

CHAPTER 1

INTRODUCTION TO COBOL-68 LANGUAGE

This chapter describes the conventions, special terms, language elements, and formats acceptable to COBOL-68. The source language statements are discussed in subsequent chapters.

NOTE

For the purposes of this document, the terms COBOL and COBOL-68 are interchangeable.

1.1 SYMBOLS AND TERMS

The symbols and terms used in the following chapters of this manual are necessary to describe the language or are commonly used COBOL terms. The single exception of this statement is the term BIS-compiler. This term refers to compiler implementations that compile COBOL-68 using the Business Instruction Set (BIS). All users of TOPS-20 get BIS code. Users of TOPS-10 who have a KS or KL central processing unit get BIS code as the default, but the compiler can be installed without the BIS option. TOPS-10 users who have a KI central processor will usually not get the BIS option on their compilers. The KI processor will not execute the BIS instructions; however, the KI will run the compiler which produces BIS code should there be a need for it. (For more information, see the COBOL-68 Installation Procedures.) You can tell if your compiler is producing BIS code by checking a listing of a compiled program. If your compiler is producing the BIS instructions, the letters BIS will follow the version and edit numbers on top of the page.

1.1.1 Symbols

The symbology used in this manual to illustrate the various COBOL statement formats is essentially the same as that used in other COBOL language manuals and is based on the CODASYL COBOL reference document.

Symbology	Meaning
Lower-case characters	Represent information that must be supplied by the programmer, such as values, names, and other parameters.

Symbology	Meaning
Upper-case characters, Underscored	Key words in the COBOL lexicon that must be used when the formats of which they are a part are used.
Upper-case characters, not underscored	Other words in the COBOL lexicon that serve only to make the COBOL statement more readable. Their use is optional and has no effect on the meaning of the formats of which they are a part.
Braces	Indicate that a choice must be made from the two or more lines enclosed.
Brackets	Indicate an optional feature. The contents of the brackets are used according to the rules above if the feature is desired.
Ellipsis...	Indicate that the information contained within the preceding pair of braces or brackets can be repeated at the programmer's option.

1.1.2 COBOL Terms

The terms block, record, and item have special meanings when used in a COBOL program.

Term	Meaning
Block	Signifies a logical grouping of records. This term commonly refers to a logical block of records on some storage medium.
Record	Signifies a logical unit of information. In relation to a data file, a record is the largest unit of logical information that can be accessed and processed at a time. Records can be subdivided into fields or items.
Item	Signifies a logical field or group of fields within a record. A group item is one that is further broken down into subitems (for example, a group item called TAX might be broken down into subitems called FED-TAX and STATE-TAX). Subitems can be further broken down into other subitems. An item that has no subitems is called an elementary item.

1.2 ELEMENTS OF COBOL LANGUAGE

1.2.1 Program Structure

A COBOL program consists of four divisions. Within each division are the program statements; some are required, others are optional.

Division	Meaning
IDENTIFICATION DIVISION	Identifies the source program.
ENVIRONMENT DIVISION	Describes the computer on which the source program is to be compiled, the computer on which the object program is to run, and certain relationships between program elements and hardware devices.
DATA DIVISION	Describes the data to be processed by the object program.
PROCEDURE DIVISION	Describes the actions to be performed on the data.

NOTE

There is no limit to the number of source lines the compiler can handle. However, the largest source line number that the compiler can generate is 8184. Beyond that number, the compiler begins again with 0001. This can cause confusion when error messages are issued.

1.2.2 Character Set

Within a source program statement, all ASCII characters are valid except:

1. Null, delete, and carriage return (which are ignored);

2. Line feed, vertical tab, form feed, and the printer control characters (20(8) through 24(8)), which mark the end of a source line;

3. Control-Z, which marks the end-of-file.

The lower case ASCII characters are translated to upper case characters except when they appear in nonnumeric literals.

Of this character set, 37 characters (the digits 0 through 9, the 26 letters of the alphabet, and the hyphen) can be used by the programmer to form COBOL words, such as data-names, procedure-names, and identifiers.

Punctuation characters include:

	(space)	" or '	(quotation mark)
,	(comma)	(	(left parenthesis)
;	(semicolon)	)	(right parenthesis)
.	(period)		(horizontal tab)

Special editing characters include:

+	(plus sign)	*	(check protection symbol)
-	(minus sign)	Z	(zero suppression)
$	(dollar sign)	B	(blank insertion)
,	(comma)	0	(zero insertion)
.	(decimal point)	CR	(credit)
		DB	(debit)

Special characters used in arithmetic expressions include:

+	(addition)	/	(division)
-	(subtraction)	**	(exponentiation)
*	(multiplication)	^	(exponentiation)

Special characters used in conditional (IF) statements include:

=	(equal)	>	(greater than)	<	(less than)

1.2.3 **Words**

A COBOL word is composed of not more than 30 characters chosen from the 37 characters A through Z, 0 through 9, and hyphen. A word is terminated by a space, period, right parenthesis, comma, semi-colon, or horizontal tab. A hyphen can not be used as the first or last character of a word. If the terminator is not a space or horizontal tab, at least one space or tab must follow the terminator.

Words used in writing COBOL source programs are of two types: COBOL reserved words and user-created words.

1.2.3.1 **COBOL Reserved Words** - COBOL reserved words are those words that constitute the COBOL lexicon and have a special meaning to the compiler (for example, DIVISION, PROCEDURE, ADD); these words are listed in Appendix A. They include all the COBOL division, section, and paragraph names, descriptive clauses, procedure verbs, certain prepositions, figurative constants, and special registers. Reserved words must be spelled and used exactly as shown in the formats given in this manual.

Figurative Constants - Figurative constants are reserved words that specify certain fixed values. When these reserved words are to be used as figurative constants, they must not be enclosed in quotation marks; otherwise they are treated by the compiler as alphanumeric literals.

The figurative constants are given below. Except for one case (the ALL constant), singular and plural forms are given; these forms are equivalent and can be used interchangeably.

Figurative Constant	**Use**
ZERO ZEROS ZEROES	Represents the value zero or one or more of the character 0 depending on context.
SPACE SPACES	Represents one or more blanks or spaces.
HIGH-VALUE HIGH-VALUES	For DISPLAY-6, DISPLAY-7, and DISPLAY-9 items this represents the highest value in the collating sequence. For COMP and COMP-1 items, this represents the largest number that can be placed in the machine word(s) containing the item. For COMP-3 items, this represents all 9s with the nonprinting plus sign.
LOW-VALUE LOW-VALUES	For DISPLAY-6, DISPLAY-7, and DISPLAY-9 items this represents the lowest value in the collating sequence. For COMP and COMP-1 items, this represents the smallest number (most negative) that can be placed in the machine word(s) containing the item. For unsigned COMP-3 items, this represents all zeros with the nonprinting plus sign; for signed COMP-3 items, this represents all 9s with a minus sign.
QUOTE QUOTES	Represents one or more quotation marks ("). It can be used anywhere that the quotation mark character (") is valid, except to delimit alphanumeric literals (see Section 1.2.4.2, Alphanumeric Literals). QUOTE(S) is frequently used where an actual quotation mark character would erroneously appear to delimit an alphanumeric literal. For example, if you wanted your program to type out the exact character string MOUNT TAPE LABELLED "MASTER" ON DRIVE 3 you could use the procedure statement DISPLAY "MOUNT TAPE LABELLED" QUOTE "MASTER" QUOTE "ON DRIVE 3".
ALL any-literal	Represents repetitions of the string of characters that constitute either an alphanumeric literal or a figurative constant (other than ALL any-literal). If a figurative constant is used, the ALL is redundant; thus, ZEROS and ALL ZEROS are equivalent.

Figurative constants generate a string of characters whose length is determined, based on context, by the compiler. For example, if TOTAL-AMOUNT is a five-character field, the procedure statement MOVE ALL ZEROS TO TOTAL-AMOUNT moves a string of five zeros to the field TOTAL-AMOUNT; MOVE ALL "AB" TO TOTAL-AMOUNT moves "ABABA" to TOTAL-AMOUNT. If the length cannot be determined by context, a single character (or a single-character sequence, in the case of ALL) is generated. For example, the procedure statement DISPLAY ALL QUOTES results in the output of a single quotation mark (") to your terminal.

Examples of Use of Figurative Constants:

DATA DIVISION Usage:	02 AMOUNT PICTURE IS 9999.99 VALUE IS ZERO. 04 MESSAGE PICTURE IS A(10) VALUE IS SPACES.
PROCEDURE DIVISION Usage:	MOVE ZEROS TO AMOUNT. MOVE SPACES TO MESSAGE. IF TOTAL IS EQUAL TO ZERO.... EXAMINE FLD-A TALLYING LEADING ZEROS.

Special Registers - In addition to figurative constants, COBOL recognizes two other special reserved-words: TALLY and TODAY.

TALLY is the name of a fixed five-digit signed COMPUTATIONAL field. It is used primarily to hold information produced by the EXAMINE verb. However, the programmer can use TALLY in any situation where a signed numeric field is valid (for example, temporary storage of any integer value of five or fewer digits).

TODAY is a 12-character alphanumeric DISPLAY field that contains the current date and time. Its format is:

yymmddhhmmss

where	yy is the year (last two digits)	hh is the hour
	mm is the month	mm is the minute
	dd is the day	ss is the second

1.2.3.2 **User-Created Words** - User-created words are labels for the various parts of your data (files, records, and fields) and your procedure (sections and paragraphs). They can contain only the symbols 0 through 9, A through Z, and the hyphen. With the exception of procedure names, they cannot be all digits. A user-created word can neither begin nor end with a hyphen. The maximum number of user-created words allowed in the program is 4681.

User-created words can be further subdivided into several categories. To understand the remainder of this manual, you should be familiar with the following types of words.

data-name	The user-created name assigned to an item (field) within a record.
file-name	The user-created name assigned to a data file.

record-name	The user-created name assigned to a data record within a file.
procedure-name	The user-created name assigned to a paragraph or section in the PROCEDURE DIVISION. When assigned to a section, it is referred to as a section-name; and when assigned to a paragraph, it is referred to as a paragraph-name.
identifier	A user-created name used in PROCEDURE DIVISION statement formats to indicate a data-name followed, as required, by the syntactically correct combination of qualifiers, and/or subscripts, and/or indexes necessary to make reference to a unique item of data.
mnemonic-name	A user-created name assigned to a hardware device or a report code.
condition-name	A user-created name assigned to a value or range of values of the associated data item. Condition-name can also be assigned to console switch settings.
index-name	A user-created name defined using the INDEXED BY clause (see OCCURS in Chapter 4). Its function is identical to that of an index data-name (see below).
index data-name	A user-created name defined with USAGE INDEX. Its function is identical to that of an index-name.

1.2.4 Literals

A literal is a string of characters, the value of which is identical to the characters that compose the literal. Literals are of two types: numeric and nonnumeric.

1.2.4.1 Numeric Literals - A numeric literal is a string of 1 to 18 numeric characters (0 through 9). It cannot contain any alphabetic characters. It can be preceded by a plus sign (+) or a minus sign (-); if no sign is used, the literal is assumed to be positive. A decimal point can appear anywhere in the literal except to the left of the sign or as the rightmost character. If no decimal point is used, the literal is assumed to be an integer. A numeric literal is considered to be of the numeric class; that is, it can be used legitimately as a value in arithmetic expressions.

Examples of Numeric Literals:

```
123      -123       +123       1.23456      .123456789
-.123456789    1234567890.12345678       -1234567890.12345678
```

1.2.4.2 **Alphanumeric Literals** - Alphanumeric literals are character strings containing from 1 to 120 characters enclosed in single or double quotation marks. The value of the literal is equal to the characters, including any spaces, enclosed by the quotation marks. Note that the compiler accepts either single or double quotation marks to enclose a literal; however, the opening and closing quotation marks must be the same type, either single or double. Any ASCII character except the quotation mark, null, delete, carriage return, and printer control can appear within a literal.

Alphanumeric literals cannot be used as values in arithmetic operations, and numeric editing cannot be performed on them. If a literal conforms to the rules for formation of a numeric literal, but is enclosed in quotation marks, it is considered to be an alphanumeric literal. That is, "120.45" is not equivalent to 120.45.

Examples of Nonnumeric Literals:

```
"A"        'THIS ACCOUNT HAS A CREDIT BALANCE'     "RETURN"
"-125.50"        'DEDUCT 10% IF PAID BEFORE JAN.31ST'
```

1.2.5 **Punctuation**

The punctuation that can be used in source programs includes the space, comma, semicolon, and period.

The space is used to separate words, phrases and clauses. The comma and semicolon can be used interchangeably within a program to improve the appearance of the program. However, both the comma and the semicolon are treated as spaces by the compiler; they can be used any place in the program where a space is expected.

The period is used to terminate a division name, a section name, and a paragraph name. It is also used in the PROCEDURE DIVISION to terminate sentences. Paragraphs and sections are terminated by the period ending the last sentence of the paragraph or section. In the DATA DIVISION, a period must be placed after the description of a data item. Examples of the use of periods are:

```
PROCEDURE DIVISION.
    .
    .
INPUT SECTION.
    .
    .
    READ INFIL AT END GO TO ENDER.
DATA DIVISION.
FILE SECTION.


01 MYDATA PICTURE IS X(10).
```

1.3 SOURCE PROGRAM FORMAT

There are two basic types of source program formats in which you can write your COBOL-68 programs. These two types arise from the methods of entering the source program into the system. The first is conventional card-type format. You should use this type if you wish your COBOL-68 program to be compatible with other compilers. The second is the standard DEC format which is designed for easy use on terminals. This format is the one to use for those programs that are to be entered into the system through a terminal using a text editor. The compiler assumes that the source program is written in terminal-type format unless the /S switch is included in the command string to the compiler (refer to Chapter 6).

Certain margins which begin the areas used for writing COBOL-68 statements are standard for source programs. The standard names for these margins are Margins L, A, B, and R. As you might expect, Margins L and R are the left and right margins of the line, respectively. Margins A and B mark the beginning of two areas, Areas A and B. Area A is where all division-names, section-names, paragraph-names, and FD (File Description) entries must begin. All other entries must begin in Area B. Although the actual character position which marks each of these margins changes from format to format, the function of each area is the same; in other words, you must begin your division-names at Margin A no matter what format you use, no matter where Margin A happens to be placed in that format.

NOTE

These rules agree with the 1968 ANSI standard for source program formats. Programs written according to the rules are more readable and transportable. The COBOL-68 compiler, however, does not do complete syntax checking to determine if you have followed all rules, and does not always issue an error message if you violate them. Thus, you are encouraged to conform to the rules to avoid unpredictable results.

Some of the rules for using source program formats remain constant regardless of which format you use. These rules are given below. Refer to them for all types of formats.

1. Continuation Area - If you wish to split a word or literal across two lines, you must use this area to indicate your wish to the compiler. To do this, write the first line up to the point at which you wish to split it, then place a hyphen (-) in the continuation area of the next line and continue the second line beginning at or after Margin A. If you are splitting a word or numeric literal you can leave spaces between the last character in the first line and the end of the source statement area. (This area ends at the identification area, when it exists; otherwise it ends at Margin R.) However, if you wish to split an alphanumeric literal you must not leave spaces after the last character of the first line, since the compiler assumes that those spaces are part of the literal. If you wish only to continue a sentence on the next line without splitting any words, you can simply write the first line, then continue on the next line; do not use the continuation column for this purpose.

2. Comment Lines - You can insert comment lines into your COBOL-68 program by using the continuation area. If the compiler finds an asterisk (*) in that area it lists the remainder of the line as a comment on the next line. If there is a slash (/) instead of an asterisk a new page is started and the comment is listed at the top of the new page.

NOTE

All formats can be used with any input medium. The names of the types of formats refer to their origins, not their uses.

1.3.1 Card-Type Format

You should use card-type format if you wish to compile your program under an operating system other than TOPS-10 or TOPS-20. Your program can be punched on an off-line card punch or created with an on-line text editor. This format uses card sequence numbers which must be created by you. The layout of a line in this format is shown in Figure 1-1(a). The numbers refer to card columns or character positions.

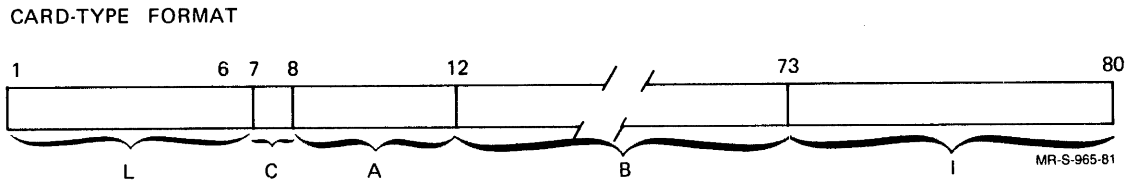


Figure 1-1(a) Card-Type Format

In this format, Margin L is to the left of position 1 and Margin R is to the right of position 80. Margin A is between positions 7 and 8 and begins the area labeled A in the figure. Margin B is between positions 11 and 12 and begins the area labeled B.

The following rules pertain to the use of this source format:

1. Line Numbers - These are placed in area L (positions 1 through 6) by you when creating the file on a terminal or a card punch.

2. Identification Area - This area is marked I in the figure (positions 73 through 80). These eight character positions can hold identifying information which can be composed of any eight characters. This information is printed on the source listing, and can be used to identify the card deck (if the source code is in fact on cards).

NOTE

The card sequence numbers are not the same as the line numbers created by a line editor. The numbers supplied by an editor are not acceptable to COBOL-68 when you specify card-type format.

The example in Figure 1-1(b) illustrate these rules. The first two lines are simple statements, with a line number in area L, COBOL-68 statements in areas A and B, and the identification area containing the name of the program. The third line shows how the continuation column is used to split a word across two lines. Note that the word can be written right up to the end of area B.

1.3.2 **Terminal-Type Format**

If you are writing your program using a text editor and a terminal to input the source code, terminal-type format is your best choice. There are two types of terminal-oriented formats, one with line numbers and one without. Layouts and examples of each type are shown in the figures which follow.

1.3.2.1 **With Line Numbers** - This format is suitable if you use a line-oriented editor such as EDIT or SOS. The format is shown in Figure 1-2(a).

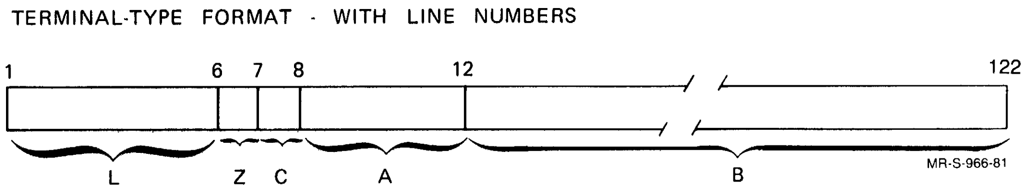


Figure 1-2(a) Terminal-Type Format with Line Numbers

In this format, margin L is to the left of position 1 and margin R is to the right of position 122. Margin A is between positions 7 and 8 and begins the area labeled A. Margin B is between positions 11 and 12 and begins the area labeled B. Therefore, areas A and B can contain a maximum of 114 characters.

The following rules pertain to the use of this source format:

1. Line Numbers - These are placed in area L (positions 1 through 5) either by the line editor or by you. If you are using an editor which supplies line numbers you must not add numbers yourself - one set is enough.

2. Position 6 - This position (marked Z in the figure) remains blank. The editor can insert a tab here for purposes of making your text more readable; if so, the compiler reads the tab as a space.

3. Continuation Area - To use the continuation area, type -, *, , or / as the first character of the line. However, if you do not wish to use the continuation area, you can ignore it altogether - you do not need to type a space at the beginning of the line. If you do type a space as the first character of a line, the compiler assumes that you meant the space to be part of the line.

The example in Figure 1-2(b) illustrate the use of this format. The first two lines are simple COBOL-68 statements with the five-character line number in area L and areas Z and C blank. The third line shows how a word is split across two lines. Note that you can leave spaces between the last letter of the word and margin R without confusing the compiler.

1.3.2.2 **Without Line Numbers** - If you decide to use a terminal to enter your program but your editor (such as TECO or TV) does not supply line numbers (or you requested that the editor remove them when you finished editing), this is the simplest format to use. The format is shown in Figure 1-3(a).

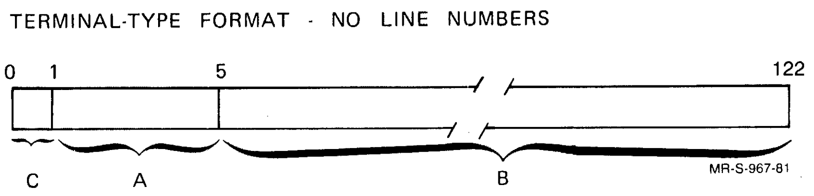


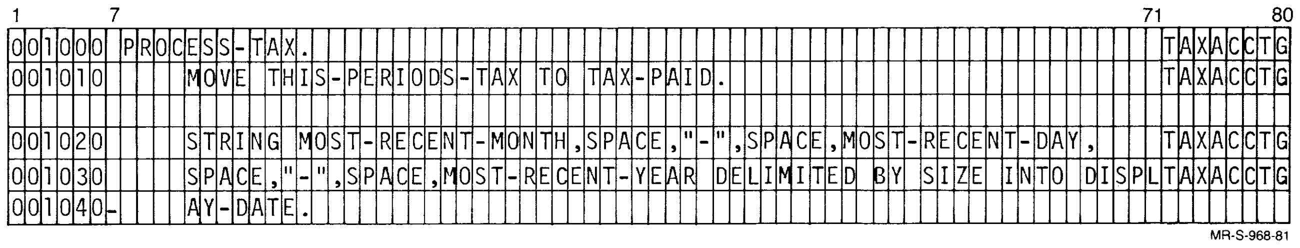
Figure 1-3(a) Terminal-Type Format without Line Numbers

In this format, margin L is to the left of position 0, if it exists, or position 1, if position 0 does not exist. Margin R is to the right of position 122. Margin A is to the left of position 1 and begins the area labeled A. Margin B is between positions 4 and 5 and begins the area labeled B. Therefore, areas A and B can contain a maximum of 114 characters.

The following rule pertains to the use of this source format:

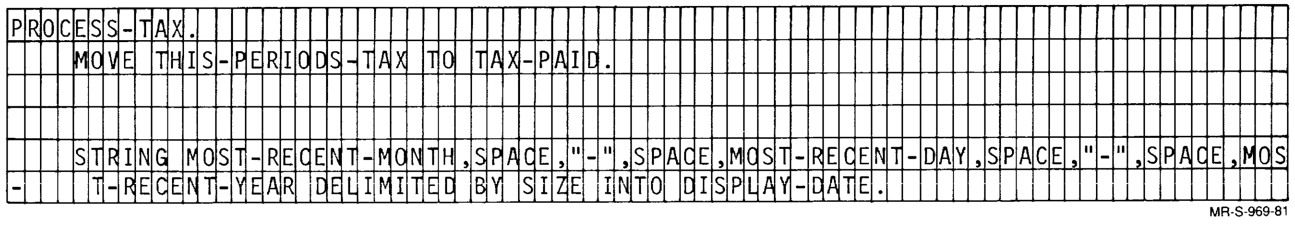
> Continuation Area - If you wish to use the continuation area, type the character you wish to enter (-, *, /) as the first character of the continued line. If the compiler finds one of these characters at the beginning of a line it assumes that the line has a position 0 - in other words, a continuation area. Otherwise, each line starts in position 1 and there is no position 0.

The example in Figure 1-3(b) show this format's simplicity. The first two lines are the same simple COBOL-68 sentences as above. Note that the paragraph-name starts in the very first character position. The third line shows how to tell the compiler that the line you enter is a continuation (or a comment) line. The first half of the line is entered beginning in the first position of Area B, while the second half begins with a hyphen and continues from the second position.


1
7
71
80
MR-S-968-81

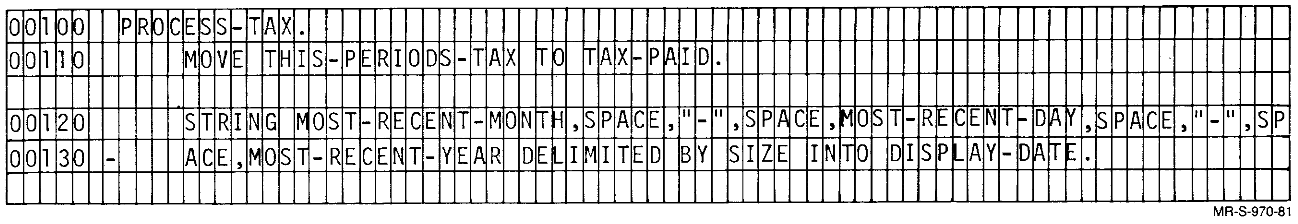
```
001000 PROCESS-TAX.                                                     TAXACCTG
001010     MOVE THIS-PERIODS-TAX TO TAX-PAID.                           TAXACCTG

001020     STRING MOST-RECENT-MONTH,SPACE,"-",SPACE,MOST-RECENT-DAY,    TAXACCTG
001030     SPACE,"-",SPACE,MOST-RECENT-YEAR DELIMITED BY SIZE INTO DISPLTAXACCTG
001040-    AY-DATE.
```

Figure 1-1 (b)



```
PROCESS-TAX.
    MOVE THIS-PERIODS-TAX TO TAX-PAID.


    STRING MOST-RECENT-MONTH,SPACE,"-",SPACE,MOST-RECENT-DAY,SPACE,"-",SPACE,MOS
-    T-RECENT-YEAR DELIMITED BY SIZE INTO DISPLAY-DATE.
```

Figure 1-2 (b)



```
00100  PROCESS-TAX.
00110      MOVE THIS-PERIODS-TAX TO TAX-PAID.

00120      STRING MOST-RECENT-MONTH,SPACE,"-",SPACE,MOST-RECENT-DAY,SPACE,"-",SP
00130 -    ACE,MOST-RECENT-YEAR DELIMITED BY SIZE INTO DISPLAY-DATE.

```

Figure 1-3 (b)

1.4 THE COBOL LIBRARY FACILITY

You can use the COBOL Library Facility to copy part of your program from a COBOL source library at compile time. This can be useful if, for example, you need to describe a complex file to be used in several different programs, and you wish to write the file description only once. You can insert the file description into the library (for directions and further description see the COBOL-68 Usage Material, Part 3 of this manual), and whenever the description is needed you can simply copy it from the library into the program you are writing. The following statement is used to accomplish this.

NOTE

The COPY facility for COBOL-68 is the enhanced version from the ANSI-74 standard, and not the original one from the ANSI-68 standard.

1.4.1 The COPY Statement

Function

The COPY statement incorporates text from a COBOL library into a COBOL source program. (For a complete description of COBOL libraries, see the COBOL-68 Usage Material, Part 3 of this manual.) The COPY statement can also be used to replace specified text in the source text being copied.

General Format

$$\underline{\text{COPY}}\ \text{text-name}\ \left[\begin{Bmatrix}\underline{\text{OF}}\\ \underline{\text{IN}}\end{Bmatrix}\ \text{library-name}\right]$$

$$\left[\underline{\text{REPLACING}}\left\{\begin{Bmatrix}\text{==pseudo-text-1==}\\ \text{identifier-1}\\ \text{literal-1}\\ \text{word-1}\end{Bmatrix}\ \underline{\text{BY}}\ \begin{Bmatrix}\text{==pseudo-text-2==}\\ \text{identifier-2}\\ \text{literal-2}\\ \text{word-2}\end{Bmatrix}\right\}\ldots\right]\underline{.}$$

MR-S-971-81

Technical Notes

NOTE

In the technical notes which follow, the term string-1 is used to denote the character string which is used in place of the following: pseudo-text-1, identifier-1, literal-1, or word-1. The term string-2 is similarly used.

1. If more than one COBOL library is available during compilation, text-name must be qualified by the library-name identifying the COBOL library in which the text associated with text-name resides.

   Within one COBOL library, each text-name must be unique.

2. The COPY statement must be preceded by a space and terminated by the separator period. The entire statement, including the period, is removed when the text is copied from the library.

3. String-1 must not be null, nor can it consist solely of the character space(s), nor can it consist solely of comment lines.

4. String-2 can be null.

5. Character-strings within string-1 and string-2 can be continued. However, both characters of a pseudo-text delimiter must be on the same line.

6. A COPY statement can occur in the source program anywhere a character-string or a separator can occur except that a COPY statement must not occur within another COPY statement.

7. The effect of processing a COPY statement is that the library text associated with text-name is copied into the source program, logically replacing the entire COPY statement, beginning with the reserved word COPY and ending with the punctuation character period, inclusive. The compilation of a source program containing COPY statements is logically equivalent to processing all COPY statements prior to the processing of the resulting source program. For clarity, use the double equal sign (==) around string-1 and string-2 to designate clearly the string that is being replaced and the string that is replacing that text. See Note 10 for an example of the use of the double equal sign.

8. If the REPLACING phrase is not specified, the library text is copied unchanged. If the REPLACING phrase is specified, the library text is copied and each properly matched occurrence of string-1 in the library text is replaced by the corresponding string-2.

9. The comparison operation to determine text replacement occurs as follows:

   a. Any separator comma, semicolon, and/or space(s) preceding the leftmost library text-word is copied into the source program. Starting with the leftmost library text-word and the first string-1 that was specified in the REPLACING phrase, the entire REPLACING phrase operand that precedes the reserved word BY is compared to an equivalent number of contiguous library text-words.

   b. String-1 matches the library text if, and only if, the ordered sequence of text-words that forms string-1 is equal, character for character, to the ordered sequence of library text-words. For purposes of matching, each occurrence of a separator comma or semicolon in string-1 or in the library text is considered to be a single space except when string-1 consists solely of either a separator comma or semicolon, in which case it participates in the match as a text-word. Each sequence of one or more space separators is considered to be a single space.

c. If no match occurs, the comparison is repeated with each next successive string-1, if any, in the REPLACING phrase until either a match is found or there is no next successive REPLACING operand.

d. When all the REPLACING phrase operands have been compared and no match has occurred, the leftmost library text-word is copied into the source program. The next successive library text-word is then considered as the leftmost library text-word, and the comparison cycle starts again with the first string-1 specified in the REPLACING phrase.

e. Whenever a match occurs between string-1 and the library text, the corresponding string-2 is placed into the source program. The library text-word immediately following the rightmost text-word that participated in the match is then considered as the leftmost library text-word. The comparison cycle starts again with the first string-1 specified in the REPLACING phrase.

f. The comparison operation continues until the rightmost text-word in the library text has either participated in a match or been considered as a leftmost library text-word and participated in a complete comparison cycle.

10. When you use the REPLACING phrase, you must treat any picture strings in the library text as complete pieces of text. That is, if you wish to replace X's in the picture string

```
EXAMPLE-ITEM PICTURE IS XXX.
```

with 9's, you must replace the entire PICTURE clause, not just the three X's, with the form shown below:

```
COPY EXAMPLE-TEXT FROM LIBARY REPLACING      ==PICTURE IS
XXX== BY ==PICTURE IS 999==.
```

11. For purposes of matching, a comment line which occurs in the library text and string-1 is interpreted as a single space. Comment lines which appear in string-2 and library text are copied into the source program unchanged.

12. The text produced as a result of the complete processing of a COPY statement must not contain a COPY statement.

13. The syntactic correctness of the library text cannot be independently determined. The syntactic correctness of the entire COBOL source program cannot be determined until all COPY statements have been completely processed.

14. Library text must conform to the rules for COBOL source program format. (See Section 1.3.) You can copy text from a library without worrying about what format your program is in, however.

15. For purposes of compilation, text-words after replacement are placed in the source program according to the rules for source program format.

CHAPTER 2

THE IDENTIFICATION DIVISION

The IDENTIFICATION DIVISION is required in every source program and identifies the source program and the output from compilation. In addition, you can include other documentary information such as the name of the program's author, the name of the installation, the dates on which the program was written and compiled, any special security restrictions, and any miscellaneous remarks.

General Structure

```
{ID            } DIVISION.
{IDENTIFICATION}

    [ PROGRAM-ID. [ program-name ] [ comment paragraph ] . ]

    [ AUTHOR. comment paragraph . ]

    [ INSTALLATION. comment paragraph . ]

    [ DATE-WRITTEN. comment paragraph . ]

    [ DATE-COMPILED. comment paragraph . ]

    [ SECURITY. comment paragraph . ]

    [ REMARKS. comment paragraph . ]  MR-S-972-81
```

Technical Notes

1. The Identification Division must begin with the reserved words IDENTIFICATION DIVISION (or ID DIVISION) followed by a period and a space. ID is the equivalent to IDENTIFICATION.

2. The PROGRAM-ID paragraph contains the name identifying the program. The program-name can have up to six characters, and must contain only letters, digits, and the hyphen. It can be enclosed in quotation marks. The program-name cannot be a reserved word and must be unique. It cannot be the same as a section, paragraph, file, data or subprogram name. This paragraph is optional. If it is not present, the name MAIN is assigned to the program.

3. The remaining paragraphs are optional and, if used, can appear in any combination and in any order. A comment paragraph consists of any combination of characters from the COBOL character set organized to conform to COBOL sentence and paragraph format. All text appears as written on the output listing except the DATE-COMPILED paragraph. The first line in this paragraph is deleted and replaced by the current date. Any remaining text in the DATE-COMPILED paragraph is treated as comments. Reserved words can be used in any comment paragraph.

CHAPTER 3

THE ENVIRONMENT DIVISION

The Environment Division allows you to describe the particular computer configurations to be used for program compilation and execution. In this division you also specify the files and devices you will use for input and output. The Environment Division consists of the division header (ENVIRONMENT DIVISION.) followed by one or more of the following sections:

CONFIGURATION SECTION.	(See Section 3.1)
INPUT-OUTPUT SECTION.	(See Section 3.2)

CONFIGURATION SECTION

3.1 CONFIGURATION SECTION

The CONFIGURATION SECTION allows you to describe the computers used for program compilation and execution, and to assign mnemonic-names for input/output devices. The Configuration Section consists of the section name (CONFIGURATION SECTION.) followed by one or more of the following paragraphs.

SOURCE-COMPUTER.	(See Section 3.1.1)
OBJECT-COMPUTER.	(See Section 3.1.2)
SPECIAL-NAMES.	(See Section 3.1.3)

Technical Notes

1. This section is optional.
2. All commas and semicolons are optional. A period must terminate the entire entry in each of the three paragraphs.

3.1.1 SOURCE-COMPUTER

Function

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

General Format

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

MR-S-973-81

Technical Notes

1. This paragraph is optional.
2. You must use one of the following terms for computer-name:

   DECsystem-10
   PDP-10
   DECSYSTEM-20
   DECsystem-10nn

   where nn is a 2-digit integer in the range from 00 to 99.

Example

```
SOURCE-COMPUTER.  DECSYSTEM-1055.
```

OBJECT-COMPUTER

3.1.2 OBJECT-COMPUTER

Function

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

General Format

```
OBJECT-COMPUTER    {computer-name}

  [                          ( CHARACTERS ) ]
  [ MEMORY SIZE  integer-1   { WORDS      } ]
  [                          ( MODULES    ) ]

  [SEGMENT-LIMIT IS integer-2]  .

             (DISPLAY-6)
  DISPLAY IS {DISPLAY-7}   .
             (DISPLAY-9)
```

MR-S-974-81

Technical Notes

1. This paragraph is optional.
2. You must use one of the following terms for computer-name:

   DECsystem-10
   PDP-10
   DECSYSTEM-20
   DECsystem-10nn

   where nn is a 2-digit integer in the range 00 to 99.
3. The MEMORY SIZE clause is optional. If it is omitted, 262,144 WORDS are assumed. If it appears, the following ranges are applicable.

CHARACTERS	Up to 1,572,864 (262,144 words x 6 characters/word)
WORDS	Up to 262,144
MODULES	Up to 256 (1 module equals 1024 words)

OBJECT-COMPUTER (Cont.)

COBOL-68 ignores the MEMORY SIZE clause. SORT uses its default algorithms to determine the amount of memory needed to execute a sort. (Refer to the TOPS-10 and the TOPS-20 SORT User's Guides for more information.)

4. If the SEGMENT-LIMIT clause is given, only those segments having segment numbers from 0 up to, but not including, the value of integer-2 are considered as resident segments of the program. Integer-2 must be a positive integer in the range 1 to 49.

   If the SEGMENT-LIMIT clause is omitted, segments having segment numbers from 0 through 49 are considered as resident segments of the program (that is, SEGMENT-LIMIT IS 50 is assumed). More on segmentation can be found in Chapter 5.

5. The DISPLAY IS clause is optional. If you specify DISPLAY IS, then all data-items described as DISPLAY defaults to the specified DISPLAY type. Using the DISPLAY IS clause also causes the recording mode for external files to default to the specified DISPLAY type.

Example

```
OBJECT-COMPUTER.  DECSYSTEM-1077
     MEMORY 50000 WORDS.
```

SPECIAL–NAMES

3.1.3 SPECIAL–NAMES

Function

THE SPECIAL–NAMES paragraph provides a means of assigning mnemonic names to input/output devices.

General Format

```
┌                                                                          ┐
│ SPECIAL-NAMES. [ CONSOLE IS mnemonic-name-1 ]                            │
│                                                                          │
│      [ CHANNEL (m) IS mnemonic-name-2 ]                                  │
│                                                                          │
│          [ CHANNEL (n) IS mnemonic-name-3 ... ]                          │
│                                                                          │
│      ┌                 ⎧ IS mnemonic-name-4 [ ON STATUS IS condition-name-1 ] ⎫
│      │                 ⎪      [ OFF STATUS IS condition-name-2 ]              ⎪
│      │                 ⎪                                                      ⎪
│      │                 ⎪ ON STATUS IS condition-name-1                        ⎪
│      │ SWITCH (m)     <       [ OFF STATUS IS condition-name-2 ]              >
│      │                 ⎪                                                      ⎪
│      │                 ⎪ OFF STATUS IS condition-name-2                       ⎪
│      │                 ⎪      [ ON STATUS IS condition-name-1 ]               ⎪
│      └                 ⎩                                                      ⎭
│                                               ┐                          │
│                                               │                          │
│          [ SWITCH (n) ... ] ...               │                          │
│                                               │                          │
│                                               ┘                          │
│      [ literal-1 IS mnemonic-name-5 ]                                    │
│                                                                          │
│      [ CURRENCY SIGN IS literal-2 ]                                      │
│                                                                          │
│      [ DECIMAL-POINT IS COMMA ] .                                        │
└                                                                          ┘
```

MR-S-975-81

SPECIAL–NAMES (Cont.)

Technical Notes

1. This paragraph is optional.

2. The reserved word CONSOLE refers to your terminal. The assigned mnemonic-name can be used with the ACCEPT and DISPLAY verbs in the PROCEDURE DIVISION to input data from and output data to the terminal.

3. The name CHANNEL refers to a channel on the line-printer control tape. m and n represent any integer from 1 to 8 and refer to any one of the eight channels on the tape. Control tape channels can be referred to in the ADVANCING clause of the WRITE verb in the PROCEDURE DIVISION to advance the paper form to the desired channel position. (Refer to the Hardware Reference Manual for a description of printer control tapes.) For example, if the entry

   ```
   CHANNEL (1) IS TOP-OF-PAGE
   ```

   is included in this paragraph, the following procedure statement prints the line and then skip to the top of the next page.

   ```
   IF LINE-COUNT IS GREATER THAN 50 WRITE PRINT-RECORD
   BEFORE ADVANCING TOP-OF-PAGE.
   ```

4. The reserved word SWITCH is provided for compatibility with other manufacturers' COBOL compilers. The use of the SWITCH feature is discouraged in a time-sharing environment. If provided, the name SWITCH refers to the hardware switches on the KA-10 or KI-10 console. The letters m and n in the general format represent any integer from 0 to 35 and refer to the corresponding console switches.

   The mnemonic-name can be used in conditional expressions in the PROCEDURE DIVISION. For example, if the entry

   ```
   SWITCH (4) IS INPUT-1
   ```

   is included in this paragraph, the following condition is considered to be true if switch (4) is on.

   ```
   IF INPUT-1 IS ON....
   ```

   If a condition-name is specified for the ON or OFF STATUS of a switch, that condition-name can be used in a conditional expression. For example, if the entry

   ```
   SWITCH (4) IS INPUT-1;  OFF STATUS IS NO-INPUT
   ```

   is included in this paragraph, the following procedure statements are functionally equivalent.

   ```
   IF INPUT-1 IS OFF....

   IF NO-INPUT....
   ```

SPECIAL-NAMES (Cont.)

5. The clause literal-1 IS mnemonic-name-5 specifies the CODE value for a particular report (refer to the CODE clause in Chapter 4). Literal-1 must be a nonnumeric literal enclosed in quotation marks, and can be from 1 through 120 characters in length.

6. If you use the CURRENCY SIGN clause in the SPECIAL NAMES paragraph, then the literal you specify replaces the standard $ character functions for PICTURE clauses in the DATA DIVISION.

   This literal is limited to a single printable character and must not be one of the following characters:

   digits 0 through 9

   alphabetic characters A, B, C, D, P, R, S, V, X, Z

   special characters * + - , . ; ( ) "

7. If you use the DECIMAL-POINT IS COMMA clause then the functions of the comma and period are interchanged for all PICTURE clauses and numeric literals.

Example

```
SPECIAL-NAMES.  CONSOLE IS MYTERM
     CHANNEL (1) IS TOP-OF-PAGE
     SWITCH (10) IS LOOPER.
```

3.2 INPUT-OUTPUT SECTION

The INPUT-OUTPUT SECTION names the files and external media required by the object program and provides information required for transmission and handling of data during execution of the object program. This section consists of the section header (INPUT-OUTPUT SECTION.) followed by one or more of the following paragraphs:

FILE-CONTROL	(See Section 3.2.1)
I-O-CONTROL	(See Section 3.2.2)

Technical Notes

1. This section is optional.
2. All semicolons and commas are optional. Each SELECT statement in the FILE-CONTROL paragraph must end with a period. The entire entry in the I-O-CONTROL paragraph must end with a period.

FILE-CONTROL

3.2.1 FILE-CONTROL

Function

The FILE-CONTROL paragraph names each file, identifies the file medium, and allows logical hardware assignments.

General Format

```
┌                                                                              ┐
│FILE-CONTROL.SELECT  OPTIONAL  file-name                                      │
│                                                                              │
│    ASSIGN TO device-name-1   [,device-name-2]  ...                           │
│   ┌                     ┐                                                    │
│   │ FOR MULTIPLE  { REEL }│                                                  │
│   │               { UNIT }│                                                  │
│   └                     ┘                                                    │
│   ┌                                            ┐                             │
│   │ RESERVE  { integer-1 }   ALTERNATE  [ AREA  ]│                           │
│   │          { NO        }              [ AREAS ]│                           │
│   └                                            ┘                             │
│  ┌ ( FILE LIMIT IS   )  ┌                      ┐                           ┐ │
│  │ ) FILE-LIMIT IS   (  │{ data-name-1 }  THRU │  { data-name-2 }          │ │
│  │ ) FILE-LIMITS ARE (  │{ literal-1   }       │  { literal-2   }          │ │
│  └ ( FILE LIMITS ARE )  └                      ┘                           │ │
│        ┌                                              ┐                    │ │
│        │ , { data-name-3 }  THRU  { data-name-4 }     │  ...               │ │
│        │   { literal-3   }        { literal-4   }     │                    ┘ │
│        └                                              ┘                      │
│ ┌                ( SEQUENTIAL  [WITH CHECKPOINT OUTPUT [EVERY integer-1 RECORDS]] )┐│
│ │                ( RANDOM                                                  )││
│ │ ACCESS MODE IS <                                                          >││
│ │                ( INDEXED    [WITH{ CHECKPOINT OUTPUT [EVERY integer-1 RECORDS]}])││
│ │                (                 { DEFERRED OUTPUT                         } )││
│ └                                                                            ┘│
│   PROCESSING MODE IS SEQUENTIAL                                              │
│   ACTUAL KEY IS data-name-5                                                  │
└                                                                              ┘
```

MR-S-976-81

FILE-CONTROL (Cont.)

```
[ { SYMBOLIC } KEY IS data-name-6, RECORD KEY IS data-name-7 ]
  { NOMINAL  }

                             { ASCII          }
                             { SIXBIT         }
                             { BINARY         }
[ RECORDING  MODE IS         { F              } ]
                             { V              }
                             { STANDARD-ASCII }
                             { STANDARD ASCII }

            { 200  }
            { 556  }                       { ODD  }
[ DENSITY IS{ 800  } ] [ PARITY IS         { EVEN } ]
            { 1600 }
            { 6250 }

[ FILE-STATUS    IS data-name-8   [,data-name-9   [,data-name-10
  FILE STATUS

    [,data-name-11   [,data-name-12   [,data-name-13

    [,data-name-14   [,data-name-15]]]]]]] .

              [ SELECT ....] ...
```

MR-S-977-81

Technical Notes

1. This paragraph is optional.
2. All semicolons and commas are optional. Each SELECT clause must end with a period.
3. The SELECT and ASSIGN clauses must appear before any other clause shown, and the SELECT clause must precede the ASSIGN clause. Every file described in the Data Division must be named in a SELECT clause in the Environment Division. Therefore, a SELECT filename ASSIGN TO device-name clause must be specified for every file. The other clauses can be in any order.
4. The individual clauses are described on the following pages in the order shown above.

SELECT

3.2.1.1 SELECT

Function

The SELECT statement names each file that is to be described in the DATA DIVISION, and assigns each file to a particular device.

General Format

$$\underline{\text{SELECT}}\ \ [\underline{\text{OPTIONAL}}]\ \ \text{file-name}\ \underline{\text{ASSIGN}}\ \text{TO}\ \begin{Bmatrix}\text{literal-1}\\ \text{device-name-1}\end{Bmatrix}\ \begin{bmatrix}\text{,literal-2}\\ \text{,device-name-2}\end{bmatrix}\ldots$$

MR-S-978-81

Technical Notes

1. Each file described in the DATA DIVISION must be named once and only once as a file-name in a SELECT statement. Conversely, each file named in a SELECT statement must have a File Description entry in the DATA DIVISION. Each file-name must be unique within a program.

2. The key word OPTIONAL is required for input files that are not necessarily present each time the object program is run. When an OPEN statement is executed for a file that has been declared OPTIONAL, the question IS file-name PRESENT? is typed and the operator responds with YES or NO. If the response is YES, the file is processed normally; if the response is NO, the first READ statement executed for that file immediately takes the AT END or INVALID KEY path. ISAM files can not be optional. They must be present at program start-up, even if only as dummies.

3. The ASSIGN clause specifies the device for a file. Device-names can be either physical device-names or logical device-names.

   Physical device-names are fixed mnemonic-names that are associated with specific peripheral devices. When specified in an ASSIGN clause, a physical device-name assigns the associated file to that device. Physical device-names are described in the Operating System Commands manual.

   Logical device-names are names created by the programmer. They can contain up to six characters, consisting of any combination of letters and digits. At object execution time, each logical device-name must be assigned to a physical device by means of the TOPS-10 ASSIGN command or the TOPS-20 DEFINE command.

4. The use of literals with the ASSIGN clause allows you to use COBOL reserved words as legal device names. The literal name must follow the same conventions as the device-name: literals can contain up to six characters, consisting of any combination of letters and digits. At object execution time, each literal must be assigned to a physical device by means of the TOPS-10/TOPS-20 ASSIGN command or the TOPS-20 DEFINE command.

5. More than one device can be assigned to a file to avoid delay when switching from one reel or unit to the next. When more than one device is specified, the object program automatically uses the next device, in a cyclic manner, when an end-of-reel condition is detected, or when a CLOSE REEL statement is executed. This automatic switching occurs only for tapes, SORT, and ISAM files. It is unconditional for tapes. For SORT/MERGE, you can assign any number of devices. If the devices are all generic disk (i.e. DSK), SORT/MERGE uses its internal optimal algorithm to determine which physical devices to use. For any other devices, all devices specified are used in a round-robin fashion. You can assign only two devices when you use ISAM.

6. If the access mode is INDEXED, and two devices are assigned, the first device is assumed to contain the index portion of the file and the second to contain the data portion of the file. If one device is specified, it is assumed to contain both the index portion and the data portion of the file.

7. For ISAM and random files, the devices must be random-access.

Examples

```
SELECT INFIL ASSIGN TO DTA1.

SELECT SRTFIL ASSIGN TO DSK, DSK, DSK.
```

FOR MULTIPLE

3.2.1.2 FOR MULTIPLE

Function

THE FOR MULTIPLE clause specifies that a tape-file occupies more reels than the number of devices assigned. The FOR MULTIPLE clause does nothing when your program is compiled. It is merely used for documentation purposes only.

General Format

$$\left[\text{FOR } \underline{\text{MULTIPLE}} \left\{ \begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{UNIT}} \end{array} \right\} \right]$$

MR-S-979-81

Example

```
     SELECT OUTFIL ASSIGN TO MTA
          FOR MULTIPLE REEL.
```

3.2.1.3 RESERVE

Function

The RESERVE clause allows you to specify an additional number of input/output buffer areas to be allocated by the compiler to this file.

General Format

$$\left[\underline{\text{RESERVE}} \left\{ \begin{matrix} \text{integer-1} \\ \underline{\text{NO}} \end{matrix} \right\} \text{ ALTERNATE } \left[\begin{matrix} \text{AREA} \\ \text{AREAS} \end{matrix} \right] \right]$$

MR-S-980-81

Technical Notes

1. If the access mode is RANDOM or INDEXED, this clause is ignored and only one buffer area is assigned.

2. If the NO option is used, only one buffer area is allocated.

3. If the integer-1 option is used, the integer specifies the number of buffer areas to be assigned in addition to the two areas always assigned by the compiler. If integer-1 is less than 0, only one buffer area is assigned.

4. You can specify a maximum of 62 areas for integer-1. However, the optimal number of areas you can specify is between 5 and 10. If you specify the number of areas to be greater than 62, a warning message is generated. If you specify a large (but legal) number of areas, you might run out of available memory. Specifying a large number of areas might also cause your program to run more slowly, since your program is that much bigger.

Example

```
SELECT INFIL ASSIGN TO DSK
     RESERVE 1 ALTERNATE AREA.
```

FILE-LIMIT

3.2.1.4 FILE-LIMIT

Function

The FILE-LIMIT clause is used to define the logical limits of a file whose access mode is RANDOM.

General Format

```
[ { FILE LIMIT IS    }   [ { data-name-1 }  THRU ]  { data-name-2 }
  { FILE-LIMIT IS    }   [ { literal-1   }       ]  { literal-2   }
  { FILE LIMITS ARE  }
  { FILE-LIMITS ARE  }

        [ , { data-name-3 }  THRU  { data-name-4 } ] ... ]
        [   { literal-3   }        { literal-4   } ]     ]
                                                              MR-S-981-81
```

Technical Notes

1. The FILE-LIMIT clause is required only for files whose access mode is RANDOM; it is optional for files with SEQUENTIAL access mode residing on mass-storage devices. It is ignored in all other cases.

2. The words FILE and LIMIT (or LIMITS) can be separated with the space or hyphen.

3. Every data-name used in this clause must be defined as USAGE COMP or INDEX and must be an integer of 10 digits or less.

4. Each pair of operands represents a logical portion of the file. If the first operand of the first pair is not specified, it is assumed to be 1.

5. The operands represent logical record numbers relative to the beginning of the file. The first record is considered to be 1.

6. The logical beginning and end of a random-access file are the records specified by the first and last operands, respectively, of the FILE-LIMIT clause.

7. The values of data items specified in this clause are used by the object-time system only when the file is opened by an OPEN statement.

8. If a file whose access mode is RANDOM is processed sequentially, the FILE-LIMIT clause is ignored. Thus, you can create records with keys higher than the upper FILE-LIMIT.

FILE-LIMIT (Cont.)

Example

```
     SELECT INFIL ASSIGN TO DSK
          FILE LIMIT IS 3000.
```

ACCESS MODE

3.2.1.5 ACCESS MODE

Function

The ACCESS MODE clause specifies the way in which a file is accessed.

General Format

```
[                  { SEQUENTIAL  [WITH CHECKPOINT OUTPUT [EVERY integer-1 RECORDS]] }  ]
[                  { RANDOM                                                         }  ]
[ ACCESS MODE IS   {                                                                }  ]
[                  { INDEXED     [WITH { CHECKPOINT OUTPUT [EVERY integer-1 RECORDS] }] } ]
[                  {                   { DEFERRED OUTPUT                             }] } ]
                                                                              MR-S-982-81
```

Technical Notes

1. The ACCESS MODE clause is required for random-access and indexed-sequential files. It is ignored for sequential files.

2. If ACCESS MODE IS SEQUENTIAL and the file is on a random-access device, the random-access records are obtained or placed sequentially. That is, the next logical record is made available from the file on a READ statement execution, and an output record is placed into the next available area on a WRITE statement execution. Thus sequential access processing on a random-access device is functionally similar to the processing of a magnetic tape file.

3. If ACCESS MODE IS RANDOM, the contents of the data item associated with the ACTUAL KEY specifies which record, relative to the beginning of the file, is made available by a READ statement, or where the record is to be placed by a WRITE statement.

4. If ACCESS MODE IS INDEXED, the contents of the data item associated with the SYMBOLIC KEY specifies which record is made available by a READ statement, or where the record is to be placed by a WRITE statement, or which record is to be deleted by a DELETE statement, or which record is replaced by a REWRITE statement.

5. The DEFERRED OUTPUT option of the INDEXED ACCESS MODE causes the object-time system to output a block of an indexed sequential file only when another block must be brought into core. Normally, to ensure security for the file, a block is output every time a record is written, even if records are written successively in the same block. When a file is opened for simultaneous update, the DEFERRED OUTPUT clause is ignored. Refer to the OPEN statement in Chapter 5.

ACCESS MODE (Cont.)

6. If you are using ISAM files sequentially, DEFERRED OUTPUT provides the advantage of running faster. However, it is also more easily damaged if the system crashes. Thus, its use is advantageous if file integrity is not important.

7. Specifying the CHECKPOINT OUTPUT phrase causes a checkpoint FILOP. UUO to occur after every physical output. For sequential files, every output may not coincide with each WRITE. For ISAM files, the CHECKPOINT OUTPUT causes the FILOP. UUO to occur after every output. For ISAM files, every output coincides with each WRITE.

   Using the CHECKPOINT OUTPUT phrase has the same effect as if you were to use a CLOSE followed by an OPEN. However, the CHECKPOINT OUTPUT phrase performs the action much faster than the OPEN and CLOSE verbs. Using the CHECKPOINT OUTPUT phrase increases the reliability of your ISAM files, but it slows down the overall performance of your program.

8. If integer-1 is zero, or if you do not specify the EVERY integer-1 RECORDS clause, the checkpointing actions occurs after every physical write.

Example

```
SELECT INFIL ASSIGN TO DSK, DSK
     ACCESS MODE IS INDEXED WITH DEFERRED OUTPUT.
```

3.2.1.6 PROCESSING MODE

Function

The PROCESSING MODE clause specifies that the file is to be processed sequentially.

General Format

[<u>PROCESSING</u> MODE IS <u>SEQUENTIAL</u>]

MR-S-983-81

Technical Notes

This clause is for documentation only; records are always processed in the order in which they are accessed.

Example

```
SELECT INFIL ASSIGN TO MTA1
     ACCESS MODE IS SEQUENTIAL
     PROCESSING MODE IS SEQUENTIAL.
```

3.2.1.7 ACTUAL KEY

Function

The ACTUAL KEY clause specifies which record is read or written in a random-access file.

General Format

```
[ ACTUAL KEY IS data-name ]
```

MR-S-984-81

Technical Notes

1. The ACTUAL KEY clause is valid only for files whose access mode is RANDOM; it must be specified for those files. This clause cannot be used for files whose access modes are INDEXED or SEQUENTIAL.

2. The ACTUAL KEY data-name must be defined in the DATA DIVISION as a COMPUTATIONAL item of ten or fewer digits. The PICTURE can contain only the characters S and 9 or their equivalent, for example S9(10).

Example

```
SELECT INFIL ASSIGN TO DSK
     ACCESS MODE IS RANDOM
     ACTUAL KEY IS RKEY.
```

SYMBOLIC KEY

3.2.1.8 SYMBOLIC KEY

Function

The SYMBOLIC KEY clause specifies the record in an indexed-sequential file that is to be read, written, deleted or rewritten.

General Format

$$\left[\left\{\begin{array}{l}\underline{\text{SYMBOLIC}}\\ \underline{\text{NOMINAL}}\end{array}\right\}\ \text{KEY IS data-name-1, }\underline{\text{RECORD}}\text{ KEY IS data-name-2}\right]$$

MR-S-985-81

Technical Notes

1. NOMINAL KEY is completely equivalent to SYMBOLIC KEY.
2. The SYMBOLIC KEY clause is valid only for files whose access mode is INDEXED; it must be specified for those files (refer to the READ statement in Chapter 5).
3. The SYMBOLIC KEY data-item must be defined in the DATA DIVISION, and must not appear in the record area of the file to which it pertains. It must agree with the description of the RECORD KEY data item in class, usage, size, and number of decimal places.
4. The RECORD KEY data-item must be defined as an item in the record area of the file to which it pertains. Though the RECORD KEY is described in only one of the records, it is assumed to occupy the same position in all records for that file.
5. The RECORD KEY is required to describe the location in the record area of the key for the file. The contents of the RECORD KEY data-item must be unique for each record in the file, and cannot be equal to LOW-VALUES (refer to the READ, WRITE, REWRITE, and DELETE statements in Chapter 5).

Example

```
SELECT INFIL ASSIGN TO DSK, DSK
     ACCESS MODE IS INDEXED
     SYMBOLIC KEY IS SYMKEY, RECORD KEY IS RECKEY.
```

3.2.1.9 RECORDING MODE/DENSITY/PARITY

Function

The RECORDING MODE clause specifies the recording mode, tape density, and parity for a magnetic tape file.

General Format

```
                                     ( ASCII          )
                                     | SIXBIT         |
[           [                        | BINARY         | ]
[ RECORDING [ MODE IS [BYTE MODE]   <  F              > ]
[           [                        | V              | ]
                                     | STANDARD-ASCII |
                                     ( STANDARD ASCII )

  [            ( 200  ) ]                              ]
  [            | 556  | ]                              ]
  [ DENSITY IS <  800 > ]  [ PARITY IS  { ODD  } ]     ]
  [            | 1600 | ]               { EVEN }       ]
  [            ( 6250 ) ]                              ]
```

MR-S-986-81

Technical Notes

1. The RECORDING MODE clause allows you to record data on the device in a format other than that used in memory. The following recording modes are acceptable.

   ASCII - The file is read/written as ASCII records, five 7-bit characters per 36-bit word. Bit 35 (the rightmost bit) is ignored.

   SIXBIT - The file is read/written as SIXBIT records, six 6-bit characters per 36-bit word with record headers.

   BINARY - The file is read/written as binary records, 36 bits per word.

   F - The file is read/written as fixed-length EBCDIC records, four 9-bit characters per 36-bit word for everything but industry-compatible magnetic tape. For industry-compatible magnetic tape (9-track, with 800, 1600, and 6250 bpi density), the file is read/written with four 8-bit characters per 36-bit word. If more than one record description is given in the FD entry, the record length must be the same for all records in the entry.

RECORDING MODE/DENSITY/PARITY (Cont.)

V - The file is read/written as variable-length EBCDIC records, four 9-bit characters per 36-bit word with record and block headers. However, for industry-compatible magnetic tape (9-track, with 800, 1600, or 6250 bpi density), the file is read/written with four 8-bit characters per 36-bit word. If a file whose recording mode is V is open for INPUT-OUTPUT and you overwrite a record, the record being written must be the same size as the overwritten record. A file whose recording mode is V cannot be opened for simultaneous update.

STANDARD-ASCII (STANDARD ASCII)

The five 7-bit bytes in each word in core are transferred to five 8-bit bytes on the tape and bit 35 is stored in bit 0 of the fifth byte on tape. The character set and the character encodings are the same as those of ASCII recording mode. This enables interchanges with other manufacturers' ASCII data files.

The format of records for each recording mode is given in Section 4.11.2.11.

2. The recording mode of a file is determined by a number of factors besides the recording mode specified in the RECORDING MODE clause. These factors are:

   a. If the device can only accept ASCII data (For example, Line Printer), the object-time system always uses ASCII as the recording mode no matter what recording mode is specified.

   b. If the ADVANCING or POSITIONING clause is included in the WRITE statement, the object-time system always uses ASCII as the recording mode no matter what recording mode is specified.

   c. If the file descriptor (FD) has a REPORT clause, the object-time system always uses ASCII as the recording mode no matter what recording mode is specified.

   d. The recording mode specified in the RECORDING MODE clause is compared to the USAGE clause for the record. The recording mode is determined in the following sequence:

      1. The recording mode that is specified is used.

      2. If the recording mode is not specified, the default recording mode depends on the usage mode that is specified.

      3. If neither the recording mode nor the usage mode is specified, the default recording mode depends on the display mode.

RECORDING MODE/PARITY/DENSITY (Cont.)

4. If none of the above has been specified, the default recording mode is SIXBIT, unless the /X switch is included in the command string to the compiler or the DISPLAY is DISPLAY-6/7/9. If the /X switch is included, the default recording mode is F. If the DISPLAY is DISPLAY-6/7/9, then the default recording mode is SIXBIT/ASCII/F.

When the recording mode is not declared, it is inferred from the usage mode for the record according to the rules given above. However, the reverse is not true. That is, when the recording mode is declared and no usage mode is given for a record, the presence of the RECORDING MODE clause serves only to specify the recording mode of the file. The usage mode of the records in the file can default to another character set, with undesirable results (see the USAGE clause in Chapter 4). Table 3-1 shows the resulting recording mode when the recording mode declared in the RECORDING MODE clause is compared to the usage mode declared in the USAGE clause.

3. The DENSITY and PARITY clauses are valid only for magnetic tape, and are ignored for all other devices. If the DENSITY clause is not present, tapes are recorded in the density standard for the installation. The density for a job can be modified by the SET DENSITY command, which is described in the TOPS-10 Operating Systems Manual and the TOPS-20 User's Guide. A density of 1600 or 6250 bpi can be specified only for tapes being read/written to or from magnetic tape drives that can use that density. If the PARITY clause is omitted, ODD is assumed. Care must be taken when using even parity. If nulls are written into a file that is recorded in even parity, the file cannot be read properly. Nulls can be written into a file without you being aware of them; that is, when SYNCHRONIZED data items appear in an item, the word preceding the word in which the item is synchronized could contain nulls.

4. If BYTE MODE is used, the exact number of bytes is written on the tape. BYTE MODE truncates; it does not round up to word boundary. This occurs only on magnetic tape. BYTE MODE applies only to users of TOPS-10 and only to sequential files. Its purpose is to enable interchanges with other manufacturers' equipment.

Example

```
SELECT INFIL ASSIGN TO MTA1
     RECORDING MODE IS V
     DENSITY IS 800
     PARITY IS ODD.
```

RECORDING MODE/DENSITY/PARITY (Cont.)

Table 3-1
Recording Modes

RECORDING MODE Clause	USAGE Clause	Recording Mode actually used
none	DISPLAY-6	SIXBIT
none	DISPLAY-7	ASCII
none	DISPLAY-9	EBCDIC
none	none	SIXBIT (no /X) or DISPLAY-6
none	none	EBCDIC (/X) or DISPLAY-9
none	none	ASCII (DISPLAY-7)
SIXBIT	DISPLAY-6	SIXBIT
SIXBIT	DISPLAY-7	SIXBIT
SIXBIT	DISPLAY-9	SIXBIT
ASCII	DISPLAY-6	ASCII
ASCII	DISPLAY-7	ASCII
ASCII	DISPLAY-9	ASCII
F or V	DISPLAY-6	EBCDIC
F or V	DISPLAY-7	EBCDIC
F or V	DISPLAY-9	EBCDIC
BINARY	DISPLAY-6	BINARY
BINARY	DISPLAY-7	BINARY
BINARY	DISPLAY-9	BINARY

NOTE

Conversions necessary to make the recording mode conform to the usage mode of the records are made automatically by the object-time system. (These conversions can cause your program to run more slowly.)

3.2.1.10 FILE STATUS

Function

The FILE STATUS clause specifies data-items into which LIBOL places values when an I/O error or warning message occurs on the file specified by the SELECT clause. A user-written USE procedure can then examine and alter these values as part of a recovery process.

General Format

```
[ { FILE-STATUS } IS data-name-1 [,data-name-2 [,data-name-3 [,data-name-4
  { FILE STATUS }

      [,data-name-5 [,data-name-6 [,data-name-7 [,data-name-8] ] ] ] ] ] ] ]
```

MR-S-987-81

Technical Notes

1. Data-name-1 is required if this clause is specified; but data-name-2 through data-name-8 are optional. If fewer than eight data-names are specified, the compiler assumes that the data-names are specified starting with data-name-1 and continuing in order. Therefore, if data-name-8 is specified, data-name-1 through data-name-7 must also be specified.

FILE STATUS (Cont.)

2. The data-names must be defined in the WORKING STORAGE SECTION of the DATA DIVISION in the following form:

```
data-name-1      PIC 9(2).
data-name-2      PIC 9(10).
data-name-3      USAGE INDEX.
data-name-4      PIC X(9).
data-name-5      USAGE INDEX.
data-name-6      USAGE INDEX.
data-name-7      PIC X(30).
data-name-8      USAGE INDEX.
```

3. After a fatal I/O error, the FILE STATUS items contain the following values:

data-name-1 contains the file status.
data-name-2 contains a ten digit error number.
data-name-3 contains the action code, which is set to zero.
data-name-4 contains the VALUE OF ID.
data-name-5 contains the current block number.
data-name-6 contains the current record number.
data-name-7 contains the file name.
data-name-8 contains the file-table pointer.

The file status, which is stored in data-name-1, is set to one of the following 2-character codes.

00	The I/O was successful.
10	No next logical record; that is, there is no next record in the file. The AT END path is taken.
22	Duplicate key; that is, an attempt was made to write a record into a record position that is already occupied. The INVALID KEY path is taken.
23	No record found on READ, REWRITE, DELETE; that is, when an indexed sequential file was accessed, an empty record position was found. The INVALID KEY path is taken.
24	Boundary violation, that is, the random file's actual key violated the file limits. The INVALID KEY path is taken.
30	Permanent error; that is, a successful hardware operation cannot be done without a hardware error signal.
34	Permanent error; that is, more space on the media cannot be obtained to extend the file for output operations.

The 10-character error number stored in data-name-2 has the form:

ABCDEFGHIJ

where the code has the meanings shown below.

AB contains a value indicating the COBOL verb that caused the error.

0	No COBOL verb error
1	OPEN
2	CLOSE
3	WRITE
4	REWRITE
5	DELETE
6	READ
7	RETAIN

FILE STATUS (Cont.)

CD contains a value indicating the monitor call (UUO) that caused the error.

0	No UUO error
1	INPUT
2	OUTPUT
3	LOOKUP
4	ENTER
5	RENAME
6	INIT
7	FILOP
8	TAPOP

EF contains a value indicating the type of file being accessed when the error occurred.

0	None of the following
1	ISAM index file
2	ISAM data file
3	A sequential file
4	A random file

G contains a value indicating the ISAM block type that was being accessed when the error occurred.

0	None of the following
1	ISAM statistics block
2	ISAM SAT block
3	ISAM index block
4	ISAM data block

HIJ contains a value indicating an error number on INPUT or OUTPUT.

If CD is set to 0, HIJ contains an error number. The numbers and their meanings are listed below. Note that these are the same as the messages issued by LIBOL after an error or warning occurs.

0	None of the following.
1	SYMBOLIC-KEY must not equal low-values.
2	No more index levels available.
3	Insufficient core while attempting to split the top index block.
4	Version number discrepancy.
5	Allocation failure - all blocks are in use.
6	The maximum record size may not be exceeded.
7	Cannot expand core while SORT is in progress.
8	Insufficient core for buffer requirements.
9	Blocking-factor differs between index file and file-table.
10	File cannot be opened, already open.
11	Locked file cannot be opened.
12	File cannot be opened shares buffer area with opened file.
13	File cannot be opened device is not available to this job.
14	File cannot be opened device is assigned to another file.
15	File cannot be opened device cannot input/output.
16	File cannot be opened device cannot input.
17	File cannot be opened device cannot output.
18	File cannot be opened device is not a device.
19	File cannot be opened directory device must have standard labels.
20	File cannot be closed because it is not open.

FILE STATUS (Cont.)

21 File cannot be closed.
The CLOSE "REEL" option may not be used with a multi-file-tape.
22 File is not open for output.
23 Zero length records are illegal.
File cannot do output.
24 "AT END" path has been taken.
File cannot do input.
25 Encountered an "EOF" in the middle of a record.
File cannot do input.
26 RECORD-SEQUENCE-NUMBER n should be m.
File cannot do input.
27 file-name on device-name should be reorganized, the top index block was just split.
28 Not used.
29 Either the ISAM file does not exist or the VALUE OF ID changed during the program.
30 Attempt to do I/O from a subroutine called by a non resident subroutine. File cannot be opened.
31 I/O cannot be done from an overlay. File cannot be opened.
32 Read an "EOF" instead of a label.
33 CLOSE REEL is legal only for magnetic tape.
34 File is not open for input.
35 Not enough free core between .JBFF and overlay area.
36 Not enough free core between .JBFF and overlay area.
Insufficient core while attempting to split the top index block.
37 Standard ASCII recording mode and density of 1600 BPI require the device to be a TU70.
38 TAPOP. Failed - Unable to set STANDARD-ASCII mode.
39 Got an EOF in middle of BLOCK/RECORD descriptor word.
40 Block descriptor word byte count is less than five.
41 ERROR - Got another buffer instead of "EOF".
42 ERROR - Record extends beyond the end of the logical block.
43 It is illegal to change the record size of an EBCDIC IO record.
44 The two low order bytes of RDW/BDW must be zero, spanned EBCDIC not supported.
45 TAPOP. failed - Unable to set HARDWARE DATA MODE.
46 Unable to get mag tape status information.
47 Cannot set requested density.
48 TAPOP. failed - unable to get/set label type/information.
52 Improper tape label format for indicated recording mode.
53 Improper default hardware data mode for ASCII system-labeled tape.
54 ANSI-labeled "S" and "D" mag tape not supported.
55 Program can not have OPEN I/O and OPEN EXTEND for same file FD.
56 TAPOP. failed, unable to switch mag tape reels.

If CD is set to 1 or 2, HIJ contains the number of an I/O error status bit. The I/O error status bits, their mnemonics, and their meanings, are shown in Table 3-2.

FILE STATUS (Cont.)

Table 3-2
Monitor File Status Bits

Bit	Mnemonic	Meaning
18	IO.IMP	Improper Mode. Attempt to write on a software write-locked file structure, or a software redundancy failure occurred. This bit is usually set by the monitor. You cannot set this bit.
19	IO.DER	Hardware device error. The disk unit is in error, rather than the data on the disk. However, data read into core or written on the disk is probably incorrect. You do not usually set this bit.
20	IO.DTE	Hard data error. The data read or written has incorrect parity as detected by the hardware. Your data is probably unrecoverable even after the device has been fixed. This bit is usually not set by you.
21	IO.BKT	Block too large. A disk data block is too large to fit into the buffer; or a block number is too large for the disk unit; or DSK has been filled; or your quota on the file structure has been exceeded. This bit is usually not set by you. This error is also returned when you try to close a file that has open locks associated with it (using Enqueue/Dequeue).
22	IO.EOF	End-of-file. Your program has requested data beyond the last block of the file with an IN or INPUT call; or USETI has specified a block beyond the last data block of the file. When IO.EOF is set, no data has been read into the buffer.
23	IO.ACT	I/O Active. The disk is actively transmitting or receiving data. This bit is always set by the monitor.
29	IO.WHD	Write disk pack headers. This is used in conjunction with the SUSET. monitor call to format a disk pack.
30	IO.SYN	Synchronous mode I/O. Stop disk after every buffer is read or written.
31	IO.UWC	User word count, supplied by you in each buffer.
32-35	IO.MOD	Data mode of the device.

FILE STATUS (Cont.)

For the file status for each device, refer to the Monitor Calls Manual.

If CD is set to 3, 4, 5, or 7, HIJ contains the error code for LOOKUP, ENTER, RENAME, or FILOP errors. Table 3-3 gives these codes and their meanings.

Table 3-3
Monitor Error Codes

Code	Explanation
0	File not found, illegal filename (0,*), filenames do not match, or RENAME after a LOOKUP failed.
1	UFD does not exist on specified file structures. (Incorrect project-programmer number.)
2	Protection failure or directory full on DTA.
3	File being modified.
4	Already existing filename (RENAME) or different filename (ENTER after LOOKUP) or supersede (on a non-superseding ENTER).
5	Illegal sequence of UUOs (RENAME with neither LOOKUP nor ENTER, or LOOKUP after ENTER).
6	1. Transmission, device, or data error. 2. Hardware-detected device or data error detected while reading the UFD RIB or UFD data block. 3. Software-detected data inconsistency error detected while reading the UFD RIB or file RIB.
7	Not a saved file.
10	Not enough core.
11	Device not available.
12	No such device.
13	No 2-register relocation capability.
14	No room on this file structure or quota exceeded (overdrawn quota not considered).
15	Write-lock error. Cannot write on file structure.
16	Not enough table space in free core of monitor.

FILE STATUS (Cont.)

Table 3-3 (Cont.)
Monitor Error Codes

Code	Explanation
17	Partial allocation only.
20	Block not free on allocated position.
21	Cannot supersede an existing directory.
22	Cannot delete a non-empty directory.
23	Sub-directory not found (some SFD in the specified path was not found).
24	Search list empty (LOOKUP or ENTER was performed on generic device DSK and the search list is empty).
25	Cannot create a SFD nested deeper than the maximum allowed level of nesting.
26	No file structure in the job's search list has both the no-create bit and the write-lock bit equal to zero and has the UFD or SFD specified by the default or explicit path (ENTER on generic device DSK only).
27	GETSEG from a locked low segment to a high segment which is not a dormant, active, or idle segment. (Segment not on the swapping space.)
30	Cannot update file.
31	Low segment overlaps high segment.
32	Not logged in.

4. The FILE STATUS items are the communications paths between LIBOL and a USE procedure. A USE procedure specifies a recovery process executed when an error or warning occurs during an I/O operation. A USE procedure determines the error or warning type from the error-number placed into data-name-2 by LIBOL. Control returns to LIBOL at the conclusion of the USE procedure. The contents of the action-code placed into data-name-3 by the USE procedure and the error-number determine the subsequent LIBOL action. If the action-code is set to 1, LIBOL ignores the error and continues the run. If the action-code is left set to 0, LIBOL issues an error message and terminates the run. If the error-number is 17, LIBOL continues the run independent of the action-code setting. If the action-code is not 0 or 1, the LIBOL action is undefined.

FILE STATUS (Cont.)

When the program comes to a normal termination and you requested that errors be ignored, LIBOL issues the following message:

```
%n ERRORS IGNORED
```

5. Refer to the USE statement in Chapter 5 for details of writing USE procedures.

6. If the FILE STATUS statement is not specified, I/O error recovery processing cannot be performed. If the FILE STATUS statement is specified with only data-name-1 included, you can examine the status of the file, but you cannot specify that LIBOL ignore the error because you cannot set the action code (data-name-3). You also cannot examine the error number (data-name-2).

Example

```
                  .
                  .
                  .
SELECT INFIL ASSIGN DSK, DSK
    ACCESS MODE IS INDEXED
    SYMBOLIC KEY IS SYMKEY, RECORD KEY IS RECKEY
    RECORDING MODE IS ASCII
    FILE STATUS IS FILSTAT, ERRNUM, ACTCODE, VID,
    BLKNUM, RECNUM, FILNAM, FILPNTR.
                  .
                  .
                  .
    DATA DIVISION.
                  .
                  .
                  .

     WORKING-STORAGE SECTION.
     77 SYMKEY   PIC X(10).
     77 FILSTAT  PIC 9(2).
     77 ERRNUM   PIC 9(10).
     77 ACTCODE  INDEX.
     77 VID      PIC X(9).
     77 BLKNUM   INDEX.
     77 RECNUM   INDEX.
     77 FILNAM   PIC X(30).
     77 FILPNTR  INDEX.
```

3.2.2 I-O-CONTROL

Function

The I-O-CONTROL paragraph specifies the points at which a rerun dump is to be performed, the memory area that is to be shared by different files, and the location of files on a multiple-file reel.

General Format

```
┌                                                                              ┐
│ I-O-CONTROL.                                                                 │
│                                                                              │
│  ┌                     ┌              ┌      ┐ ┐                      ┐       │
│  │                     │              │ REEL │ │                      │       │
│  │ RERUN  EVERY        │  END  OF     │ UNIT │ │  OF  file-name-1     │       │
│  │                     │              └      ┘ │                      │       │
│  │                     │  integer-1   RECORDS  │                      │       │
│  └                     └                       ┘                      ┘       │
│                                                                              │
│  ┌      ┌ ┌        ┐ ┐                                                     ┐  │
│  │ SAME │ │ RECORD │ │  AREA  FOR  file-name-2,file-name-3 [ ,file-name-4 ] ... │  │
│  │      │ │ SORT   │ │                                                     │  │
│  └      └ └        ┘ ┘                                                     ┘  │
│                                                                              │
│  ┌                                                                           │
│  │ MULTIPLE  FILE  TAPE  CONTAINS  file-name-5 [ POSITION  integer-2 ]       │
│  │                                                                           │
│  └                                                                           │
│                                                                              │
│  [ ,file-name-6 [ POSITION  integer-3 ] ] ... ] ... .                        │
│                                                                              │
└                                                                              ┘
```

MR-S-988-81

Technical Notes

1. This paragraph is optional.

2. The RERUN clause specifies when a rerun dump is to be performed.

   The dump is always written onto a disk file, using the program's low segment name as the filename, and an extension of CKP. If the program has no filename because it was never SAVEd, the program name (from the PROGRAM-ID paragraph in the IDENTIFICATION DIVISION) is used as a filename, with the extension CKP.

   If the END OF UNIT option is used, a rerun dump is taken at the end of each input or output reel of the specified REEL file.

I-O-CONTROL (Cont.)

If the integer-1 RECORDS option is used, a rerun dump is taken whenever a number of logical records equal to a multiple of integer-1 is either read or written for the file.

A rerun dump is not taken if any file is open on a device other than magnetic tape, disk, or terminal. Also, RERUN cannot be used if overlays are used or if files are open for simultaneous update. Do not attempt to have a rerun dump taken while a sort is in progress.

3. The SAME AREA clause specifies that two or more files are to use the same area during processing; this includes all buffer areas and the record area. However, unless the RECORD option is used, only one of the named files can be open at one time.

   If the RECORD option is specified, the files share only the record area (that is, the area in which the current logical record is processed). If files sharing a record area are open at the same time but their records do not have the same usage mode, no conversion automatically takes place.

   The SORT option is used for sort files. However, this option need not be specified because all sort files always use the same sort area.

4. The MULTIPLE FILE clause is required when more than one file shares the same physical reel of tape. This clause is invalid for media other than magnetic tape.

   Regardless of the number of files on a single reel, only those files defined in the program can be listed. If all files residing on the tape are listed in consecutive order, the POSITION option need not be given. If any file on the tape is not listed, the POSITION option must be included; integer-2, integer-3, and so forth, specify position of the file relative to the beginning of the tape. All files on the same reel of tape must be ASSIGNed to the same device in the FILE-CONTROL paragraph.

   Not more than one file on the same reel of tape can be open at one time.

Example

```
I-O-CONTROL.
     RERUN EVERY 300 RECORDS OF INFIL
     SAME RECORD AREA FOR INFIL, OUTFIL
     MULTIPLE FILE TAPE CONTAINS INFIL POSITION 4.
```

CHAPTER 4

THE DATA DIVISION

The Data Division, required in every COBOL program, describes the characteristics of the data to be processed by the object program.

This data can be divided into six major types:

1. Data contained in files, both input and output.

2. Data contained in a database and accessed through the Data Base Management System (DBMS).

3. Data to be sent to or received from the Message Control System (MCS).

4. Data initially stored as part of the program, as variables or constants. This can include constant data such as messages, tables of fixed values, and the like, or data developed during processing, that is, intermediate information such as partial arithmetic results.

5. Data in a subprogram that is passed from the program calling it.

6. Data to be printed in a report, and the format used to print such data.

To handle these types of data, the Data Division consists of the following sections:

1. The FILE SECTION, which describes the characteristics and the data formats for each file processed by the object program.

2. The SCHEMA SECTION, which names the sub-schema and schema that link a program or subprogram to the Data Base Management System (DBMS).

3. The COMMUNICATION SECTION, which defines the special data items that link a program or subprogram to the Message Control System (MCS).

4. The WORKING-STORAGE SECTION, which contains any fixed values and the working areas in which intermediate data can be stored.

5. The LINKAGE SECTION, which describes the data in a subprogram that is available from the calling program.

6. The REPORT SECTION, which describes the data and format of a report.

Unused sections of the Data Division can be omitted. However, the sections included must be in the following order:

FILE SECTION.
SCHEMA SECTION.
COMMUNICATION SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
REPORT SECTION.

4.1 FILE SECTION

In the File Section, the characteristics of each file to be processed are described by two types of entries.

The first type of entry, the file description, describes the physical aspects of the file. These aspects include:

1. How the logical data records of the file are physically grouped into blocks on the file medium.

2. The maximum length of a logical record, which cannot exceed 4095 characters.

3. Whether or not the file contains header and trailer labels and, if so, whether the format of these labels is standard or nonstandard.

4. The names of the records contained in the file.

5. The names of any reports in the file.

The second type of entry, the data description, describes the data formats of the logical records in the files.

The File Section begins with the section-header FILE SECTION. If present, it must be the first section in the DATA DIVISION.

4.2 SCHEMA SECTION

In the Schema Section, the names of the sub-schema and schema to be processed are specified by either an INVOKE statement or an ACCESS statement.

The Schema Section begins with the section-header (SCHEMA SECTION.) and must follow the File Section, if present.

If the installation does not include DBMS, the Schema Section cannot be used.

A description of the contents of the Schema Section can be found in the Data Base System Programmer's Procedures Manual.

4.3 COMMUNICATION SECTION

In the Communication Section, input and output communication-description entries are defined.

CD entries define records, called CD records, that contain special data items used to link the program to the Message Control System (MCS-10).

The Communication Section begins with the section-header COMMUNICATION SECTION. and must follow the File Section and precede the Report Section. The Communication Section must also follow the Schema Section if both are present.

If the installation does not include MCS, the Communication Section cannot be used.

Details of the Communication Section entries can be found in the Message Control System Programmer's Procedures Manual.

4.4 WORKING-STORAGE SECTION

The Working-Storage Section defines (1) data that is stored when the object program is loaded, and (2) areas used for intermediate results. The Working-Storage Section is similar to the File Section, except that the Working-Storage Section can contain level-77 items and cannot contain FD, SD, or RD entries.

The Working-Storage Section begins with the section-header WORKING-STORAGE SECTION.

The maximum size of a data item in WORKING STORAGE is 262,143 characters.

4.5 LINKAGE SECTION

The Linkage Section describes data available from a calling program and can appear only in a subprogram. The structure is the same as that of the Working-Storage Section with the following restrictions:

1. The VALUE clauses can only be used in condition-name entries.
2. The data-names used in the VALUE OF IDENTIFICATION (or ID), the VALUE OF DATE-WRITTEN, and the VALUE OF USER NUMBER cannot appear in this section.
3. The OCCURS clause with the DEPENDING phrase cannot be defined in this section.
4. The SYMBOLIC KEY and ACTUAL KEY data items cannot be defined in this section.
5. The data items in the FILE-LIMITS clause cannot be defined in this section.

Data described in the Linkage Section of a subprogram is not allocated storage space. Instead, at link-time, the link program sequentially equates the Linkage Section identifiers (listed in the USING clause of the ENTRY statement within the subprogram or in the USING clause of the PROCEDURE DIVISION header within the subprogram) to the calling program identifiers (listed in the USING clause of the CALL statement within the calling program). Thus, when the Procedure Division of a subprogram executes, references to the Linkage Section data refer instead to the calling program data.

Thus:

```
     CALLING PROGRAM              CALLED PROGRAM
            .                            .
            .                            .
            .                            .
     DATA DIVISION.               DATA DIVISION.
     FILE SECTION.                FILE SECTION.
     FD...                        LINKAGE SECTION
     01 MAIN...                   01 SUB...
     02 MAIN1...                  02 SUB1...
     02 MAIN2...                  02 SUB2...
            .                            .
            .                            .
            .                            .
     PROCEDURE DIVISION.          PROCEDURE DIVISION.
            .                     ENTRY ENTRPT USING SUB.
            .                            .
            .                            .
     CALL ENTRPT USING MAIN.             .
            .                     EXIT PROGRAM.
            .
            .
```

The identifier MAIN is defined in the File Section of the calling program; the identifier SUB is defined in the Linkage Section of the called program. When the Procedure Division of the called program executes, references to SUB refer instead to MAIN. See the COBOL-68 User's Guide for more information about subprograms.

Each 01- or 77-level item in the Linkage Section must have a unique name because it cannot be qualified. Also, each 01- and 77-level item must correspond to a word-aligned item of the same size or larger in the calling program. Word-aligned items start at the beginning of a computer word. All 01- and 77-level items fulfill this requirement; items that do not, can be made to do so by the SYNCHRONIZED LEFT statement.

4.6 REPORT SECTION

The Report Section defines reports by describing the physical appearance of the particular format and data rather than by specifying the procedure used to produce the report.

The data for a report can be read from a file or another part of the program or can be summed within the Report Section. The format of the report is given in the record description and report group entries in the Report Section.

The Report Section begins with the section-header REPORT SECTION., and must follow the File Section, the Working-Storage Section and the Linkage Section.

4.7 DATA DESCRIPTIONS

4.7.1 Elementary Items and Group Items

The basic user-defined datum in a COBOL program is called an elementary item; it can be referenced directly only as a unit. An elementary item can be associated with contiguous elementary items to form sets of data items called group items. Group items can be associated with other group items and/or elementary items to form more inclusive group items. Thus, an elementary item can be contained within one or more group items, and a group item can contain more than one elementary item.

4.7.2 Level Numbers

Level numbers indicate a hierarchy in which data items are ranged. The highest level is 01, which signifies that the data item is a record within a file named in an FD clause (or is a contiguous area in the WORKING-STORAGE SECTION). Level numbers of 02 through 49 indicate items that are subordinate to a 01-level data item. For example, an employee record can be described in the following manner.

```
01 EMPLOYEE-RECORD.
       02 NAME.
          03 FIRST-NAME PICTURE IS A(6).
          03 MIDDLE-INITIAL PICTURE IS A.
          03 LAST-NAME PICTURE IS A(20).
       02 BADGE-NUMBER PICTURE IS X(5).
       02 SALARY-CLASS PICTURE IS X(2).
```

Within a record description, the level numbers indicate which items are contained within higher-level items. That is, in the above example, the items that have a 03 level are subordinate to NAME, which has a 02 level, which is in turn subordinate to EMPLOYEE-RECORD, which has a 01 level. The example also shows elementary items (those that contain PICTURE clauses) contained within group items. In this example, EMPLOYEE-RECORD is a group item, NAME is a group item contained within a group item, and FIRST-NAME is an elementary item contained within the group item NAME. An item at a 01 level can be an elementary item as well as a group item as long as it is referenced as a unit. For example:

```
01 EMPLOYEE-RECORD PICTURE IS A(34).
```

shows the same record as above, but in this case the record is always operated on as a single entity.

Three other level numbers are available to the COBOL programmer: 77, 66, and 88.

Items with a level number of 77 are noncontiguous elementary items that are written only in the WORKING-STORAGE SECTION to define constant values and to store intermediate results.

Level-66 data items are those items that contain an explicitly specified portion of a record, or even the whole record. A data item at a level of 66 is used in a RENAMES clause to regroup items within a record. After a record is described, a level-66 item RENAMES a portion of that record. The level-66 data item can be a regrouping of the whole record, a group within the record, or a combination of group and elementary items. For example:

```
01 EMPLOYEE-RECORD
       02 NAME
          03 FIRST-NAME...
          03 MIDDLE-INITIAL...
          03 LAST-NAME...
       02 BADGE-NO...
       02 SALARY-CLASS...
       66 PERSONNEL-REC RENAMES NAME THRU BADGE-NO.
       66 PAY-REC RENAMES LAST-NAME THRU SALARY-CLASS.
```

When the level-66 item PAY-REC is referenced, the items LAST-NAME, BADGE-NO., and SALARY-CLASS are referenced as a unit. The programmer can thus regroup portions of a record for differing purposes.

Level-88 items are condition-names that cause a value or a range of values to be associated with a data item. The condition-name can then be used in place of the relation condition in conditional expressions in the PROCEDURE DIVISION. For example:

```
03 BADGE-NO...
       88  FIRST-BADGE VALUE IS "A0001".
       88  LAST-BADGE VALUE IS "Z9999".
```

In a comparison, the following statements would then be equivalent:

```
           Conditional Variable               Condition-Name

IF BADGE-NO IS EQUAL TO "A0001"...      IF FIRST-BADGE...
IF BADGE-NO IS EQUAL TO "Z9999"...      IF LAST-BADGE...
```

4.7.3 Records and Files

Records can be divided into two categories; those associated with a file and those not associated with a file. A file is the highest level of data organization in COBOL; it represents a collection of data records held on some external medium, that is, not wholly in real or virtual memory. Records not associated with a file are those values stored in the WORKING-STORAGE and LINKAGE SECTIONS or sum counters in the REPORT SECTION.

4.8 QUALIFICATION

Any data item that is to be referenced must be uniquely identified. This unique identification can be achieved by the assignment of a unique name to each item. However, in many applications this is tedious and inconvenient (1) because of the large number of names required, and (2) because items containing the same type of information in different records would have different names. Therefore, qualification is introduced to allow similar items and certain records to have identical names.

Qualification means giving enough information about the item to specify it uniquely. In COBOL, this information is the name of the group items containing it, in order of increasing inclusiveness. It is not necessary to name each group containing it, but only enough groups so that no other item with the same name as the original item could be identically qualified. It is also unnecessary to name each successively higher group containing the item until a unique qualification is made. Any set of names that uniquely describe the item can be used.

Example:

```
     01    RECORD-1.          01      RECORD-2.
       02      ITEM-1.          02      ITEM-2.
         03      SUB-ITEM.        03      SUB-ITEM.
           04      FIELD PIC X.     04      FIELD PIC X.
```

FIELD in the left-hand example can be referenced uniquely in any of the following ways:

```
     FIELD OF SUB-ITEM OF ITEM-1 IN RECORD-1.
     FIELD OF SUB-ITEM OF ITEM-1.
     FIELD OF SUB-ITEM IN RECORD-1.
     FIELD IN ITEM-1 OF RECORD-1.
     FIELD IN RECORD-1.
     FIELD IN ITEM-1.
```

The connectives OF and IN are equivalent and can be used interchangeably.

The only data items which need have unique names are level-77 items and records not associated with files, since they are not contained in any higher level data structure. Records associated with files can be qualified by the file name, as can any item contained within the record. File names must be unique.

Level-66 items can be qualified only (1) by the name of the record with which they are associated and (2) by the name of any file with which that record is associated.

4.9 SUBSCRIPTING AND INDEXING

It can sometimes be more convenient for you to specify a set of data values as a table rather than assigning a name to each element of the set. A table (or array) is a set of homogeneous items stored together in memory for use by the program. You define the table elements in the program by specifying an OCCURS clause in the description of a data item. The data item thus defined represents not one item but a set of items having the identical format. Subscripting and indexing are used to refer to one of the elements of the set. In DIGITAL COBOL-68, subscripting and indexing are identical in use and can be used interchangeably. However, the manner in which they are defined differs.

Subscripting is defined simply by the fact that an item has an OCCURS clause in its description. For example,

```
     01  RATE-TABLE.
         02 VOLUME OCCURS 25 TIMES.
```

describes VOLUME as 25 elements of RATE-TABLE. If you wish to refer to one of the elements of this set, you must qualify the data-name with a subscript. Thus, VOLUME(10) is the tenth element (or occurrence) of VOLUME. A subscript can be either an integer or a data-name to which an integer value has been assigned. Thus, when DIST has been assigned to value 10, VOLUME(DIST) is the same as VOLUME(10).

To specify indexing you must add the INDEXED BY option to the OCCURS clause. Thus,

```
     01  RATE-TABLE.
         02 VOLUME OCCURS 25 TIMES INDEXED BY IND.
```

defines VOLUME as 25 elements of the table and defines IND as the index by which each element of the table can be indexed; that is, VOLUME (IND) is an element in the table. The index-name IND is treated exactly like the data-name DIST because the compiler recognizes an index-name as being exactly the same as a data-name. An item defined as an index in an OCCURS clause has an implicit usage of INDEX, and is equivalent to a data item that is declared USAGE INDEX. However, this usage is included in DIGITAL COBOL for compatibility with other compilers because an item whose usage is INDEX (implicit or explicit) is treated as if its usage were COMPUTATIONAL. In fact, a data-name that is used as a subscript can be explicitly declared as USAGE INDEX; it can be treated as a COMPUTATIONAL data item by the compiler.

COBOL-68 tables can be one, two, or three dimensions. The number of dimensions is defined by the number of subscripts or indexes required to refer to an individual item. For example:

```
     C(1,3)
```

represents the item located in the first row and third column of a 2-dimensional table which is defined by the DATA DIVISION entries:

```
     01 TABLEA.
        02 ROW OCCURS 20 TIMES.
           03 COLUMN OCCURS 5 TIMES.
```

The subscript/index must be enclosed in parentheses and must appear in the PROCEDURE DIVISION statement where the subscripted/indexed data name is used. The subscript/index must appear after the data-name. A space between the data-name and the parentheses is optional. Multiple subscripts/indexes are separated by a comma or by a space. No spaces can appear immediately following the left parenthesis or immediately preceding the right parenthesis. When referring to elements in multi-dimensional tables, subscript/indexes are written from left to right in the order of major (subscript/index varying least rapidly), intermediate, and minor (subscript/index varying most rapidly). The major index corresponds to the item written with the smallest level-number, that is, the most inclusive item.

As an illustration, consider a table having a major element occurring ten times, an intermediate element occurring five times within each occurrence of the major element, and a minor element occurring three times within each intermediate element. The last major element of the table has a subscript of (10,1,1), while the final element of the table has a subscript of (10,5,3).

There are two forms of subscripting/indexing: direct and relative. Direct subscripting/indexing means that the subscript/index refers directly to the desired element. Relative subscripting/indexing means that the element of the table is referred to indirectly by a subscript/index to a data name to which an integer is added or subtracted. The form for direct subscript/indexing is shown in Figure 4-1.

$$\text{data-name} \left(\begin{Bmatrix} \text{subscript} \\ \text{index} \end{Bmatrix} \left[\begin{Bmatrix} \text{,subscript} \\ \text{,index} \end{Bmatrix} \right] \ldots \right)$$

MR-S-581-80

Figure 4-1 Direct Subscripting/Indexing

In relative subscripting/indexing, the subscript/index is followed by the operator plus (+) or minus (-) followed by an unsigned integer numeric literal. The operator plus (+) or minus (-) must be delimited by spaces. The subscript/index, the operator, and the numeric literal must follow the data-name and must be enclosed in parentheses. The form for relative subscripting/indexing is shown in Figure 4-2.

$$\text{data-name} \left(\begin{Bmatrix} \text{subscript} \\ \text{index} \end{Bmatrix} \begin{Bmatrix} + \\ - \end{Bmatrix} \text{integer} \left[\begin{Bmatrix} \text{,subscript} \\ \text{,index} \end{Bmatrix} \begin{Bmatrix} + \\ - \end{Bmatrix} \text{integer} \right] \ldots \right)$$

MR-S-582-80

Figure 4-2 Relative Subscripting/Indexing

When you use relative subscripting/indexing, the element of the table that you refer to is not the one to which the subscript/index refers, but the element to which the subscript/index plus or minus the integer refers. That is, if the item

```
     VOLUME (IND + 2)
```

is specified, and IND is set at 3, the fifth occurrence of VOLUME is referred to, not the third. However, the value of the subscript/index is not changed by relative subscripting/indexing; the value of IND remains 3.

You can also combine direct subscripting/indexing and relative subscripting in the same statement. For example, if you specify the following statement

```
     TABLE(IND, VOL + 3)
```

the first subscript value is the value of IND and the second subscript value is the value of VOL + 3.

When you need to qualify a table element for uniqueness, you should use the format for direct subscripting/indexing shown in Figure 4-3.

$$\text{data-name} \quad \left[\left\{ \begin{matrix} \underline{\text{OF}} \\ \underline{\text{IN}} \end{matrix} \right\} \quad \text{data-name-1} \right] \quad \ldots \quad \left(\left\{ \begin{matrix} \text{subscript} \\ \text{index} \end{matrix} \right\} \left[\left\{ \begin{matrix} \text{,subscript} \\ \text{,index} \end{matrix} \right\} \right] \quad \ldots \right)$$

MR-S-583-80

Figure 4-3 Qualified Direct Subscripting/Indexing

For example, to refer to ANAME in the following example:

```
01 AREC1.
   02 AGROUP1 OCCURS 5.
      03 ASUBGROUP1 OCCURS 10.
         04 ANAME PIC X(5) OCCURS 20.
```

you could specify the following:

```
ANAME (I,J,4)
```

NOTE

Subscripts can not be subscripted.

4.10 FILE DESCRIPTION (FD)

Function

The File Description (FD) furnishes information concerning the physical structure, identification, and record names pertaining to a given file.

General Format

```
FD  file-name
   [ BLOCK CONTAINS [integer-1 TO] integer-2 { RECORD(S)  } ]
                                             { CHARACTERS }

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

   [ LABEL { RECORD IS   } { STANDARD                                  } ]
   [       { RECORDS ARE } { OMITTED                                   } ]
   [                       { record-name-1  [,record-name-2]  ...      } ]

   [ { REPORT IS   }  report-name-1   [,report-name-2...] ]
   [ { REPORTS ARE }                                      ]

   [ VALUE  OF  [ { ID             }  IS  { data-name-2 } ]
   [            [ { IDENTIFICATION }      { literal-1   } ]

                [ DATE-WRITTEN  IS  { data-name-2 } ]
                [                   { literal-2   } ]

                [ USER-NUMBER  IS  { data-name-3           } ] ]
                [                  { literal-3,literal-4   } ] ]

   [ DATA { RECORD IS   }  record-name-3   [,record-name-4]   ... ]
   [      { RECORDS ARE }                                         ]
```

MR-S-989-81

FILE DESCRIPTION (FD) (Cont.)

```
┌                          ┐
│           ┌  ( ASCII           ) ┐│
│           │  | SIXBIT          | ││
│           │  | BINARY          | ││
│ RECORDING │ MODE IS < F        > ││ .
│           │  | V               | ││
│           │  | STANDARD-ASCII  | ││
│           └  ( STANDARD ASCII  ) ┘│
└                          ┘
```

MR-S-1000-81

The clauses shown in the General Format appear in alphabetical order on the following pages.

Technical Notes

1. An FD entry must be present for each file-name selected in the FILE-CONTROL paragraph of the Environment Division.
2. All semicolon and commas are optional. The entire FD entry must terminate with a period.
3. The clauses can appear in any order within the File Description entry.
4. The ability to place the RECORDING MODE clause in the FD has been provided for compatibility with other manufacturers. If you place the RECORDING MODE clause for a file in the FD, you cannot also place it in the FILE-CONTROL paragraph for that file in the ENVIRONMENT DIVISION. Also, if you wish to use the RECORDING DENSITY and RECORDING PARITY clauses, you must put them in the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION, even if the RECORDING MODE clause is in the FD. The description of the RECORDING MODE clause can be found in Chapter 3 with the full description of the ENVIRONMENT DIVISION.
5. The maximum number of files that can be open at one time is 16.

NOTE

ISAM files count as two files: one index (.IDX) file and one data (.IDA) file.

4.10.1 BLOCK CONTAINS

Function

The BLOCK CONTAINS clause specifies the size of a logical block.

General Format

$$\left[\ \underline{\text{BLOCK}}\ \text{CONTAINS}\ \ \left[\text{integer-1}\ \ \underline{\text{TO}}\right]\ \ \text{integer-2}\ \left\{\begin{matrix}\underline{\text{RECORD(S)}}\\ \underline{\text{CHARACTERS}}\end{matrix}\right\}\ \right]$$

MR-S-1001-81

Technical Notes

1. You must specify the BLOCK CONTAINS clause if you want the file to be organized into logical blocks. If you do not specify this clause, or if you specify integer-2 to be 0, then all records are packed in the file with no empty space between the records. The file is then considered to be "unblocked" or "blocked zero". However, if you use magnetic tape and have used the RECORDING MODE clause to specify that the recording mode as V or F, or standard ASCII, the default is a blocked file with a blocking factor of one.

2. If the CHARACTERS option is used, the logical block size is specified in terms of the number of character positions required to contain the record. If the recording mode is ASCII (that is, all records for the file are described, explicitly or implicitly, as USAGE DISPLAY-7), it is assumed that the size is specified in terms of DISPLAY-7 characters. If the recording mode is SIXBIT (that is, the records for the file are all described, explicitly or implicitly, as DISPLAY-6) it is assumed that the size is specified in terms of SIXBIT characters. If the recording mode is F or V (that is, the data is recorded on the medium as EBCDIC characters), it is assumed that the size is specified in terms of EBCDIC characters, either fixed- or variable-length. When variable-length EBCDIC records are used (i.e., the recording mode is V), the number of records in a block is also variable. If the blocking factor is not zero, the number of records in a block is determined by dividing the block size in characters by the number of characters in the longest record as specified by the FD statement. For example, if the FD statement specifies a maximum record length of 248 characters and the BLOCK CONTAINS 2400 CHARACTERS clause is used, the number of records in a block are 9.

3. Integer-1 and integer-2 must be positive integers. If only integer-2 is specified, it represents the exact size of the logical block. If both integer-1 and integer-2 are given, integer-1 is ignored and integer-2 is used as the blocking factor.

4. Files whose access modes are RANDOM or INDEXED must have a nonzero blocking factor. Files whose access mode is sequential and opened for I/O should have a nonzero blocking factor. If not, the compiler will calculate one.

DATA RECORD

4.10.2 DATA RECORD

Function

The DATA RECORD clause cross references the record-name with its associated file.

General Format

DATA $\left\{\begin{matrix}\underline{\text{RECORD}}\ \text{IS}\\ \underline{\text{RECORDS}}\ \text{ARE}\end{matrix}\right\}$ record-name-1 [,record-name-2] ...

MR-S-1002-81

Technical Notes

1. This clause is optional because all records not associated with a LABEL RECORDS clause are assumed to be data records.

2. Both record-name-1 and record-name-2 must be the names given in 01-level data entries subordinate to this FD. The presence of more than one such record-name indicates that the file contains more than one type of data record. These records can have different descriptions. The order in which they are listed is not significant.

3. All records within a file share the same area.

4.10.3 FD file-name

Function

The FD file-name clause identifies the file to which this file description entry and the subsequent record descriptions relate.

General Format

```
FD file-name
```

Technical Notes

1. This entry must begin each file description.
2. The file-name must appear in a SELECT statement in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

LABEL RECORD

4.10.4 LABEL RECORD

Function

The LABEL RECORD clause specifies whether or not labels are present on the file and, if so, identifies the format of the labels.

General Format

```
[        { RECORD  IS  }  { OMITTED                                   } ]
[ LABEL  {             }  { STANDARD                                  } ]
[        { RECORDS ARE }  { record-name-1  [,record-name-2]  ...      } ]
```

MR-S-1003-81

Technical Notes

1. If the clause is omitted, LABEL RECORDS ARE STANDARD is assumed.

2. The OMITTED option is used when the file has no header or trailer labels.

3. The STANDARD option is used when the file has header and trailer labels that conform to the DECsystem-10 standard format. If this clause is used for files on disk or DECtape, LABEL RECORDS ARE STANDARD must be specified. See the VALUE OF IDENTIFICATION clause for the association between the label and the filename on disk or DECtape.

   The standard label for DECtape and random-access devices is the directory block used by the monitor. For magnetic tape, if the file is recorded in SIXBIT, the standard label is 78 SIXBIT characters in length and is written in a separate physical record from the data, with the same recording mode as the data. If the recording mode is ASCII, the label contains 78 ASCII characters, plus carriage return and line feed, for a total of 80 characters. Table 4-1 shows the contents of each character in a standard label for nonrandom-access devices.

   Magnetic tapes are the only devices with ending labels. Each ending label is preceded by and followed by an end-of-file mark.

4. The record-name option is used when the file labels do not conform to the system standard format. The record-names must appear as the names of record description (level-01) subordinate to this FD; the record-names must not appear in a DATA RECORDS clause. When a file is opened, the beginning non-standard label is read (as input) or written (as output) automatically by the I-O routines. If the file is being opened for output, the data for the record must be supplied by a USE procedure in the DECLARATIVES (see Chapter 5). If the file is being opened for input, no checks are made by the I-O routines to determine the validity of the label; you can program any checks in a USE procedure.

LABEL RECORD (Cont.)

The presense of TOPS-10 or TOPS-20 system labels on a magnetic tape causes the system to handle tape volume processing normally done by LIBOL and overrides the COBOL labeling described above. Thus, a magnetic tape only has TOPS-10 or TOPS-20 system labels or LIBOL created labels, but never both.

5. Files whose recording mode is F or V (fixed- or variable-length EBCDIC), must have LABELS RECORDS ARE OMITTED if they are on magnetic tape. If they are on disk or DECtape, they are assumed to have DECsystem-10 standard labels. The record-name option cannot be used for EBCDIC files.

Table 4-1
Standard Label for Nonrandom-Access Media

Characters	Contents

1-4	HDR1 = Beginning File.
	EOF1 = Ending file.
	EOV1 = Ending reel.

5-13	Value of identification.
14-21	Always spaces.
22-27	Not used.
28-31	Reel number. The first reel is always 0001.
32-41	Not used.
42-47	Creation date; two characters each for the year, month, and day, respectively.
48-78	Not used.
79-80	Carriage-return/line-feed if file is ASCII. (Note that this is on the label only; it is not kept internally.)

RECORD CONTAINS

4.10.5 RECORD CONTAINS

Function

The RECORD CONTAINS clause specifies the size of the data records in this file.

General Format

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

MR-S-1004-81

Technical Notes

1. Because the size of each data record is completely defined by its record description entry, this clause is never required. However, if you use it, it replaces the record description entry in setting the size of the record, and the following rules must be observed.

2. Integer-1 and integer-2 must be positive integers. Integer-2 can not be less than the size of the largest record but cannot exceed 4095, which is the limit on the size of a record.

3. The data record size is specified in terms of the number of character positions required to contain the record.

4. The maximum size of a record in an FD is 4095 characters.

5. This clause is ignored if the FD contains a REPORT clause and there is no data record description. In this case, the record size defaults to 132 characters.

4.10.6 REPORT

Function

The REPORT clause specifies the name of each report that is associated with the file.

General Format

$$\left[\left\{\begin{matrix}\underline{\text{REPORT}}\ \text{IS}\\ \underline{\text{REPORTS}}\ \text{ARE}\end{matrix}\right\}\ \text{report-name-1}\ \ [\ ,\text{report-name-2}\ ]\ \ldots\right]$$

MR-S-1005-81

Technical Notes

1. This clause is optional; it is used only when Report-Writer statements cause output to be written on the file.

2. Report-name-1 and report-name-2 must be the names of Report Descriptor items in the REPORT SECTION.

3. If this clause is used, the data record description can be omitted because the name of the data record is not referred to directly in the PROCEDURE DIVISION. When the data record description is omitted, the compiler automatically assumes a 132-character record.

SD file-name

4.10.7 SD file-name

Function

The SD file-name clause identifies the sort to which this file description entry and the subsequent record description relate.

General Format

$$\underline{\text{SD}}\ \text{file-name}\ \left[\underline{\text{DATA}}\left\{\begin{array}{l}\underline{\text{RECORD}}\ \text{IS}\\ \underline{\text{RECORDS}}\ \text{ARE}\end{array}\right\}\ \text{record-name-1}\ [,\text{record-name-2}]\ \ldots\right]$$

$$\left[\underline{\text{RECORD}}\ \text{CONTAINS}\ [\text{integer-1}\ \underline{\text{TO}}]\ \text{integer-2}\ \text{CHARACTERS}\right]\ \underline{.}$$

MR-S-1006-81

Technical Notes

1. The SD entry must begin each sort file description.
2. The file-name must appear in a SELECT statement in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.
3. The DATA RECORD and RECORD CONTAINS clauses are the only descriptive clauses allowed.

VALUE OF IDENTIFICATION

4.10.8 VALUE OF IDENTIFICATION/DATE-WRITTEN/USER-NUMBER

Function

The VALUE OF IDENTIFICATION clause provides specific data for an item within the label records associated with a file.

General Format

```
[VALUE OF [{ID            } IS {data-name-1}]
          [{IDENTIFICATION}    {literal-1  }]

          [DATA-WRITTEN IS {data-name-2}]
          [                {literal-2  }]

          [USER-NUMBER IS {data-name-3          }]]
          [               {literal-3,literal-4  }]]
                                      MR-S-1007-81
```

Technical Notes

1. The VALUE OF IDENTIFICATION clause is required only if label records are standard; it is ignored in all other cases. The VALUE OF DATE-WRITTEN and the VALUE OF USER-NUMBER are always optional.

2. The three clauses can be written in any order, but only one of each can be specified for a file.

3. IDENTIFICATION represents the file-name and extension of a file with standard labels. If a data-name is specified, it must be associated with a DISPLAY-6, DISPLAY-7, or DISPLAY-9 data item nine characters in length. If a literal is specified, it must be a alphanumeric literal nine characters in length. The first six characters are taken as the file-name, and last three characters are taken as the extension. The programmer must provide spaces as required to conform to this convention. The literal cannot consist exclusively of spaces. The period which the system prints between the file-name and the extension must not be included in the VALUE OF IDENTIFICATION clause.

   Examples:

   a. VALUE OF IDENTIFICATION IS "COST TST"

   b. VALUE OF IDENTIFICATION IS FILE-1-NAME

   ```
           .
           .
      (WORKING-STORAGE SECTION.)
           .
           .
      77 FILE-1-NAME PICTURE IS X(9).
   ```

VALUE OF IDENTIFICATION (Cont.)

4. DATE-WRITTEN represents the date that a magnetic tape file (with STANDARD labels) was written. If a data-name is specified, it must be associated with a DISPLAY-6, DISPLAY-7 or DISPLAY-9 data item six characters in length. If a literal is specified, it must be a alphanumeric literal six characters in length. The first two characters are taken as year, the next two as month, and the last two as day. The DATE-WRITTEN clause is ignored when the file is OPENed for output; instead, the current date is used.

   Examples:

   a. VALUE OF IDENTIFICATION IS "RANDOMXYZ", DATE-WRITTEN IS 760112

   b. VALUE OF IDENTIFICATION IS "DATA   ", DATE-WRITTEN IS FILE-1-DATE

```
              .
              .

   (WORKING-STORAGE SECTION.)
   77 FILE-1-DATE PICTURE IS 9(6).
```

5. USER-NUMBER represents the project-programmer number of the owner of a disk file; it is ignored for all other devices. Data-name-3 must be a COMPUTATIONAL item of 10 or fewer digits in which the project-programmer number is stored. Literal-3 and literal-4 are numeric literals of six or fewer digits that are treated as octal. Literal-3 is the project number and literal-4 is the programmer number.

6. For input files the VALUEs specified are checked against the file when it is opened. ISAM files are checked as soon as your program is run. For output files, the VALUE OF IDENTIFICATION is written when the file is opened. If the specified values do not match a file on the selected medium, a run-time error message is issued.

7. If the access mode is INDEXED and data-name-1 is used in the VALUE OF IDENTIFICATION clause, data-name-1 must contain the filename and extension of the index-file for the indexed sequential file being referenced. The contents of data-name-1 can not be altered during program execution. You need not specify the identification for the data file of an indexed sequential file because this identification is stored in the index file.

8. If data-name-3 is used to represent the project-programmer number, you must be aware that the value of data-name-3 is treated as decimal, even though the project-programmer number is octal. The data-name-3 value is translated from decimal to binary by the COBOL conversion routine. Thus, the project-programmer is not accurate unless you provide a conversion routine in your program to convert your octal project-programmer number to its decimal equivalent so that it is converted to the correct binary number. The following example is a suggested method for performing the conversion.

VALUE OF IDENTIFICATION (Cont.)

```
77  ERR-FLAG            PIC 9,                  USAGE COMP.
77  HALF-NUM,           PIC S9(7),              USAGE COMP.
77  OCTAL-PPN,          PIC S9(10),             USAGE COMP.
77  DIGIT,              PIC 9.
01  PP-NUMBER.
    02 PROJ-NUMBER,     PIC 9(6).
    02 PROG-NUMBER,     PIC 9(6).
    02 EITHER-NUM,      PIC 9(6).
    02 X REDEFINES EITHER-NUM.
       03 PP-DIGIT,     PIC 9, OCCURS 6 TIMES, INDEXED BY I.
             .
             .
             .
    ACCEPT PROJ-NUMBER, PROG-NUMBER.
    SET ERR-FLAG TO ZERO.
    MOVE PROJ-NUMBER TO EITHER-NUM.
    MOVE ZERO TO HALF-NUM.
    PERFORM CONVERT VARYING I FROM 1 BY 1 UNTIL I>6.
    IF ERR-FLAG IS NOT = 0 GO TO OCTAL-ERROR.
    COMPUTE OCTAL-PPN = HALF-NUM * 262144.
    MOVE PROG-NUMBER TO EITHER-NUM.
    MOVE ZERO TO HALF-NUM.
    PERFORM CONVERT VARYING I FROM 1 BY 1 UNTIL I>6.
    IF ERR-FLAG IS NOT = 0 GO TO OCTAL-ERROR.
    COMPUTE OCTAL-PPN = OCTAL-PPN + HALF-NUM.

.
.
.

CONVERT.

    IF PP-DIGIT (I) = 8 OR 9, SET ERR-FLAG UP BY 1.
    COMPUTE HALF-NUM = 8 *HALF-NUM + PP-DIGIT (I).

*   THIS ROUTINE INVALID FOR PROJECT NUMBERS LARGER THAN
    77777.
```

If the access mode is INDEXED and data-name-3 is used to represent the project-programmer number, the following rules must be observed:

a. Data-name-3 must have a value that is the decimal equivalent of an octal project-programmer number, and that project-programmer number must contain a file with the name used in the VALUE OF IDENTIFICATION clause.

b. Data-name-3 can be altered during program execution only if all files referenced have identical parameters.

c. If several files are read through the same File Description, data-name-3 should point to the file with the largest number of levels of index (this is usually the largest file).

9. None of the data-names in the VALUE clauses can appear in the LINKAGE SECTION.

RECORD DESCRIPTIONS

4.11 RECORD DESCRIPTIONS

Following the FD for a file, a record description is given for each different record format in the file. A record description begins with a level-01 entry:

```
     01  data-name
```

A complete record description can be as simple as

```
     01  data-name PICTURE picture-string.
```

or it can be more complex, where the 01-level is followed by a long series of data description entries of varying hierarchies that describe various portions and subportions of the record. A 01-level data-name in the File Section cannot be explicitly redefined (using the REDEFINES clause). However, because a file has only one record area, if more than one data-name is specified, they implicitly redefine the first data-name. Also, if the additional data-names have usages different from that of the first data-name, the last usage given is used as the usage in determining the usage mode of the file if it is necessary to use a default.

4.11.1 **Record Concepts**

A record description consists of a set of data description entries which describe a particular logical record. Each data description entry consists of a level-number followed by a data-name (or FILLER) which is followed, as required, by a series of descriptive clauses.

The general format of a data description entry follows.

4.11.2 DATA DESCRIPTION ENTRY

Function

A data description entry describes a particular item of data.

General Format

```
level-number {data-name-1} [REDEFINES data-name-2] [{PICTURE} IS picture-string]
             {FILLER     }                         [{PIC    }                  ]

 [               ( COMPUTATIONAL   ) ]
 [               ( COMP            ) ]
 [               ( COMPUTATIONAL-1 ) ]
 [               ( COMP-1          ) ]
 [               ( COMPUTATIONAL-3 ) ]
 [               ( COMP-3          ) ]
 [ [USAGE IS]   <  DISPLAY          > ]  [{SYNCHRONIZED} {LEFT }]
 [               ( DISPLAY-6       ) ]  [{SYNC        } {RIGHT}]
 [               ( DISPLAY-7       ) ]
 [               ( DISPLAY-9       ) ]
 [               ( INDEX           ) ]
 [               ( DATABASE-KEY    ) ]
 [               ( DBKEY           ) ]

 [{JUSTIFIED} {RIGHT}]  [BLANK WHEN ZERO]  [VALUE IS literal-1]
 [{JUST     } {LEFT }]

 [
 [ OCCURS [integer-1 TO] integer-2 TIMES [DEPENDING ON data-name-1]
 [
        [{ASCENDING } KEY IS data-name-2 [,data-name-3] ... ] ...
        [{DESCENDING}                                       ]
                                                          ]
         [INDEXED BY index-name-1 [,index-name-2] ...]    ] .
                                                          ]

66 data-name-1 RENAMES data-name-2 [THRU data-name-3] .

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

         [,literal-3 [THRU literal-4]] ... .
```

MR-S-1008-81

DATA DESCRIPTION ENTRY (Cont.)

The clauses shown in the General Format appear in alphabetical order on the following pages.

Technical Notes

1. Each data description entry must be terminated by a period. All semicolons and commas are optional.

2. The clauses can appear in any order, with one exception: the REDEFINES clause, when used, must immediately follow the data-name being redefined.

3. The VALUE clause must not appear in a data description entry which also contains an OCCURS clause, or in an entry which is subordinate to an entry containing an OCCURS clause. The latter part of this rule does not apply to condition-name (level-88) entries.

4. The PICTURE clause must be specified for every elementary item, except a USAGE INDEX, COMP-1, DATABASE-KEY, or DBKEY.

5. The clauses SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO can be specified only at the elementary level.

4.11.2.1 BLANK WHEN ZERO

Function

The BLANK WHEN ZERO clause causes the blanking of an item when its value is zero.

General Format

$$\left[\ \underline{\text{BLANK}}\ \ \text{WHEN}\ \ \underline{\text{ZERO}}\ \right]$$

MR-S-1009-81

Technical Notes

1. When the BLANK WHEN ZERO option is used and the item is zero, the item is set to blanks.

2. BLANK WHEN ZERO can be specified only at the elementary level and only for numeric or numeric-edited items whose usage is DISPLAY-6, DISPLAY-7, or DISPLAY-9.

3. More comprehensive editing features are available in the PICTURE clause. For example, if a PICTURE clause appears in the same data description entry and contains the zero suppression symbol * (zero suppress and replace with *), the field is replaced with * when the item is given a zero value (see Section 4.11.2.7, PICTURE). The only exception is fields containing a decimal point, in which case the decimal point is not replaced.

Condition-name (level–88)

4.11.2.2 Condition-name (level-88)

Function

To assign a name to a value or range of values of the associated data item.

General Format

$$88\ \text{condition-name} \left\{\begin{matrix}\underline{\text{VALUE}}\ \text{IS}\\ \underline{\text{VALUES}}\ \text{ARE}\end{matrix}\right\}\ \text{literal-1}\ \left[\ \underline{\text{THRU}}\ \text{literal-2}\ \right]$$

$$\left[\ ,\text{literal-3}\ \left[\ \underline{\text{THRU}}\ \text{literal-4}\ \right]\ \right]\ \ldots\ \underline{.}$$

MR-S-1010-81

Technical Notes

1. Each condition-name requires a separate level-88 entry. This entry contains the name assigned to the condition, and the value or values associated with that condition. Condition-name entries must immediately follow the data description entry with which the condition-name is to be associated.

2. A condition-name entry can be associated with an elementary or group item except:

   a. another condition-name entry, or

   b. a level-66 item.

3. Some examples of possible level-88 entries are given below:

   a.
   ```
   05  B-FIELD PICTURE IS 99.
       88 B1 VALUE IS 3.
       88 B2 VALUES ARE 50 THRU 69.
       88 B3 VALUES ARE 20, 25, 28, 31 THRU 37.
       88 B4 VALUES ARE 70 THRU 75, 80 THRU 85, 90 THRU 95.
   ```

   b.
   ```
   02  C-FIELD PICTURE IS XXX.
       88 C-YES VALUE IS "YES".
       88 C-NO VALUE IS "NO ".
   ```

4. The data item with which the condition-name is associated is called a conditional variable. A conditional variable can be used to qualify any of its condition-names. If references to a conditional variable require indexing, subscripting, or qualification, then reference to its associated condition-names also require the same combination of indexing, subcripting, or qualification.

Condition-name (level-88) (Cont.)

5. A condition-name is used in conditional expressions as an abbreviation for the related condition. Thus, if the above DATA DIVISION entries (NOTE c) are used, the statements in each pair below are functionally equivalent.

	Relational Expression	Condition-Name
a.	IF B-FIELD IS EQUAL TO 3....	IF B1....
b.	IF B-FIELD IS GREATER THAN 49 AND LESS THAN 70....	IF B2....
c.	IF B-FIELD IS EQUAL TO 20 OR EQUAL TO 25 OR EQUAL TO 28 OR GREATER THAN 30 AND LESS THAN 38....	IF B3....
d.	IF B-FIELD IS GREATER THAN 69 AND LESS THAN 76 OR GREATER THAN 79 AND LESS THAN 86 OR GREATER THAN 89 AND LESS THAN 96....	IF B4....
e.	IF C-FIELD IS EQUAL TO "YES"..	IF C-YES

6. Literal-1 must always be less than literal-2, and literal-3 less than literal-4. The values given must always be within the range allowed by the format given for the conditional variable. For example, any condition-name values given for a conditional variable with a PICTURE of PP999 must be in the range of .00000 to .00999. (See Note 10 under PICTURE in this chapter for the meaning of P in a picture-string.)

4.11.2.3 data-name/FILLER

Function

A data-name specifies the name of the data being described. The word FILLER specifies an unreferenced portion of the logical record.

General Format

$$\text{level-number} \left\{ \begin{array}{l} \text{data-name} \\ \underline{\text{FILLER}} \end{array} \right\}$$

MR-S-1011-81

Technical Notes

1. A data-name or the word FILLER must immediately follow the level-number in each data description entry.

2. A data-name must be composed of a combination of the characters A through Z, 0 through 9, and the hyphen. It must contain at least 1 alphabetic character and must not exceed 30 characters in length. It must not duplicate a COBOL reserved word. Refer to Section 1.2.3.2, User-Created Words, for further information.

3. The key word FILLER is used to name an unreferenced item in a record (that is, an item to which the programmer has no reason for assigning a unique name). A FILLER item cannot, under any circumstances, be referenced directly in a PROCEDURE DIVISION statement. However, it can be indirectly referenced by referring to a group-level item of which the FILLER item is a part. FILLER can be used at any level, including the 01 level.

4.11.2.4 JUSTIFIED

Function

The JUSTIFIED clause specifies nonstandard positioning of data within a receiving data item.

General Format

$$\left[\begin{Bmatrix}\underline{\text{JUSTIFIED}}\\ \underline{\text{JUST}}\end{Bmatrix}\begin{Bmatrix}\text{RIGHT}\\ \underline{\text{LEFT}}\end{Bmatrix}\right]$$

MR-S-1012-81

Technical Notes

1. The JUSTIFIED clause cannot be specified at a group level or for numeric edited items.

2. If neither RIGHT nor LEFT is specified, RIGHT is assumed.

3. An item subordinate to one containing a VALUE clause cannot be JUSTIFIED.

4. DISPLAY-6, DISPLAY-7 and DISPLAY-9 items can be JUSTIFIED.

5. The standard rules for positioning data within an elementary data item are as follows:

   a. Receiving data item described as numeric or numeric-edited (see definition in Notes 6 and 9 under PICTURE in this chapter). A numeric or numeric-edited item is justified according to the following rules, thus the JUSTIFIED clause cannot be used.

      The data is aligned by decimal point and is moved to the receiving character positions with zero fill or truncation on either end as required.

      If an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately following its rightmost character, and the sending data is aligned according to this decimal point.

   b. Receiving data item described as alphanumeric (other than numeric edited) or alphabetic (see definition in Notes 5 and 7 under PICTURE in this chapter).

      The data is moved to the receiving character positions and aligned at the leftmost character position with space fill or truncation at the right end as required.

JUSTIFIED (Cont.)

6. When a receiving item is described as JUSTIFIED LEFT, positioning occurs as in 4a above.

7. When a receiving data item is described with the JUSTIFIED RIGHT clause and is larger than the sending data item, the data is aligned at the rightmost character position in the receiving item with space fill at the left end.

   When a receiving data item is described with the JUSTIFIED RIGHT clause and is smaller than the sending data item, the data is aligned at the right most character position in the receiving item with truncation at the left end.

Examples are given below:

```
     03  ITEM-A PICTURE IS
         X(8) VALUE IS "ABCDEFGH".

     03  ITEM-B PICTURE IS
         X(4) VALUE IS "WXYZ".

     03  ITEM-C PICTURE IS X(6).

     03  ITEM-D PICTURE IS X(6).
         JUSTIFIED RIGHT.
```

```
                                   Contents of Receiving Field

     MOVE ITEM-A TO ITEM-C.        A B C D E F

     MOVE ITEM-A TO ITEM-D.        C D E F G H

     MOVE ITEM-B TO ITEM-C.        W X Y Z

     MOVE ITEM-B TO ITEM-D.            W X Y Z
```

4.11.2.5 level-number

Function

The level-number shows the hierarchy of data within a logical record. In addition, special level-numbers are used for condition-names (level-88), noncontiguous WORKING-STORAGE items (level-77), and the RENAMES clause (level-66).

General Format

$$\text{level-number} \left\{ \begin{array}{l} \text{data-name} \\ \underline{\text{FILLER}} \end{array} \right\}$$

MR-S-1013-81

Technical Notes

1. A level-number is required as the first element in each data description entry.

2. Level-numbers can be placed anywhere on the source line, at or after margin A.

3. Level-number 88 is described under "condition-name (level-88)", and level-number 66 is described under "RENAMES (level-66)", both in this section.

4. A further description of level-numbers and data hierarchy can be found in the introduction to this chapter.

OCCURS

4.11.2.6 OCCURS

Function

The OCCURS clause eliminates the need for separate entries for repeated data, and supplies information required for the application of subscripts and indexes.

General Format

```
[OCCURS  {integer-1  TO     integer-2  TIMES  DEPENDING ON data-name-1}
         {integer-3  TIMES                                             }

   [{ASCENDING }  KEY  IS  data-name-2  [,data-name-3]   ... ] ...
    {DESCENDING}

  [INDEXED BY  index-name-1   [,index-name-2]   ... ]]
                                                       MR-S-1014-81
```

Technical Notes

1. This clause cannot be specified in a data description entry that has a 66 or 88 level-number, or in one that contains a VALUE clause.

2. The OCCURS clause is used to define tables or other homogeneous sets of repeated data. Whenever this clause is used, the associated data-name and any subordinate data-names must always be subscripted or indexed when used in all PROCEDURE DIVISION statements.

3. All clauses given in a data description that includes an OCCURS clause apply to each repetition of the item.

4. The integers must be positive. If integer-1 is specified, it must have a value less than integer-2. No value of a subscript can exceed integer-2; in addition, if the DEPENDING option is specified, no subscript can exceed the value of data-name-1 at the time of subscripting.

5. The value of data-name-1 is the count of the number of occurrences of the item described by the OCCURS clause; its value must not exceed integer-2.

6. If the DEPENDING option is specified, the integer-1 TO integer-2 phrase must be included. Data-name-1 must be USAGE INDEX or USAGE COMP of 10 digits or less with no scaling or decimal places. It cannot be subscripted or appear in the LINKAGE SECTION.

OCCURS (Cont.)

7. The KEY IS option indicates that the repeated data has been sorted by you into either ascending or descending order according to the values associated with data-name-2, data-name-3, and so forth. The data-names are listed in order of decreasing significance.

8. Data-name-2 must be either the name of the entry containing the OCCURS clause, or the name of an entry subordinate to the entry containing the OCCURS clause. Data-name-3, etc., must be the name of an entry subordinate to the group item that is the subject of this entry.

9. An index-name defined in a OCCURS clause must not be defined elsewhere; its appearance in the INDEXED option is its only definition. There can be no items of the same name defined elsewhere. The USAGE of each index-name is assumed to be INDEX.

10. Subscripting and indexing are described in the introduction to this chapter.

11. The maximum number of OCCURS for a single data item is 32,767.

PICTURE

4.11.2.7 PICTURE

Function

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

General Format

$$\left[\begin{Bmatrix}\underline{\text{PICTURE}}\\ \underline{\text{PIC}}\end{Bmatrix}\ \text{IS picture-string}\right]$$

MR-S-1015-81

Technical Notes

1. A PICTURE clause can be used only at the elementary level. It can not be used with an item described as USAGE INDEX or COMP-1.

2. A picture-string consists of certain allowable combinations of characters in the COBOL character set used as symbols. These symbols are as follows:

   a. Symbols representing data characters

      9 represents a numeric character (0 through 9)
      A represents an alphabetic character (A through Z, and the space)
      X represents an alphanumeric character (any allowable character)

   b. Symbols representing arithmetic signs and assumed decimal point positioning

      V represents the position of the assumed decimal point
      P represents an assumed decimal point scaling position
      S represents the presence of an arithmetic sign

   c. Symbols representing zero suppression operations

      Z represents standard zero suppression (replacement of leading zeros by spaces)
      * represents check protection (replacement of leading zeros by asterisks)

PICTURE (Cont.)

d. Symbols representing insertion characters

$ represents a dollar sign (this sign floats from left to right and replaces the rightmost leading zero when more than one $ apears)[1]
, represents an insertion comma[2]
. represents an actual decimal point[2]
B represents an insertion blank
0 represents an insertion zero

e. Symbols representing editing sign-control symbols

+ represents an editing plus sign
- represents an editing minus sign
CR represents an editing Credit symbol
DB represents an editing Debit symbol

The plus and minus signs (+ and -) float when more than one appear, and replace the rightmost leading zeroes.

f. Consecutive repetitions of a picture-symbol can be abbreviated to the symbol followed by (n), where n indicates the number of occurrences.

3. A maximum number of 30 symbols can appear in a picture-string. Note that the number of symbols in a picture-string and size of the item represented are not necessarily the same. There are two reasons for this discrepancy. First, the abbreviated form for indicating consecutive repetitions of a symbol can result in fewer symbols in the picture-string than character positions in the item being described. For example, a data item having 40 alphanumeric character positions can be described by a picture-string of only 5 symbols:

```
PICTURE IS X(40).
```

The second reason is that some symbols are not counted when calculating the size of the data item being described. These symbols include the V (assumed decimal point), P (decimal point scaling position), and S (arithmetic sign); these symbols do not represent actual physical character positions within the data item. For example, the character-string

```
S999V99
```

represents a 5-position data item.

Other size restrictions for numeric and numeric edited items are given under the appropriate headings below.

[1] If the CURRENCY SIGN IS clause appears in the SPECIAL-NAMES paragraph, the symbol specified by the literal must be used in all instances in place of the $.

[2] If the DECIMAL-POINT IS COMMA clause appears in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point are reversed.

PICTURE (Cont.)

4. There are five categories of data that can be described with a PICTURE clause: alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited. A description of each category is given in the notes below.

5. Definition of an Alphabetic Item

   a. Its picture-string can contain only the symbol A.

   b. It can contain only the 26 letters of the alphabet and the space.

6. Definition of a Numeric Item

   a. Its picture-string can contain only the symbols 9, P, S, and V. It must contain at least one 9.

      The picture-string must have from 1 to 18 digit positions.

   b. It can contain only the digits 0 through 9 and an operational sign.

7. Definition of an Alphanumeric Item

   a. Its picture-string can consist of all Xs, or a combination of the symbols A, X, and 9 (except all 9s or all As). The item is treated as if the character-string contained all Xs.

   b. Its contents can be any combination of characters from the complete character set (see Section 1.2.2.2).

8. Definition of an Alphanumeric Edited Item

   a. Its picture-string can consist of any combination of As, Xs, or 9s (it must contain at least one A or one X), plus at least one of the symbols B or 0.

   b. Its contents can be any combination of characters from the complete character set.

9. Definition of a Numeric Edited Item

   a. Its picture-string must contain at least one of the following editing symbols:

      , . * + - 0 B CR DB $

      It can also contain the symbols 9, V, or P.

      The allowable sequences are determined by certain editing rules for each symbol and can be found in Note 10.

      The picture-string must have from 1 to 18 digit positions.

   b. The contents can be any combination of the digits 0 through 9 and the editing characters.

PICTURE (Cont.)

10. The symbols used to define the category of an elementary item and their functions are as follows.

A Each A in the picture-string represents a character position which can contain only a letter of the alphabet or a space.

B Each B in the picture-string represents a character position into which a space character is inserted during editing.

Examples: (A-FLD contains the value 092469)

	B-FLD picture-string	Result
MOVE A-FLD TO B-FLD	99B99B99	0 9 Δ 2 4 Δ 6 9
MOVE A-FLD TO B-FLD	9999BBBB	0 9 2 4 Δ Δ Δ Δ

Also see Note 14, "Simple Insertion Editing".

P Each P in the picture-string indicates an assumed decimal point scaling position and is used to specify the location of an assumed decimal point when the point is outside the positions defined for the item. Ps are not counted in the size of the data item. They are counted in determining the maximum number of digit positions (18) allowed in numeric edited items or numeric items. Ps can appear only to the left or right of the picture-string and must appear together. The assumed decimal point is assumed to be to the left of the string of Ps it the Ps are at the left end of the picture-string and to the right of the string of Ps if the Ps are at the right end of the picture-string. If the V symbol is used in this case, it must appear in either of those positions; it is redundant.

Examples:

PPP9999 (or VPPP9999) defines a data item of four character positions whose contents is treated as .000nnnn during any decimal point alignment operation (such as in a MOVE or ADD). 9PPP (or 9PPPV) defines a data item of one character position whose contents is treated as n000 during any decimal point alignment operation.

S An S in a picture-string indicates that the item has an operational sign and retains the sign of any data stored in it. The S must be written as the leftmost character in the picture-string. If S is not included, all data is stored in the item as an absolute value and is treated as positive in all operations. The S symbol is not counted in the size of the data item.

PICTURE (Cont.)

V A V in a picture-string indicates the location of the assumed decimal point and can appear only once in a picture-string. The V does not represent a physical character position and is not counted in the size of the data item. If the assumed decimal point position is at the right of the rightmost character position of the item, the V is redundant (that is, 9999 is functionally equivalent to 9999V).

X Each X in a picture-string represents a character position which can contain any allowable character from the complete character set.

Z Each Z in a picture-string represents the leftmost leading numeric character positions in which leading zeros are to be replaced by spaces. Each Z is counted in the size of the item.

* Each * in a picture-string represents the leftmost leading numeric character positions in which leading zeros are to be replaced by *. Each * is counted in the size of the item.

Examples: (A-FLD contains the value 00305)

	B-FLD picture-string	Result
MOVE A-FLD TO B-FLD	999999	0 0 0 3 0 5
MOVE A-FLD TO B-FLD	ZZ9999	△ △ 0 3 0 5
MOVE A-FLD TO B-FLD	ZZZZZZ	△ △ △ 3 0 5
MOVE A-FLD TO B-FLD	ZZZZ.ZZ	△ 3 0 5 . 0 0

Also see Note 18, "Zero Suppression Editing".

9 Each 9 in a picture-string represents a character position which can contain a digit. Each 9 is counted in the size of the item.

0 Each 0 in a picture-string represents a character position into which a zero is inserted. It is counted in the size of the item. The 0 symbol works in the same manner as the B symbol.

, Each , in a picture-string represents a character position into which a comma is inserted.

Examples: (A-FLD contains 362577)

	B-FLD picture-string	Result
MOVE A-FLD TO B-FLD	9,999,999	0 , 3 6 2 , 5 7 7
MOVE A-FLD TO B-FLD	Z,ZZZ,ZZZ	3 6 2 , 5 7 7

PICTURE (Cont.)

Also see Note 14, "Simple Insertion Editing".

- . A . (dot or period) in a picture-string is an editing symbol that represents an actual decimal point. It is used for decimal point alignment and also indicates where a point (.) is to be inserted. This symbol is counted in the size of the item. Only one . can appear in a picture-string.

Examples: (A-FLD contains $3526_{\wedge}99$)[1]

	B-FLD picture-string	Result
MOVE A-FLD TO B-FLD	99,999.99	0 3 , 5 2 6 . 9 9
MOVE A-FLD TO B-BLD	ZZ,ZZZ.ZZ	Δ 3 , 5 2 6 . 9 9
MOVE A-FLD TO B-FLD	99999.9999	0 3 5 2 6 . 9 9 0 0

See Note 4 under MOVE in Chapter 5 for a clarification of the rule governing the third example.

Also see Note 15, "Special Insertion Editing".

+, -, CR, DB: Each of these symbols is used as an editing sign-control symbol. When used, they represent the character position(s) into which the editing sign-control symbol is placed. Only one of these symbols can appear in a character-string.

The + and - symbols can appear either at the beginning or at the end of a picture-string. The CR and DB symbols can appear only at the end of a picture-string.

- + The character position containing this symbol contains a + if the sending field either was unsigned (absolute) or had a positive operational sign; it contains a - if the sending field had a negative operational sign.

- - The character position containing this symbol contains a space if the sending field either was unsigned (absolute) or had a positive operational sign; it contains a - if the sending field had a negative operational sign.

- CR, DB: Each of these symbols requires two character positions. The character positions containing either of these symbols contains spaces if the sending field either was unsigned (absolute) or had a positive operational sign; they contain the symbol specified if the sending field had a negative operational sign.

[1] The caret (^) symbol is used to indicate the location of the assumed decimal point.

PICTURE (Cont.)

Examples: (A-FLD contains 3456‸25, B-FLD contains -3456‸25)[1]

	C-FLD picture-string	Result
MOVE A-FLD TO C-FLD	9999.99BCR	3456.25△△△
MOVE B-FLD TO C-FLD	9999.99BCR	3456.25△CR
MOVE A-FLD TO C-FLD	+9999.99	+3456.25
MOVE B-FLD TO C-FLD	+9999.99	-3456.25
MOVE A-FLD TO C-FLD	-9999.99	△3456.25
MOVE B-FLD TO C-FLD	-9999.99	-3456.25
MOVE A-FLD TO C-FLD	9999.99DB	3456.25△△
MOVE B-FLD TO C-FLD	9999.99DB	3456.25DB
MOVE B-FLD TO C-FLD	$9999.99+	$3456.25-

Also see Note 16, "Fixed Inserting Editing".

The + and - can also be used to perform floating insertion editing, a combination of zero suppression and symbol insertion. Floating insertion editing is indicated by the occurrence of two or more consecutive + or - symbols at the beginning of the picture-string. The total number of significant positions in the editing field must be at least one greater than the number of significant digits in the data to be edited. The floating + or - moves from left to right through any high-order zeros until a decimal point or the picture character 9 is encountered.

Examples: (A-FLD contains 0056‸25; B-FLD contains -0056‸25)

	C-FLD picture-string	Result
MOVE A-FLD TO C-FLD	++999.99	△+056.25
MOVE B-FLD TO C-FLD	++++9.99	△△-56.25
MOVE ZERO TO C-FLD	++999.99	△+000.00
MOVE ZERO TO C-FLD	+++++.++	△△△△△△△△

[1] The caret (ˆ) symbol is used to indicate the location of the assumed decimal point.

PICTURE (Cont.)

(In order for floating to go past decimal point, all numeric positions of item must be represented by the floating insertion symbol)

MOVE A-FLD TO C-FLD	--999.99	△△056.25
MOVE B-FLD TO C-FLD	--999.99	△-056.25
MOVE ZERO TO C-FLD	---99.99	△△△00.00
MOVE ZERO TO C-FLD	-------	△△△△△△△△

Also see Note 15, "Floating Insertion Editing".

Note that the + and - symbols are distinct from the S (operational sign) symbol. Normally, the + and - symbols are used to describe display items that are to appear on some printed report; they provide visual sign indication and cannot be used with items appearing as operands in arithmetic statements.

$ A $ (or the symbol specified by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph) represents the character position into which a $ (or the currency symbol) is to be placed. This symbol is counted in the size of the item.

Example: (A-FLD contains 3456‸75)

	B-FLD character-string	Result
MOVE A-FLD TO B-FLD	$9,999.99	$3,456.75
MOVE A-FLD TO B-FLD	$999,999.99	$003,456.75

Also see Note 16, "Fixed Insertion Editing".

The $ symbol can also be used to perform floating insertion editing. Floating insertion editing is indicated by the occurrence of two or more consecutive $ symbols at the beginning of the character string. The total number of significant positions in the editing field must be at least one greater than the number of significant digits in the data to be edited. The floating $ symbol floats from left to right through any high-order zeros until a decimal point or the picture character 9 is encountered.

Examples: (A-FLD contains 0056‸25)

	B-FLD picture-string	Result
MOVE A-FLD TO B-FLD	$$9,999.99	△$0,056.25
MOVE A-FLD TO B-FLD	$$$,$$$.99	△△△△$56.25
MOVE ZERO TO B-FLD	$$$,999.99	△△△$000.00
MOVE ZERO TO B-FLD	$$$,$$$.$$	△△△△△△△△△△

PICTURE (Cont.)

Also see Note 17, "Floating Insertion Editing".

11. There are two general methods of performing editing in the PICTURE clause:

    a. insertion, or

    b. suppression and replacement.

    There are four types of insertion editing available:

    a. Simple insertion

    b. Special insertion

    c. Fixed insertion

    d. Floating insertion

    There are two types of suppression and replacement editing:

    a. Zero suppression and replacement with spaces

    b. Zero suppression and replacement with asterisks

12. The type of editing that can be performed upon an item depends on the category to which the item belongs.

Category	Type of Editing Allowed
Alphabetic	None
Numeric	None
Alphanumeric	None
Alphanumeric edited	Simple insertion: 0 and B
Numeric Edited	All (except for the restriction given in Note 13)

13. Floating insertion editing and zero suppression/replacement editing are mutually exclusive in a PICTURE clause. Only one type of replacement can be used with zero suppression in a PICTURE clause.

14. Simple Insertion Editing (, B 0)

    The , (comma), B (space), and 0 (zero) constitute those editing symbols used in simple insertion editing. These insertion characters represent the character position in the item into which the character is inserted. These symbols are counted in the size of the item..

PICTURE (Cont.)

15. Special Insertion Editing (.)

    The . (decimal point) symbol is used in special insertion editing. In addition to its use as an insertion character, it also represents the position of the decimal point for decimal point alignment. This symbol is counted in the size of the item. The symbols . and V (assumed decimal point) are mutually exclusive in a PICTURE clause. If the . is the last symbol in the character-string, it must be immediately followed by one of the punctuation characters (semicolon or period).

16. Fixed Insertion Editing ($ + - CR DB)

    The currency symbol ($) and the editing sign control characters (+ - CR DB) constitute the characters used in fixed insertion editing. Only one $ and one of the editing sign control characters can be used in a PICTURE character-string. When the symbols CR or DB are used, they represent two character positions in determining the size of the item. The symbols + or - when used must be the leftmost or rightmost character positions to be counted in the size of the item. The $ when used must be the leftmost character position to be counted in the size of the item, except that it can be preceded by a + or - symbol. A fixed insertion editing character appears in the same character position in the edited item as it occupied in the PICTURE character-string.

    When the $ is used as a floating insertion editing character, the picture string must contain at least one $ more than the maximum number of significant digits in the item to be edited. If you use a comma and the $ simultaneously for editing, there must always be at least two $ to the left of the comma because one $ is always printed; there is no place for a significant digit to the left of the comma if you have used only one $. (If the item has a picture of $,$$$ then no digit ever appears to the left of the comma; a $ is always there.) A comma is omitted only when what appears to its left consists only of zeroes. (With the picture string $,$$$ the comma is never omitted.)

    Editing sign control symbols produce the following results depending on the value of the data being edited.

Editing Symbol in PICTURE character-string	Result	
	Data Positive	Data Negative
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

PICTURE (Cont.)

17. Floating Insertion Editing ($$ ++ --)

The $ and the editing sign control symbols + and - are the floating insertion editing characters and are mutually exclusive in a given PICTURE string.

Floating insertion editing is indicated in a PICTURE character-string by using a string of at least two of the allowable insertion characters to represent the leftmost numeric character positions into which the insertion characters can be floated. Any of the simple insertion characters embedded in the string of floating insertion characters or to the immediate right of this string are part of the floating string.

In a PICTURE character-string, there are only two ways of representing floating insertion editing.

a. Represent any two or more of the leading numeric character positions on the left of the decimal point by the insertion character. The result is that a single insertion character is placed in the character position immediately preceding the leftmost nonzero digit of the data being edited or in the character position immediately preceding the decimal point, or in the character position represented by the rightmost insertion character, whichever is encountered first.

b. Represent all numeric character positions in the character-string by the insertion character. If the value is not zero, the result is the same as when the insertion character appears only to the left of the decimal point. If the value is zero, the entire item is set to spaces.

A picture-string containing floating insertion characters must contain at least one more floating insertion character than the maximum number of significant digits in the item to be edited. For example, a data field containing five significant digit positions requires an editing field of at least six significant positions.

All floating insertion characters are counted in the size of the item.

18. Zero suppression Editing (Z *)

The suppression of leading zeros and commas in a data field is indicated by the use of the Z or the * symbol in a picture-string. These symbols are mutually exclusive in a given picture-string. Each suppression symbol is counted in the size of the item. If a Z is used, the replacement character is a space. If an * is used, the replacement character is an *. Zero suppression and replacement is indicated by a string of one or more Zs or *s to represent the leading numeric-character positions which are to be replaced when the associated character position in the data contains a leading zero. Any of the simple insertion characters embedded in this string of zero suppression symbols or to the immediate right of this string are part of the string.

PICTURE (Cont.)

If the zero suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to a zero suppression symbol in the string is replaced by the replacement character.

Suppression terminates at the first nonzero digit in the data represented by the suppression symbol in the string or at the decimal point, whichever is encountered first.

If all numeric character positions in the picture-string are represented by the suppression symbol 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, the entire item (including any sign) is set to the replacement character (with the exception of the decimal point if the suppresson symbol is an *).

When the * is used and the clause BLANK WHEN ZERO appears in the same entry and zeros are moved to the field, all character positions with the exception of the decimal point are replaced by *.

19. The symbols + - * Z and $ when used as floating replacement characters are mutually exclusive within a given picture-string.

20. The following chart shows the order of precedence of the various picture-string symbols. Each "Y" on the chart indicates that the symbol in the top row directly above can precede the symbol at the left of the row in which the "Y" appears.

    { } indicate that the symbols are mutually exclusive.

    The P and the fixed insertion + and - appear twice.

    P9, +9, and -9 represent the case where these symbols appear to the left of any numeric positions in the string.

    9P, 9+, and 9- represent the case where these symbols appear to the right of any numeric positions in the string.

    The Z, *, and the floating ++, --, and $$ also appear twice.

    Z., *., $$., and --. represent the case where these symbols appear before the decimal point position.

    .Z, .*, .$$, .++, and .-- represent the case where these symbols appear following the decimal point position.

PICTURE (Cont.)

		FIXED INSERTION								OTHER											
		B	0	,	.	{+9 -9}	{9+ 9-}	{CR DB}	$	A X	P9	9P	S	V	{Z. *.}	{.Z .*}	9	{++. --.}	{.++ .--}	$$.	.$$
FIXED INSERTION	B	Y	Y	Y	Y	Y			Y	Y	Y			Y	Y	Y	Y	Y	Y	Y	Y
	0	Y	Y	Y	Y	Y			Y	Y	Y			Y	Y	Y	Y	Y	Y	Y	Y
	,	Y	Y	Y	Y	Y			Y		Y			Y	Y	Y	Y	Y	Y	Y	Y
	.	Y	Y	Y		Y			Y		Y				Y		Y	Y		Y	
	{+9 -9}										Y			Y							
	{9+ 9-}	Y	Y	Y	Y				Y		Y	Y		Y	Y	Y	Y			Y	Y
	{CR DB}	Y	Y	Y	Y				Y		Y	Y		Y	Y	Y	Y			Y	Y
	$					Y					Y			Y							
OTHER	A X	Y	Y							Y							Y				
	P9					Y			Y		Y		Y	Y							
	9P	Y	Y	Y		Y	Y	Y	Y			Y	Y		Y		Y	Y		Y	
	S																				
	V	Y	Y	Y		Y	Y	Y	Y			Y	Y		Y		Y	Y		Y	
	{Z. *.}	Y	Y	Y		Y			Y						Y						
	{.Z .*}	Y	Y	Y	Y	Y			Y		Y			Y	Y	Y					
	9	Y	Y	Y	Y	Y			Y	Y	Y		Y	Y	Y		Y	Y		Y	
	{++. --.}	Y	Y	Y					Y									Y			
	{.++ .--}	Y	Y	Y	Y				Y		Y			Y				Y	Y		
	$$.	Y	Y	Y		Y														Y	
	.$$	Y	Y	Y	Y	Y					Y			Y						Y	Y

MR-S-1016-81

4.11.2.8 REDEFINES

Function

The REDEFINES clause allows the same memory area to be allocated to two or more data items.

General Format

level-number data-name-1 <u>REDEFINES</u> data-name-2

Technical Notes

1. The REDEFINES clause, when used, must immediately follow data-name-1.

2. The level-numbers of the data-name-1 and data-name-2 entries must be identical.

3. This clause must not be used for level-number 66 or 88 items. Also, it must not be used for level-01 entries in the FILE SECTION; implicit redefinition is provided by specifying more than one data-name in the DATA RECORDS ARE clause in the FD.

4. When the level-number of the data-names is other than level-01, the storage area for data-name-2 should be of the same size or shorter than data-name-1. FILLER items can be used to comply with this rule.

5. The REDEFINES entry must immediately follow the entries describing data-name-2.

6. The REDEFINES entry cannot be a subordinate to the OCCURS clause.

7. The redefinition entries cannot contain VALUE clauses.

8. Data-name-2 must not be qualified.

REDEFINES (Cont.)

9. The following example illustrates the use of the REDEFINES entry. The entries shown cause AREA-A and AREA-B to occupy the same area in memory.

```
03  AREA-A USAGE DISPLAY-6.
    04  FIELD-1 PICTURE IS X(7).
    04  FIELD-2 PICTURE IS A(13).
    04  FIELD-3.
        05  SUBFIELD-1 PICTURE IS
            S999V99 USAGE IS COMP.
        05  SUBFIELD-2 PICTURE IS
            S999V99 USAGE IS COMP.
03  AREA-B REDEFINES AREA-A USAGE DISPLAY-6.
    04  FIELD-A PICTURE IS X(22).
    04  FIELD-B PICTURE IS X(5).
    04  FILLER PICTURE IS X(9).
```

Note how the length of each area is calculated so that AREA-B can be defined so that its size is equal to that of AREA-A.

AREA-A:	FIELD-1	7	6-bit characters (DISPLAY-6 assumed)
	FIELD-2	13	6-bit characters (DISPLAY-6 assumed)
		4	6-bit characters (not used because COMP items must start at a new word boundary)
	SUBFIELD-1	6	6-bit characters (COMP items occupy one word, or six 6-bit character positions)
	SUBFIELD-1	6	6-bit characters (COMP items occupy one word, or six 6-bit character positions)
Total 6-bit characters		36	
AREA-B:	FIELD-A	22	6-bit characters (DISPLAY-6 assumed)
	FIELD-B	5	6-bit characters (DISPLAY-6 assumed)
	FILLER	9	6-bit characters (needed to make AREA-B size equal to AREA-A)
Total 6-bit characters		36	

4.11.2.9 RENAMES (level-66)

Function

The RENAMES clause permits alternate, possible overlapping, groupings of elementary items.

General Format

```
66 data-name-1 RENAMES data-name-2   [THRU data-name-3] .
```

Technical Notes

1. All RENAMES entries associated with items in a given record must immediately follow the last data description entry for that record.

```
01  data-name-a
     .
    (data description entries)
     .
     .
    (level-66 entries associated with this logical record)
01  data-name-b.
     .
     .
```

2. Data-name-1 cannot be used as a qualifier, and can be qualified only by the names of the level-01 or FD entries associated with it.

3. Data-name-2 and data-name-3 must be the names of items in the assoicated logical record and cannot be the same data-name.

   Neither data-name-2 nor data-name-3 can have a level-number of 01, 66, 77, or 88. Neither of these data-names can have an OCCURS clause in its data description entry, nor be subordinate to an item that has an OCCURS clause in its data description entry.

   Data-name-2 must precede data-name-3 in the record description, and data-name-3 cannot be subordinate to data-name-2. If there is any associated redefinition (REDEFINES), the ending point of data-name-3 must logically follow the beginning point of data-name-2. When data-name-3 is specified, data-name-1 is a group item that includes all elementary items starting with data-name-2 (if data-name-2 is an elementary item) or the first elementary item in data-name-2 (if data-name-2 is a group item) and concluding with data-name-3 (or the last elementary item in data-name-3).

RENAMES (level-66) (Cont.)

If data-name-3 is not specified, data-name-2 can be either a group or elementary item. If it is a group item, data-name-1 is treated as a group item and includes all elementary items in data-name-2; if data-name-2 is an elementary item, data-name-1 is treated as an elementary item with the same descriptive clauses.

4. The following examples illustrate the use of the RENAMES entry.

```
01  RECORD-NAME.
    02  FIRST-PART.
        03  PART-A.
            04  FIELD-1 PICTURE IS ...
            04  FIELD-2 PICTURE IS ...
            04  FIELD-3 PICTURE IS ...
        03  PART-B.
            04  FIELD-4 PICTURE IS ...
            04  FIELD-5.
                05  FIELD-5A PICTURE IS ...
                05  FIELD-5B PICTURE IS ...
    03  SECOND-PART.
        03  PART-C.
            04  FIELD-6 PICTURE IS ...
            04  FIELD-7 PICTURE IS ...
    66  SUBPART RENAMES PART-B THRU PART-C.
    66  SUBPART1 RENAMES FIELD-3 THRU SECOND-PART.
    66  SUBPART2 RENAMES FIELD-5B THRU FIELD-7.
    66  AMOUNT RENAMES FIELD-7.
```

4.11.2.10 SYNCHRONIZED

Function

The SYNCHRONIZED clause specifies the positioning of an elementary item within a computer word (or words).

General Format

$$\left[\left\{ \begin{array}{l} \underline{\text{SYNCHRONIZED}} \\ \underline{\text{SYNC}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{LEFT}} \\ \underline{\text{RIGHT}} \end{array} \right\} \right]$$

MR-S-1017-81

Technical Notes

1. This clause can appear only in the data description of an elementary item.

2. This clause specifies that the item being defined is to be placed in an integral number of computer words and that it is to begin or end at a computer word boundary. No other adjacent fields are to occupy these words. The unused positions, however, must be counted when calculating:

   a. The size of any group to which this elementary item belongs, and

   b. The computer memory allocation when the item appears as the object of a REDEFINES clause. However, when a SYNCHRONIZED item is referenced, the original size of the item (as indicated by the PICTURE clause) is used in determining such things as truncation, justification, and overflow.

3. SYNCHRONIZED LEFT or SYNC LEFT specifies that the item is to be positioned in such a way that it begins at the left boundary of a computer word.

   SYNCHRONIZED RIGHT or SYNC RIGHT specifies that the item is to be positioned in such a way that it terminates at the right boundary of a computer word.

4. When the SYNCHRONIZED clause is specified for an item within the scope of an OCCURS clause, each occurrence of the item is SYNCHRONIZED.

5. Any FILLER required to position the item as specified is automatically generated by the compiler. The content of this FILLER is indeterminate.

SYNCHRONIZED (Cont.)

6. COMP(UTATIONAL), COMP(UTATIONAL)-1, and INDEX items are always implicitly SYNCHRONIZED RIGHT, and therefore cannot be SYNCHRONIZED LEFT.

7. An item subordinate to one containing a VALUE clause cannot be SYNCHRONIZED.

8. Only DISPLAY-6, DISPLAY-7, DISPLAY-9, or COMP-3 items can be SYNCHRONIZED.

4.11.2.11 USAGE

Function

The USAGE clause specifies the format of a data item in computer storage.

General Format

$$\left[\ \left[\underline{\text{USAGE}}\ \text{IS}\right]\left\{\begin{array}{l}\underline{\text{COMPUTATIONAL}}\\ \underline{\text{COMP}}\\ \underline{\text{COMPUTATIONAL-1}}\\ \underline{\text{COMP-1}}\\ \underline{\text{COMPUTATIONAL-3}}\\ \underline{\text{COMP-3}}\\ \underline{\text{DISPLAY}}\\ \underline{\text{DISPLAY-6}}\\ \underline{\text{DISPLAY-7}}\\ \underline{\text{DISPLAY-9}}\\ \underline{\text{INDEX}}\\ \underline{\text{DATABASE-KEY}}\\ \underline{\text{DBKEY}}\end{array}\right\}\ \right]$$

MR-S-1018-81

Technical Notes

1. The USAGE clause can be written at any level. If it is written at a group level, it applies to each elementary item in the group. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

   Note that the recording mode of a file determines how the data is recorded on the external medium. The recording mode can be inferred from the usage mode of the data records, but the reverse is never true. The usage of a data record is never inferred from the declared recording mode of the file.

   The implied USAGE of a group item is DISPLAY-7 if the first elementary item subordinate to it is declared as DISPLAY-7, or DISPLAY-9 if the first elementary item subordinate to it is declared as either DISPLAY-9 or COMP-3; otherwise, its USAGE is DISPLAY-6. However, if the /X switch is included in the compiler command string, the default USAGE is DISPLAY-9.

   USAGES of DISPLAY-6, DISPLAY-7, and DISPLAY-9/COMP-3 cannot be mixed. However, USAGES of COMP, COMP-1 and INDEX can be mixed with the aforementioned USAGES.

2. This clause specifies the manner in which a data item is represented within computer memory.

USAGE (Cont.)

3. COMPUTATIONAL (COMP)

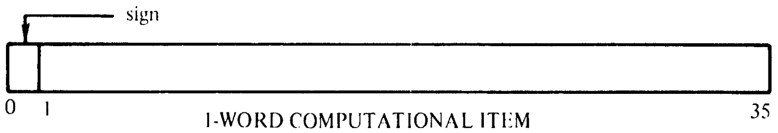
   a. COMP is equivalent to COMPUTATIONAL.

   b. A COMPUTATIONAL item represents a value to be used in computations and must be numeric. Its picture-string can contain only the symbols: 9 S V P. Its value is represented as a binary number with an assumed decimal point.

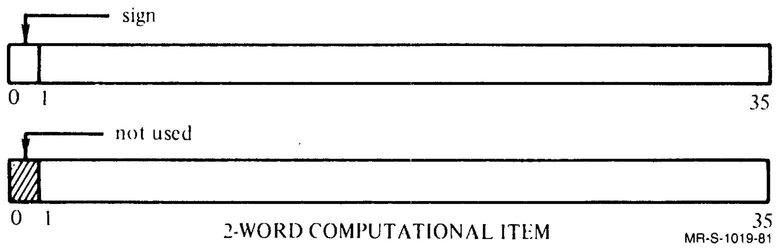
   c. If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. However, the group itself is not COMPUTATIONAL and cannot be used as an operand in arithmetic computations.

   d. COMPUTATIONAL items of 10 or fewer decimal positions are SYNCHRONIZED RIGHT in one computer word. Computational items of more than 10 decimal positions are SYNCHRONIZED RIGHT in two full computer words.

   e. The following illustrations give the format of a COMPUTATIONAL item.



1-WORD COMPUTATIONAL ITEM



2-WORD COMPUTATIONAL ITEM

MR-S-1019-81

USAGE (Cont.)

4. COMPUTATIONAL-1 (COMP-1)

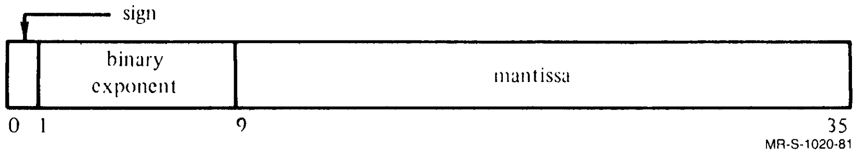
   a. COMP-1 is equivalent to COMPUTATIONAL-1.

   b. A COMPUTATIONAL-1 item can contain a value, in floating point format, to be used in computations. It must be numeric. A COMP-1 item must not have a PICTURE.

   c. If a group item is described as COMPUTATIONAL-1, the elementary items within the group are COMPUTATIONAL-1. However, the group item itself is not COMPUTATIONAL-1 and cannot be used as an operand in arithmetic computations.

   d. COMPUTATIONAL-1 items are SYNCHRONIZED in one full computer word.

   e. The following illustration gives the format of a COMPUTATIONAL-1 item.



USAGE (Cont.)

5. COMPUTATIONAL-3 (COMP-3)

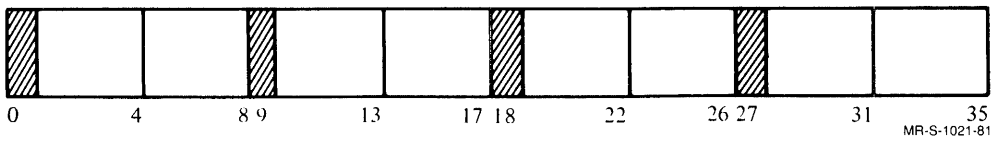
   a. COMP-3 is equivalent to COMPUTATIONAL-3.

   b. A COMP-3 item's picture string can contain only the symbols 9 S V P. Its value is represented as a packed decimal number with an assumed decimal point.

   c. If a group item is declared as COMP-3 the elementary items in the group are COMP-3. However, the group item itself is not COMP-3 and cannot be used as an operand in arithmetic computations.

   d. The maximum size of a COMP-3 item is 18 decimal digits.

   e. The following illustration gives the format of a COMP-3 item. Note that bits 0, 9, 18 and 27 of the word are not used.



   f. COMP-3 items can be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT.

   g. COMP-3 items can share a computer word with other COMP-3 items or with DISPLAY-9 items. However, COMP-3 items always begin at one of the following bit positions in a word: 1, 10, 19, 28.

   h. The actual size of a COMP-3 item in memory is at least four bits larger and can be nine bits larger than the number of character positions because the sign is stored in the last four bits of the item and the item is stored right justified on a nine-bit byte boundary.

   i. The octal values 12, 14, and 16 represent plus signs and the octal values 13 and 15 represent minus signs. The octal value 17 represents the non-printing plus sign. Although octal 12, 14 and 16 represent plus signs, the sign given to the positive result of any arithmetic operation is 14. Similarly, the minus sign given to the negative result of any arithmetic operation is 15.

      The non-printing plus sign is actually an absolute value indicator. Any positive or negative number which is moved into an item with this sign receives this sign. In arithmetic computations and numeric editing operations, items containing the nonprinting plus sign are treated as positive.

USAGE (Cont.)

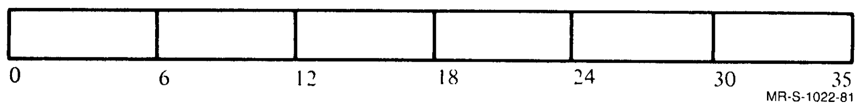
6. DISPLAY-6

   a. DISPLAY is equivalent to DISPLAY-6 when the /X switch is not given in the compiler command string.

   b. A DISPLAY-6 item represents a string of 6-bit characters. Its picture-string can contain any picture symbols. Refer to Appendix B for the SIXBIT collating sequence.

   c. DISPLAY-6 items can be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT, as desired. Otherwise, they can share a computer word with other DISPLAY-6 items.

   d. The illustration below given the format of a DISPLAY-6 word.



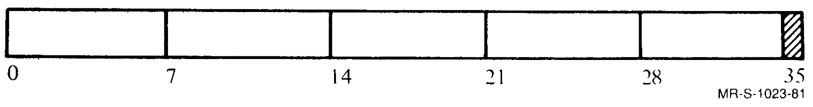
   e. If the USAGE clause is omitted for an elementary item, its USAGE is assumed to be DISPLAY-6 if the /X switch has not been included in the compiler command string.

7. DISPLAY-7

   a. A DISPLAY-7 item represents a string of 7-bit ASCII characters. Its picture-string can contain any picture symbols.

   b. DISPLAY-7 items can be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT, as desired; otherwise, they can share a computer word with other items. If the item is SYNCHRONIZED RIGHT, the last character of the item ends in bit 34 of a computer word.

   c. The illustration below gives the format of a DISPLAY-7 word.



8. DISPLAY-9

   a. DISPLAY is equivalent to DISPLAY-9 when the /X switch is included in the command string to the compiler.

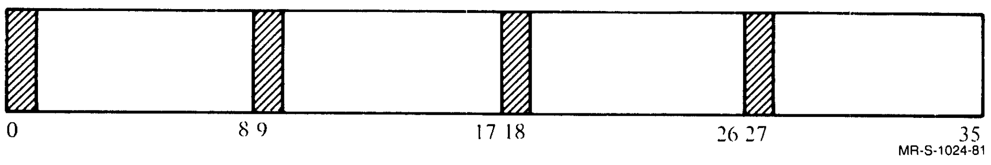
   b. A DISPLAY-9 item represents a string of EBCDIC characters. Its picture string can contain any picture symbol.

USAGE (Cont.)

c. DISPLAY-9 items can be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT as desired; otherwise, they can share a computer word with other DISPLAY-9 OR COMP-3 items. If the item is SYNCHRONIZED RIGHT, the last character of the item ends in bit 35 of a computer word.

d. The maximum length of a DISPLAY-9 item is 4,096 characters.

e. The illustration below gives the format of a DISPLAY-9 item. Note that bits 0, 9, 18, and 27 are not used.



f. If the USAGE clause is omitted for an elementary item, its USAGE is assumed to be DISPLAY-9 if the /X switch has been included in the computer command string.

9. INDEX

a. An elementary item described as USAGE INDEX is called an index data-item. It is treated as a COMP item with PICTURE S9(5) and can be used as a COMP item.

b. An index data-item must not have a PICTURE.

c. If a group item is described as INDEX, the elementary items within the group are treated as INDEX. However, the group item itself is not INDEX and cannot be used as an operand in arithmetic statements.

d. Index data items and index-names (defined in the OCCURS clause by the INDEXED BY option) are equivalent.

e. If an index-name is defined in an OCCURS clause, it cannot be defined elsewhere.

10. DATABASE-KEY

a. DATABASE-KEY and DBKEY are equivalent and interchangeable.

b. An item described as USAGE DATABASE-KEY is treated as a COMP item with PICTURE S9(10) and can be used as a COMP item.

USAGE (Cont.)

c. The item with USAGE DATABASE-KEY must not have a PICTURE.

d. An item with USAGE DATABASE-KEY is primarily used in programs accessing data bases through the Data Base Management System (DBMS). This item can be used to store the value of a data base key. All data base keys are assigned by DBMS and cannot be changed by you. Refer to the DBMS Programmer's Procedures Manual for more information about DBMS.

VALUE

4.11.2.12 VALUE

Function

The VALUE clause defines the initial value of WORKING-STORAGE items, and the values associated with condition-names (level-88).

General Format

Format 1:

```
[ VALUE IS literal ]
```

Format 2:

```
[ { VALUE  IS  } literal-1 [ THRU literal-2 ]
  { VALUES ARE }

     [ ,literal-3 [ THRU literal-4 ] ] ... ]
                                    MR-S-1025-81
```

Technical Notes

1. Format 2 can be specified only for level-88 items.

2. In the FILE SECTION, the VALUE clause can be used only with level-88 items. In the WORKING-STORAGE SECTION, it can be used at all levels, except level-66. It must not be stated in a data description entry that contains an OCCURS clause or that is subordinate to an entry containing an OCCURS clause. Also, it must not be stated in an entry that contains a REDEFINES clause or that is subordinate to an entry that contains a REDEFINES clause.

3. If the VALUE clause is used at a group level, the literal must be a figurative constant or a nonnumeric literal. The group item is initialized to this value without consideration for the individual elementary or group items contained within this group. No VALUE clauses can appear at subordinate levels within the group.

4. If no VALUE clause appears for a WORKING-STORAGE item, the initial value of the item is unpredictable.

VALUE (Cont.)

5. More information concerning Format 2 can be found under "condition-name (Level-88)" in this chapter.

6. The VALUE clause must not conflict with other clauses in the data description entry or in the data description entries within the hierarchy of the item. The following rules apply:

   a. If the category of an item is numeric, all literals in the VALUE clause must be numeric. All literals in a VALUE clause must have a value within the range of values indicated by the PICTURE clause; for example, an item with PICTURE PPP9 can have only the values in the range .0000 through .0009.

   b. If the category of the item is alphabetic or alphanumeric, all literals in the VALUE clause must be alphanumeric literals. The literal is aligned according to the normal alignment rules (see "JUSTIFIED") except that the number of characters in the literal must not exceed the size of the item.

   c. If the category of an item is numeric-edited or alphanumeric-edited, no editing of the value is performed in the VALUE clause.

   d. The USAGE of the literal agrees with the USAGE of the item. Thus, if the item has USAGE DISPLAY-6, the literal also has USAGE DISPLAY-6 and its value must contain legal SIXBIT characters.

7. The figurative constants SPACE(S), ZERO(E)(S), QUOTE(S), LOW-VALUE(S), and HIGH-VALUE(S) can be substituted for a literal. If the item is numeric, only ZERO(E)(S), LOW-VALUE(S), and HIGH-VALUE(S) are allowed.

REPORT SECTION

4.12 REPORT SECTION

The REPORT SECTION contains the descriptions of one or more reports and the report groups that make up each report.

Report groups are the basic elements of a report. Each report group is divided into report lines, which are in turn divided into fields. The report groups that can appear in a report are:

REPORT HEADING	Printed once at the beginning
REPORT FOOTING	Printed once at the end
PAGE HEADING	Printed at the beginning of each page
PAGE FOOTING	Printed at the end of each page
DETAIL	Printed for each set of report data
CONTROL HEADING	Printed at the beginning of each detail report group when a control break occurs
CONTROL FOOTING	Printed at the end of each detail report group when a control break occurs

The detail report groups contain the data items that constitute the report. Data items within a detail group can be designated by the programmer as controls. These control items are in descending order of rank from FINAL, through major, intermediate, to minor. Each time a control item changes, a control break is said to occur; the control footings for the detail group are printed, and control headings for the next detail group are printed before the next detail group is printed. A FINAL control break occurs twice during the generation of a report, before the first detail line is printed and after the last detail line is printed. The most major control breaks least often and the most minor control breaks most often. If the most minor control field breaks, the control footing for that control field is generated, and the control heading for the next detail group for that control is generated. If a more major control field breaks, the control footings for all fields more minor than that which broke are generated, starting with the most minor and continuing up to the control footing for the control that broke. The control headings are then printed starting with the control field that broke and continuing through the most minor control field. An example of a skeleton report follows.

```
REPORT HEADING
PAGE HEADING
CONTROL HEADING (FINAL)
CONTROL HEADING (MAJOR)
CONTROL HEADING (MINOR)
DETAIL GROUP
          .
          .
          .
CONTROL FOOTING (MINOR) (control break occurred)
CONTROL HEADING (MINOR)
DETAIL GROUP
          .
          .
          .
```

REPORT SECTION (Cont.)

```
CONTROL FOOTING (MINOR)
CONTROL FOOTING (MAJOR)  (control break occurred)
CONTROL HEADING (MAJOR)
CONTROL HEADING (MINOR)
DETAIL GROUP
          .
          .
          .
CONTROL FOOTING (MINOR)
CONTROL FOOTING (MAJOR)
CONTROL FOOTING (FINAL)  (control break occurred)
PAGE FOOTING
REPORT FOOTING
```

Within a report file, more than one report can be written. If more than one report is written in a file, the names of all the reports must be specified in the REPORTS clause of the file description entry, and a unique code must be specified for each report by means of the CODE clause in the Report Description fo each report. The code must also be identified in the SPECIAL-NAMES section of the ENVIRONMENT DIVISION.

To print one of the reports within a report file, you enter the filename and the code of the desired report into the queue for the line-printer spooler, LPTSPL. LPTSPL copies the report lines with the designated code to the line printer, but does not erase the lines from the file. The file is entered into the line-printer queue by means of the PRINT monitor command. The code is specified by the /REPORT switch in the Queue command string.

```
PRINT filespec/REPORT:code
```

Only the first 12 characters of the code are accepted in the PRINT command string.

Included in the description of a report are the number of lines on a report page, where headings should begin on the page, where footings should end, the column on the page where each item in a report group is to be placed, and the number of lines that are left between report groups.

To cause a report to be printed, in addition to specifying its format and data in the DATA DIVISION, you must include certain verbs in the PROCEDURE DIVISION. These verbs are: INITIATE, which initializes the report and sets sum counters to zero; GENERATE, which causes report groups to be generated on specified control breaks; and TERMINATE, which ends the report. An additional statement, USE BEFORE REPORTING, causes a programmer-specified procedure to be performed before a report group is produced.

REPORT DESCRIPTION (RD)

4.12.1 Report Description (RD)

Function

The Report Description furnishes information concerning the physical structure for a report.

General Format

```
RD report-name

    [CODE mnemonic-name]

   [{CONTROL IS  } {identifier-1 ,identifier-2 ...                     }]
   [{CONTROLS ARE} {FINAL [,identifier-1 [,identifier-2] ...]          }]

    [PAGE {LIMIT IS  } integer-1 {LINES}
    [     {LIMITS ARE}           {LINES}

          [HEADING integer-2] [FIRST DETAIL integer-3]

          [LAST DETAIL integer-4] [FOOTING integer-5] ] .
```

MR-S-1026-81

Technical Notes

1. The order of appearance of the optional clauses is immaterial.

2. A fixed data-name PAGE-COUNTER is automatically generated for each RD entry.

   Its function is to contain the current page number of a report. It is a COMPUTATIONAL item; its size is equal to the size of the largest field that refers to it in a SOURCE clause. The contents of the PAGE-COUNTER are set to 1 by the INITIATE statement.

3. The fixed data-name LINE-COUNTER is automatically generated for each RD entry. Its function is to contain the current line number within a report page. It is a COMPUTATIONAL item; its size is based on the number of lines specified in the PAGE-LIMIT clause.

REPORT DESCRIPTION (RD) (Cont.)

4. PAGE-COUNTER or LINE-COUNTER can be referenced as if it were any data-name. It must be qualified by the report-name if more than one RD entry is present in the program.

5. Each of the above clauses appears on the following pages.

CODE

4.12.1.1 CODE

Function

The CODE clause defines a unique string of one or more characters that is affixed to each line of the report.

General Format

<u>CODE</u> mnemonic-name

Technical Notes

1. This clause is necessary only if more than one report is to be written in a single file.
2. Mnemonic-name is defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.
3. The character string represented by mnemonic-name is affixed to the beginning of each report line, and is used to uniquely define the lines of separate reports written in one file.
4. The number of characters represented by mnemonic-name must be the same for the codes of all reports in the same file.

4.12.1.2 CONTROL(S)

Function

The CONTROL clause indicates the identifiers that control the printing of totals in the report.

General Format

$$\left\{\begin{array}{l}\underline{\text{CONTROL}}\ \text{IS}\\ \underline{\text{CONTROLS}}\ \text{ARE}\end{array}\right\}\left\{\begin{array}{l}\underline{\text{FINAL}}\\ \text{identifier-1}\ \ [\text{,identifier-2}]\ \ldots\\ \underline{\text{FINAL}}\text{,identifier-1}\ \ [\text{,identifier-2}]\ \ldots\end{array}\right\}$$

MR-S-1027-81

Technical Notes

1. The CONTROL clause is required when CONTROL HEADING or CONTROL FOOTING report groups are specified.

2. The identifiers specify the control hierarchy for this report. They are listed in order from major to minor; FINAL is the highest level of control, identifier-1 is the major control, identifier-2 is the intermediate control, etc. The last identifier specified is the minor control.

3. Identifiers must not be defined in the Report Section. Identifiers can be qualified, but they cannot be subscripted or indexed.

PAGE LIMIT

4.12.1.3 PAGE LIMIT

Function

The PAGE LIMIT clause indicates the specific line control to be maintained within the presentation of a report page.

General Format

```
PAGE {LIMIT IS  } integer-1 {LINE }
     {LIMITS ARE}           {LINES}

     [HEADING integer-2] [FIRST DETAIL integer-3]
     [LAST DETAIL integer-4] [FOOTING integer-5]
```

MR-S-1028-81

Technical Notes

1. The PAGE LIMIT clause is required when page format must be controlled by the Report Writer.

2. All integers must have a positive value less than 512. Integer-2 through integer-5 must not be greater than integer-1.

3. If absolute line spacing is indicated for all report groups (see the LINE NUMBER and NEXT GROUP clauses Sections 4.12.2.3 and 4.12.2.4 respectively), integer-2 through integer-5 need not be specified.

4. The integers specify line numbers relative to the beginning of a page.

5. The HEADING clause specifies the first line of a page to be used; no line precedes integer-2.

6. The FIRST DETAIL clause specifies the first line of the first DETAIL or CONTROL print group; no DETAIL or CONTROL group precedes integer-3.

7. The LAST DETAIL clause specifies the last line of a DETAIL or CONTROL HEADING report group; no such group extends beyond integer-4.

8. The FOOTING clause specifies the last line number of the last CONTROL FOOTING report group; no CONTROL FOOTING group extends beyond integer-5.

PAGE LIMIT (Cont.)

9. If any optional clause is omitted, a value is assumed for its integer. The default values are:

integer-2:	Default is 1
integer-3:	Default is the value of integer-2
integer-4:	Default is the value of integer-5 if specified; if integer-5 is also omitted, the default is the value of integer-1
integer-5:	Default is the value of integer-4 if specified; if integer-4 is omitted, the default is the value of integer-1.

Report Group Description (RD)

4.12.2 Report Group Description

Function

The Report Group Description entry specifies the characteristics and format of a particular report group.

General Format

Option 1:

```
01 [data-name-1]

   [                   { integer-1       } ]
   [ LINE NUMBER IS    { PLUS integer-2  } ]
   [                   { NEXT PAGE       } ]

   [                   { integer-3       } ]
   [ NEXT GROUP IS     { PLUS integer-4  } ]
   [                   { NEXT PAGE       } ]

           { REPORT HEADING                                        }
           { RH                                                    }
           { PAGE HEADING                                          }
           {                                                       }
           { PH  { CONTROL HEADING }  { identifier-1 }             }
           {     { CH              }  { FINAL        }             }
           {                                                       }
  TYPE IS  { DETAIL                                                }
           {                                                       }
           { DE  { CONTROL FOOTING }  { identifier-2 }             }
           {     { CF              }  { FINAL        }             }
           {                                                       }
           { PAGE FOOTING                                          }
           { PF                                                    }
           { REPORT FOOTING                                        }
           { RF                                                    }
                                                     MR-S-1029-81
    [               { DISPLAY   } ]
    [ [USAGE IS]    { DISPLAY-6 } ]  .
    [               { DISPLAY-7 } ]
    [               { DISPLAY-9 } ]
```

Report Group Description (RD) (Cont.)

Option 2:

```
level-number [data-name-1]

    [BLANK WHEN ZERO]

    [COLUMN NUMBER IS integer-1]

    [GROUP INDICATE]

    [{JUSTIFIED} RIGHT]
     {JUST     }

    [                {integer-2        }]
    [LINE NUMBER IS  {PLUS integer-3   }]
    [                {NEXT PAGE        }]

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

    [RESET ON {identifier-1}]
              {FINAL       }

    {SOURCE IS identifier-2                                       }
    {SUM identifier-3 [,identifier-4] ... [UPON data-name-2]      }
    {VALUE IS literal-1                                           }

    [              {DISPLAY  }]
    [[USAGE IS]    {DISPLAY-6}] .
    [              {DISPLAY-7}]
    [              {DISPLAY-9}]
```

MR-S-1030-81

Report Group Description (RD) (Cont.)

Technical Notes

1. Except for the data-name, which when present must immediately follow the level-number, the clauses can be written in any order.

2. In order for a report group to be referred to by a PROCEDURE DIVISION statement, it must have a data-name.

3. All elementary items must have a PICTURE clause and one of the clauses SOURCE, SUM, or VALUE.

4. For a detailed description of the BLANK WHEN ZERO, JUSTIFIED, PICTURE, VALUE, and USAGE clauses, see the pages following the Data Description Entry.

5. The data-name need not appear in an entry unless it is referred to by a GENERATE or USE statement, or reference is made to the SUM counter.

6. If the 01-level item is elementary, the clauses in Format 2 can be used in addition to the clauses in Format 1.

7. The remaining clauses are described in detail on the following pages.

4.12.2.1 COLUMN

Function

The COLUMN NUMBER clause indicates the column on the printed page in which the high-order (leftmost) character of an item is printed.

General Format

COLUMN NUMBER IS integer

Technical Notes

1. Integer must have a positive value less than 512.
2. This clause is valid only for an elementary item.
3. Within a report group and a particular LINE NUMBER specification, COLUMN NUMBER entries must be indicated from left to right.
4. If the COLUMN NUMBER clause is omitted, the elementary item, though included in the description, is suppressed when the report group is produced at object time.

GROUP INDICATE

4.12.2.2 GROUP INDICATE

Function

The GROUP INDICATE clause indicates that this elementary item is to be produced only on the first occurrence of the item after any CONTROL or PAGE breaks.

General Format

GROUP INDICATE

Technical Notes

1. This clause can only be used at the elementary level within a TYPE DETAIL report group.
2. A GROUP INDICATEd item is presented in the first detail line of a report after any control breaks and after any page breaks; it is suppressed at all other times.

4.12.2.3 LINE NUMBER

Function

The LINE NUMBER clause indicates the absolute or relative line number entry in reference to the page or the previous entry.

General Format

$$\underline{\text{LINE}}\ \text{NUMBER IS} \left\{ \begin{array}{l} \text{integer-1} \\ \underline{\text{PLUS}}\ \text{integer-2} \\ \underline{\text{NEXT}}\ \underline{\text{PAGE}} \end{array} \right\}$$

MR-S-1031-81

Technical Notes

1. Integer-1 and integer-2 must be positive integers with values less than 512. Integer-1 must be within the range specified by the PAGE LIMITS clause in the RD entry.

2. The LINE NUMBER clause must be given for each report line of a report group, and must be specified at or before the first elementary item that contains a COLUMN clause of each report line. If an item does not contain a COLUMN clause and the LINE NUMBER clause is specified for it, no printing is done, but the LINE NUMBER clause causes vertical spacing to be done.

3. If a LINE NUMBER clause is specified for an item, all entries following that item, up to but not including the next item with a LINE NUMBER clause, are presented on the same line.

4. A LINE NUMBER at a subordinate level can not contradict a LINE NUMBER at a group level.

5. Integer-1 indicates that the current line is to be presented at that line number.

6. PLUS integer-2 indicates that the LINE-COUNTER is to be incremented by the value of integer-2, and that the current line is to be presented on the line specified by the new value of the LINE-COUNTER.

7. NEXT PAGE is used to indicate an automatic skip to the next page before the current line is presented. If there is no PAGE-LIMIT clause, there is only a skip to the top of the next page. However, if there is a PAGE-LIMIT clause, after skipping to the next page, the Report Writer spaces as follows:

Type of Line	Space To
Detail, control heading, control footing	First detail line
Report heading, report footing, page heading	Heading line
Page footing	Footing line

4.12.2.4 NEXT GROUP

Function

The NEXT GROUP clause specifies the spacing condition following the last line of the report group.

General Format

```
                 ⎧ integer-1         ⎫
NEXT GROUP IS    ⎨ PLUS integer-2    ⎬
                 ⎩ NEXT PAGE         ⎭
```

MR-S-1032-81

Technical Notes

1. The NEXT GROUP clause can appear only at the 01 level of a report group.

2. Integer-1 and integer-2 must be positive integers with values less than 512. Integer-1 cannot exceed the number of lines specified by the PAGE LIMIT clause.

3. Integer-1 specifies a line number to which the LINE-COUNTER is set after the group is presented.

4. PLUS integer-2 specifies a relative line number that increments the LINE-COUNTER by the value of integer-2 after the group is presented. Integer-2 is the number of lines skipped following the last line of the report group.

5. NEXT PAGE indicates an automatic skip to the next page after the group is presented.

RESET

4.12.2.5 RESET

Function

The RESET clause indicates the CONTROL data-item that causes the SUM counter to be reset to zero on a control break.

General Format

$$\underline{\text{RESET}}\ \text{ON} \left\{ \begin{array}{l} \text{identifier-1} \\ \underline{\text{FINAL}} \end{array} \right\}$$

MR-S-1033-81

Technical Notes

1. Identifier-1 must be one of the identifiers associated with the CONTROL clause in the RD entry.

2. The RESET clause can be used only in conjunction with a SUM clause at a CONTROL FOOTING elementary level.

3. Identifier-1 must be a higher level (more major) control identifier than the control identifier associated with this report group.

4. After a TYPE CONTROL FOOTING report group is presented, the sum counters associated with that group are automatically set to zero, unless an explicit RESET clause directs that the counter be cleared at a higher level.

4.12.2.6 SOURCE

Function

The SOURCE clause indicates the source of the data for a report item.

General Format

```
    SOURCE IS identifier
```

Technical Notes

1. The SOURCE clause can only be given at the elementary level.
2. Identifier must indicate an item that appears in the FILE or WORKING-STORAGE SECTION.
3. When the report group is presented, the contents of this report item are replaced by the contents of identifier.

4.12.2.7 SUM

Function

The SUM clause indicates the items to be summed to produce the source of data for a report item.

General Format

```
SUM identifier-1 [,identifier-2] ... [UPON data-name-1]
```

MR-S-1034-81

Technical Notes

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

2. Each identifier must indicate a SOURCE item in a TYPE DETAIL report group, or a SUM counter in a TYPE CONTROL FOOTING report group.

3. If the SUM counter is referred to by a PROCEDURE DIVISION or REPORT SECTION statement, a data-name must be specified for the item. The data-name then represents the summation counter automatically generated by the Report Writer; that data-name does not represent the report group item itself.

4. A summation counter is incremented just before the presentation of the identifiers. Any editing of the SUM counters is done only when the sum item is presented; at all other times it is treated as a numeric item.

5. If higher-level report groups are indicated in the control hierarchy, each lower level that is figured into the sum is summed into the higher level before each lower level is reset, that is, counters are rolled forward prior to the reset operation.

6. The UPON option is required to obtain selective summation for a particular data item that is named as a SOURCE item in two or more TYPE DETAIL report groups. Identifier-1 and identifier-2 must be SOURCE data items in data-name-1; data-name-1 must be the name of a TYPE DETAIL report group.

7. When the UPON option is used, summation occurs only when a GENERATE statement references data-name-1. It does not occur during summary reporting (refer to the GENERATE statement in the PROCEDURE DIVISION).

8. The identifiers cannot be subscripted or indexed.

4.12.2.8 TYPE

Function

The TYPE clause specifies the particular type of report group that is described by this entry and indicates when the report group is generated.

General Format

```
          ( REPORT HEADING                                    )
          | RH                                                |
          | PAGE HEADING                                      |
          |                                                   |
          | PH  { CONTROL HEADING }  { identifier-n }         |
          |     { CH              }  { FINAL        }         |
          |                                                   |
TYPE IS  <  DETAIL                                            |
          |                                                   |
          | DE  { CONTROL FOOTING }  { identifier-n }         |
          |     { CF              }  { FINAL        }         |
          |                                                   |
          | PAGE FOOTING                                      |
          | PF                                                |
          | REPORT FOOTING                                    |
          ( RF                                                )
                                                  MR-S-1035-81
```

Technical Notes

1. RH is an abbreviation for REPORT HEADING;
   PH is an abbreviation for PAGE HEADING;
   CH is an abbreviation for CONTROL HEADING;
   DE is an abbreviation for DETAIL;
   CF is an abbreviation for CONTROL FOOTING;
   PF is an abbreviation for PAGE FOOTING;
   RF is an abbreviation for REPORT FOOTING.

2. If the report group is described as TYPE DETAIL, the GENERATE statement in the PROCEDURE DIVISION directs the Report Writer to produce the named report group.

3. The REPORT HEADING entry indicates a report group that is produced only once at the beginning of a report, during the execution of the first GENERATE statement. There can be only one report group of this type in a report.

4. The PAGE HEADING entry indicates a report group that is automatically produced at the beginning of each page of the report. There can be only one report group of this type in a report.

5. The CONTROL HEADING entry indicates a report group that is produced at the beginning of a control group for a designated identifier. In the case of FINAL, it is produced once before the first control group during the execution of the first GENERATE statement. There can be only one report group of this type for each identifier and for FINAL.

TYPE (Cont.)

6. The CONTROL FOOTING entry indicates a report group that is produced at the end of a control group for a designated identifier, or that is produced only once at the termination of a report in the case of FINAL. There can be only one report group of this type for each identifier and for FINAL. In order to produce any CONTROL FOOTING report groups, a control break must occur.

7. The PAGE FOOTING entry indicates a report group that is automatically produced at the bottom of each page of the report. There can be only one report group of this type in a report.

8. The REPORT FOOTING entry indicates a report group that is produced only once, at the termination of a report. There can be only one report group of this type in a report.

9. Each identifier, as well as FINAL, must be one of the identifiers associated with the CONTROL clause in the RD entry.

CHAPTER 5

THE PROCEDURE DIVISION

The Procedure Division specifies the processing to be performed on the files and the file data described in the Environment and Data Divisions. The Procedure Division contains a series of COBOL procedure statements which describe the processing to be done. Statements, sentences, paragraphs, and sections are described in Section 5.1. Sections are optional and permit a group of consecutive paragraphs to be referenced by a single procedure-name. Sections can also be used for segmentation purposes (see Section 5.3, Segmentation). If any section appears in the Procedure Division, then all paragraphs must appear within a section.

The first entry in the Procedure Division of a source program must be the division-header. The next entry must be either the DECLARATIVES header (see the USE statement, Section 5.9.42), or a paragraph-name or section-name.

```
 PROCEDURE DIVISION [USING data-name-1  [data-name-2]  ...]
[DECLARATIVES.
{section-name SECTION  [segment-number]  .  declarative-sentence
[paragraph-name. [sentence]  ...]  ...}  ...
 END DECLARATIVES.]
{section-name SECTION  [segment-number]  .
[paragraph-name. [sentence]  ...]  ...}  ...
                                          MR-S-1036-81
```

Only in a subprogram can USING clauses appear in the PROCEDURE DIVISION header.

When a program-name is specified in a CALL statement in a calling program, control is transferred to the beginning of the executable code in the subprogram (that is, the Procedure Division).

The identifiers in the USING clause indicate those data items in the called program that can reference data items in the calling program. The order of identifiers in the CALL statement of the calling program and in the PROCEDURE DIVISION header of the called program is critical. The items in the USING clauses are related by their corresponding positions, not by name. Corresponding identifiers refer to a single set of data that is available to both the calling and called programs.

The number of identifiers in the USING clause in the PROCEDURE DIVISION header must be less than or equal to the number of identifiers in the USING clause in the CALL statement in the calling program.

5.1 SYNTACTIC FORMAT OF THE PROCEDURE DIVISION

The PROCEDURE DIVISION consists of a series of procedure statements grouped into sentences, paragraphs, and sections. By grouping the statements in this manner, reference can be made to them by a procedure-name (that is, a paragraph-name or a section-name). The order in which procedure-statements are executed can be controlled by using the sequence-control verbs ALTER, GO TO, and PERFORM.

5.1.1 Statements and Sentences

Statements fall into three categories: imperative, conditional, and compiler-directing, depending upon the verb used. Verbs, in turn, are also classified into certain categories. These categories and their relationship to the three statement categories are given in Table 5-1.

Table 5-1
Procedure Verb and Statement Categories

Verb	Verb Category	Statement Category
ADD COMPUTE DIVIDE MULTIPLY SUBTRACT	ARITHMETIC	IMPERATIVE
ALTER CALL ENTER ENTRY EXIT PROGRAM GOBACK GO TO PERFORM STOP	SEQUENCE-CONTROL	IMPERATIVE
EXAMINE MOVE SET STRING UNSTRING	DATA MOVEMENT	IMPERATIVE

Table 5-1 (Cont.)
Procedure Verb and Statement Categories

Verb	Verb Category	Statement Category
CANCEL EXAMINE FREE MERGE RELEASE RETAIN RETURN SEARCH SORT TRACE	MISCELLANEOUS	IMPERATIVE
GENERATE INITIATE SUPPRESS TERMINATE	REPORT	IMPERATIVE
ACCEPT CLOSE DELETE DISPLAY OPEN READ REWRITE SEEK WRITE	I-O	IMPERATIVE
IF	CONDITIONAL	CONDITIONAL
COPY EXIT NOTE USE	COMPILER-DIRECTING	COMPILER-DIRECTING

5.1.2 Sentences

A statement or sequence of statements terminated by a period forms a sentence. Sentences are classified into the same three categories as statements.

An imperative sentence consists solely of one or more imperative statements. Except for imperative sentences containing one of the sequence-control verbs, control passes to the next procedural sentence following execution of the imperative sentence. If a GO TO or STOP RUN statement is present in an imperative sentence, it must be the last statement in the sentence.

A conditional sentence performs some test and, on the basis of the results of that test, determines whether a true or a false path should be taken. A conditional sentence is one that contains the conditional verb (IF) or one of the option clauses ON SIZE ERROR (used with arithmetic verbs), AT END (used with the READ verb), or INVALID KEY (used with the READ verb for mass storage devices).

A compiler-directing sentence consists of a single compiler-directing statement. Compiler-directing sentences are used to indicate the end point of a PERFORM loop (EXIT), insert comments in the PROCEDURE DIVISION (NOTE), copy library entries (COPY) and specify procedures for input-output errors and label handling (USE). Generally, compiler-directing sentences generate no object program code.

5.1.3 **Paragraphs**

A single sentence or a group of sequential sentences can be assigned a paragraph-name for reference. The paragraph-name must begin in Area A (see Chapter 1) and terminate with a period. The first sentence of the paragraph can begin after the space following this period or it can begin on the next line, beginning in Area B.

A paragraph-name must be unique within its section, but need not be unique within the program. A non-unique paragraph-name must be qualified by its section-name except when it is referenced from within its own section.

5.1.4 **Sections**

A single paragraph or a group of sequential paragraphs can be assigned a section-name for reference. The section-name must begin in Area A, be followed by the word SECTION, and optionally, followed by a priority number, and terminated by a period.

```
     section-name SECTION nn.
```

If the section-name is in the DECLARATIVES portion, it can not have a priority number. A USE statement can appear following the terminating space after the period.

The section-name applies to all paragraphs following it until another section-header is encountered.

All section-names must be unique within a program. Sections are optional within the PROCEDURE DIVISION, but if a DECLARATIVES portion is used there must be a named section immediately following the END DECLARATIVES statement.

When a section-name is referenced, the word SECTION is not allowed in the reference.

5.2 SEQUENCE OF EXECUTION

In the absence of sequence-control verbs, sentences are executed consecutively within paragraphs, paragraphs are executed consecutively within sections, and sections are executed consecutively within the PROCEDURE DIVISION (with the exception of sections within the DECLARATIVES portion, which are executed individually when the related condition occurs).

5.3 SEGMENTATION AND SECTION-NAME PRIORITY NUMBERS

COBOL source programs can be written to enable certain portions of the PROCEDURE DIVISION code to share the same memory area at object run time, thus decreasing the amount of memory required to run the object program, though not the time. The method used to achieve this reduction is called segmentation.

Segmentation consists of dividing the PROCEDURE DIVISION sections into logically related groupings called segments. The programmer defines a segment by assigning the same priority-number (a priority-number follows the word SECTION in the section-header, and can be in the range 00 through 99) to all the sections to be included in that segment; these sections need not appear consecutively in the source program.

Segments are classified into three groups, depending upon their priority-number. These three groups are described in Table 5-2.

Table 5-2
Types of Segments

Priority Number	Type	Description
None, or 00 up to SEGMENT-LIMIT minus 1	Resident Segment	This segment is always resident in memory and is never overlaid.
SEGMENT-LIMIT up to 49	Nonresident; ALTERed GO TOs retained	These segments are nonresident and are brought into memory when needed. Any ALTERed GO TOs retain their most recently set values.
50 through 99	Nonresident; ALTERed GO TOs reset	These segments are also nonresident and are brought into memory when needed. Any ALTERed GO TOs do not retain their latest values, but are reset to their original setting each time the segment is entered from another segment.

In addition to the resident segment, all data areas described in the DATA DIVISION are resident at all times. Thus, memory can be thought of as being divided into two parts:

1. A resident area, in which reside all data areas and the resident segment, and

2. A nonresident area, equal to the size of the largest nonresident segment, into which each nonresident segment is read when needed. Since each nonresident segment reads into the same memory area, any previous nonresident segment in that area is overlaid and must be brought in again when it is to be executed again.

The resident segment should consist of those sections that constitute the main portion of the processing. Infrequently-used sections can be allocated to the nonresident segments.

NOTE

Non-resident code can never be shareable.

5.4 ARITHMETIC EXPRESSIONS

An arithmetic expression is an identifier of a numeric elementary item, or a numeric literal, or such identifiers and literals separated by arithmetic operators.

Algebraic negation can be indicated by a unary minus symbol.

5.4.1 Arithmetic Operators

There are five arithmetic operators that can be used in arithmetic expressions. They are represented by specific character symbols that must be preceded by a space and followed by a space.

Arithmetic Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
**	Exponentiation
^	Exponentiation

5.4.2 Formation and Evaluation Rules

The following rules for information and evaluation apply to arithmetic expressions.

1. Parentheses specify the order in which elements within an arithmetic expression are to be evaluated. Expressions within parentheses are evaluated first. Within a nest of parentheses, the evaluation proceeds from the elements within the innermost pair of parentheses to the outermost pair of parentheses. When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchal order of operations is implied:

First:	unary +, unary -	
then	** and ^	(exponentiation)
then	* and /	(multiplication and division)
and then	+ and -	(addition and subtraction)

2. When the order of a sequence of operations on the same hierarchal level (for example, a sequence of + and - operations) is not completely specified by use of parentheses, the order of operations is from left to right.

3. An arithmetic expression can begin with one of the following:

   (- + variable

   and can end only with one of the following:

) variable

4. There must be a one-to-one correspondence between left and right parentheses in an arithmetic expression; each left parenthesis must precede its corresponding right parenthesis.

5.5 CONDITIONAL EXPRESSIONS

A conditional expression causes the object program to select between alternate paths (called the true and false paths) of control depending upon the truth value of a test. Conditional expressions can be used in conditional (IF) statements and in PERFORM statements (options 3 and 4). A conditional expression can be one of the following types:

Relation condition	(greater than, equal to, less than)
Class condition	(numeric or alphabetic)
Condition-name condition	(level-88 condition-names)
Switch-status condition	(SPECIAL-NAMES paragraph)
Sign condition	(positive, negative, zero)

Each of these types is discussed below.

5.5.1 Relation Condition

A relation condition causes a comparison of two operands, each of which can be an identifier, a literal, a figurative constant, or an arithmetic expression. Comparison of two numeric operands is permitted regardless of their formats as described by their respective USAGE clauses. Comparison of two operands is permitted if each is DISPLAY-6, DISPLAY-7, or DISPLAY-9.

A numeric-edited operand can not be compared to a numeric operand. An alphanumeric operand can not be compared to a numeric operand unless the alphanumeric operand contains no characters other than numeric digits. For example, the statement:

```
    IF NUM < "2".
```

is permissible but the statement:

```
    IF NUM < "2.0".
```

is not.

5.5.1.1 **Format of a Relation-Condition** - The general format for a relation condition is

$$\left\{\begin{array}{l}\text{identifier-1}\\ \text{literal-1}\\ \text{arithmetic-expression-1}\\ \text{figurative-constant-1}\end{array}\right\} \quad \text{relational-operator} \quad \left\{\begin{array}{l}\text{identifier-2}\\ \text{literal-2}\\ \text{arithmetic-expression-2}\\ \text{figurative-constant-2}\end{array}\right\}$$

MR-S-1037-81

The first operand is called the subject of the condition; the second operand is called the object of the condition. Either the subject or the object must be an identifier or an arithmetic expression.

5.5.1.2 **Relational Operators** - Relational operators specify the type of comparison to be made in the relation condition. Relational operators must be preceded by a space and followed by a space.

Relational Operator	Meaning
IS [NOT] GREATER THAN IS [NOT] > THAN	Greater than, not greater than
IS [NOT] LESS THAN IS [NOT] < THAN	Less than, not less than
IS [NOT] EQUAL (EQUALS) TO IS [NOT] = TO	Equal to, not equal to

5.5.1.3 **Comparison of Numeric Items** - A comparison between two numeric items determines that the algebraic value of one item is less than, equal to, or greater than the algebraic value of the other item. The length of the operands is not significant. Zero is considered a unique value; +0 and -0 are equal. Unsigned operands are considered positive. Blanks and tabs are ignored when a numeric item is compared to zero. Since blanks and tabs make an item alphanumeric, a true zero condition can be established by an alphanumeric test followed by a comparison with zero.

5.5.1.4 **Comparison of Alphanumeric Items** - For operands whose category is alphanumeric (or where one operand is numeric and the other is alphanumeric), a comparison results in the determination that one of the operands is less than, equal to, or greater than the other operand with respect to a specified collating sequence of characters (see Appendix B). The size of an operand is the total number of characters in the operand. Blanks and tabs are not ignored when an alphanumeric item is compared to ZERO. The presence of either blanks, tabs, or both in the operand causes the test result to be NOT EQUAL.

There are three cases to consider: operands of equal length, operands of unequal length, and operands with differing justification.

1. Operands of equal length - If the operands are of equal length, characters in corresponding character positions of the two operands are compared, starting at the higher-order (leftmost) end and continuing through the low-order end. If all pairs of characters compare equally through the last pair, the operands are considered to be equal. If they do not all compare equally, the first pair of unequal characters encountered is compared to determine their relative position in the collating sequence. The operand containing the character that is positioned higher in the collating sequence is considered to be the greater operand.

2. Operands of unequal length - If the operands are of unequal length, the comparison of characters proceeds from the high-order end to the low-order end until either

   a. A pair of unequal characters is encountered, or

   b. One of the operands has no more characters to compare.

   If a pair of unequal characters is encountered, the comparison is determined in the manner described for equal-sized operands.

   If the end of one of the operands is encountered before unequal characters are encountered, this shorter operand is considered to be less than the longer operand unless the remaining characters in the longer operand are spaces, in which case the two operands are considered equal.

3. If one operand is right-justified and the other is left-justified, they are compared just as they appear in the record. That is, PICTURE XXX, VALUE "B" and PICTURE XXX, VALUE "B", JUSTIFIED RIGHT are not equal because the first appears in the record as B   and the second as   B.

5.5.2 Class Condition

The class condition tests the contents of an item for being wholly alphabetic or wholly numeric.

5.5.2.1 Format of a Class Condition -

identifier IS $\left[\underline{\text{NOT}}\right]\left\{\begin{array}{l}\underline{\text{ALPHABETIC}}\\ \underline{\text{NUMERIC}}\end{array}\right\}$

MR-S-1038-81

5.5.2.2 **Restrictions** - The item named by identifier must be described, implicitly or explicitly, as DISPLAY, DISPLAY-6, DISPLAY-7, or DISPLAY-9. The NUMERIC test cannot be applied to an item described as alphabetic. The ALPHABETIC test cannot be applied to an item described as numeric. A compiler diagnostic results if either of the two previously mentioned tests are attempted.

5.5.2.3 **The ALPHABETIC Test** - The ALPHABETIC test result is TRUE when the item consists of characters from the alphabet (A through Z) and the space or tab.

5.5.2.4 **The NUMERIC Test** - The NUMERIC test result is TRUE under the following conditions:

1. For alphanumeric and unsigned numeric items, each character must be a digit (0 through 9). No signs are permitted. Spaces and tabs cause the test result to be FALSE.
2. For signed numeric items, the sign can have only one of the three following representations: a leading graphic sign ("+" or "-"), a trailing graphic sign, or a trailing embedded sign. All other characters must be digits. Spaces or tabs cause the test result to be FALSE.

NOTE

An alternative form of NUMERIC test can be selected by a switch setting during system installation, which causes leading and trailing blanks and tabs to be ignored. This alternative form is described in Appendix D.

5.5.3 Condition-Name Condition

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name (level-88).

5.5.3.1 **Format of a Condition-Name** - The general format for a condition-name is

[<u>NOT</u>] condition-name

If the condition-name is associated with a range of values, then the conditional variable is tested to determine whether or not its value falls within this range, including the end values.

The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

The result of the test is true if one of the values associated with the condition-name equals the value of its associated conditional variable.

5.5.4 Switch-status Condition

A switch-status condition determines the on or off status of a hardware switch.

5.5.4.1 Format of a Switch-Status Condition

Format of a Switch-Status Condition - The general formats for a switch-status condition are

Format 1:

$\left[\underline{\text{NOT}}\right]$ condition-name

Format 2:

mnemonic-name IS $\left[\underline{\text{NOT}}\right] \left\{ \begin{matrix} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{matrix} \right\}$

Format 3:

$\underline{\text{SWITCH}}$ (integer) IS $\left[\underline{\text{NOT}}\right] \left\{ \begin{matrix} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{matrix} \right\}$

MR-S-1039-81

In format 1, condition-name is associated with a SWITCH IS ON or OFF STATUS clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

In format 2, mnemonic-name is associated with a SWITCH (not an ON or OFF STATUS) in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

In format 3, integer must be in the range from 0 through 35.

In format 1, the result of the test is true if the switch is [NOT] set to the position associated with the condition-name.

In formats 2 and 3, the result of the test is true if the switch is [NOT] set to the position specified in the condition.

5.5.5 Sign Condition

The sign condition determines whether or not the algebraic value of a numeric operand is less than, greater than, or equal to zero.

5.5.5.1 **Format of a Sign Condition** - The general format for a sign condition is:

$$\left\{\begin{array}{l}\text{identifier}\\ \text{arithmetic-expression}\end{array}\right\} \text{ IS } \left[\underline{\text{NOT}}\right] \left\{\begin{array}{l}\underline{\text{POSITIVE}}\\ \underline{\text{NEGATIVE}}\\ \underline{\text{ZERO}}\end{array}\right\}$$

MR-S-1040-81

The POSITIVE test result is TRUE if the identifier or arithmetic-expression is algebraically greater than zero. The NEGATIVE test result is TRUE if the identifier or arithmetic-expression is algebraically less than zero. The ZERO test result is TRUE if the identifier or arithmetic-expression is equal to zero or contains all spaces, all tabs, or a combination of spaces and tabs. However, any spaces or tabs makes an item alphanumeric.

5.5.6 Logical Operators

The interpretation of any of the above conditions is reversed by preceding the condition with the logical operator NOT. Any of the above types of conditions can be combined by either of two logical operators. A logical operator must be preceded by a space and followed by a space.

Logical Operator	Meaning
OR	Entire condition is true if either or both of the simple conditions are true.
AND	Entire condition is true if both of the simple conditions are true.

5.5.7 Formation and Evaluation Rules

A conditional expression can be composed of either a simple-condition or a compound-condition. A simple-condition is one that performs a single test. A compound-condition is one that contains a string of simple-conditions connected by the logical operators AND, OR. A compound-condition can contain any combination of types of conditional expressions (relational, class, condition-name, switch-status, and sign).

The evaluation rules for conditions are analogous to those given for arithmetic expressions, except that the following hierarchy applies:

arithmetic-expressions
all relational operators
NOT
AND
OR

Parentheses can be used either to improve readability or to override the hierarchy given above. Each set of conditions within a pair of parentheses is reduced to a single condition. When this is accomplished, reductions which cross parentheses are done.

You can use parentheses in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first; within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive set. In the absence of parentheses or when parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of execution is implied:

1st - Unary plus and minus
2nd - Exponentiation
3rd - Multiplication and division
4th - Addition and subtraction

NOTE

The precedence of unary minus over exponentiation is different from algebraic notation, and from some other programming languages. If the data-names A and B have the values 3 and 2 respectively, then the COBOL statement

```
COMPUTE C = - A ** B
```

yields C as 9 (not -9 as in algebra).

Examples:

1. Using parentheses for ease of reading.
   The following expression

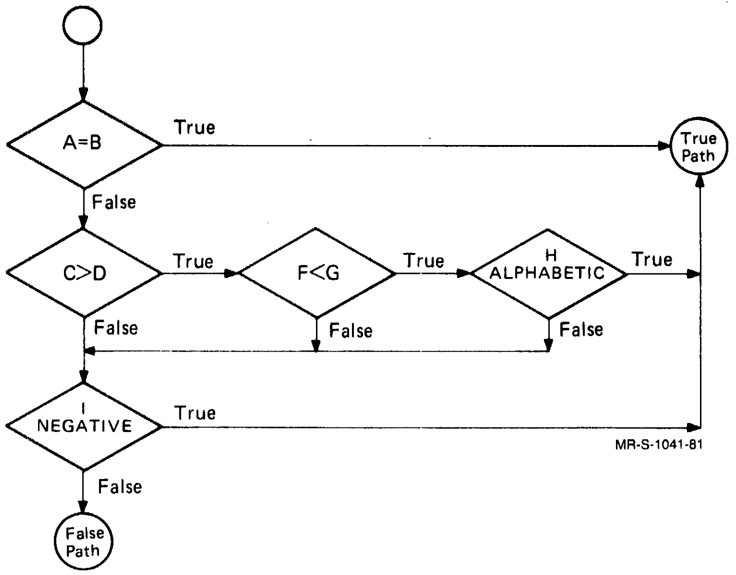
   ```
   A = B OR C > D AND F < G AND H IS ALPHABETIC OR I IS
   NEGATIVE
   ```

   can be parenthesized for readability without changing its effect as shown below:

   ```
   (A = B) OR (C > D AND F < G AND H IS ALPHABETIC) OR (I
   IS NEGATIVE)
   ```

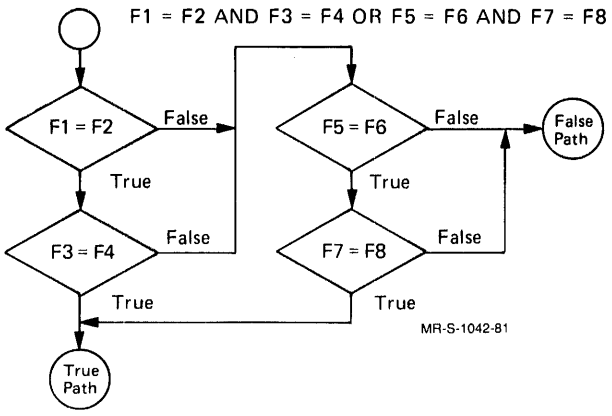
   If all the conditions within any of the three sets of parentheses are true, then the entire conditional expression is true.

   The diagram below illustrates the effect of this statement and the order of evaluation.

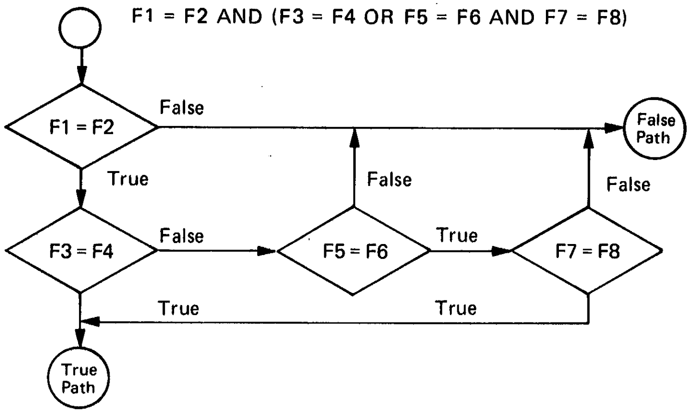


2. Using parentheses to override normal order of evaluation.

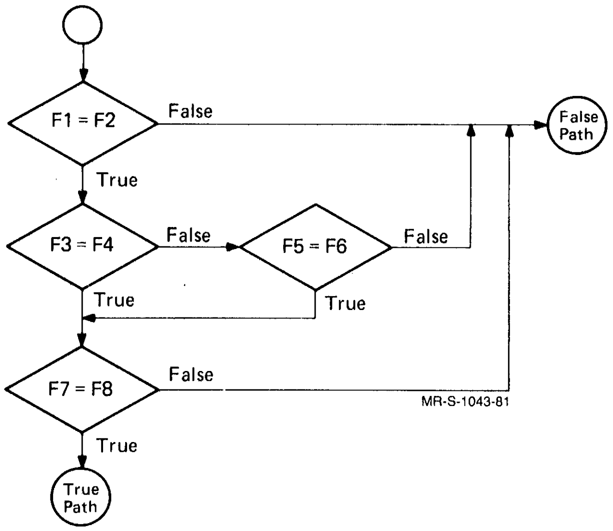
To illustrate this usage, a compound-conditional is shown in three forms, each accompanied by a flow diagram showing the result of each.



F1 = F2 AND (F3 = F4 OR F5 = F6 AND F7 = F8)



F1=F2 AND ((F3 = F4 OR F5 = F6) AND F7 = F8)



5.5.8 Abbreviations in Relation Conditions

When a string of consecutive relation conditions appears in a statement, abbreviations can be used, in certain cases, for any relation condition other than the first. The subject, or the subject and relational operator, or the subject, relational operator and logical connective can be omitted. In each of these cases, the effect of the abbreviated relation condition is as if the omitted parts were the same as those in the nearest preceding complete relation condition within the same sentence. There are two valid forms of abbreviation.

1. Abbreviation 1
   If the subject is identical in a series of relational conditions, it can be omitted in all the relational conditions except the first.
   Example: A = B OR A < C AND A = D OR A = E
   can be abbreviated to
   A = B OR < C AND = D OR = E

2. Abbreviation 2
   If subjects and relational operators are identical in a series of relational conditions, they can be omitted in all the relational conditions except the first.
   Example: A = B OR A = C AND A = D OR A = E
   can be abbreviated to
   A = B OR C AND D OR E

5.6 COMMON OPTIONS ASSOCIATED WITH THE ARITHMETIC VERBS

Associated with the five arithmetic verbs (ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT) are two options: the ROUNDED option, and the ON SIZE ERROR option. These two options are described here to avoid the necessity of including their descriptions with each of the arithmetic verbs.

If the ROUNDED option is specified, the absolute value of the item is increased by 1 if the leftmost truncated digit is greater than or equal to 5.

Example:	value:	567.8756
	resultant-identifier picture:	999V99
	stored result without ROUNDED option:	567.87
	stored result with ROUNDED option:	567.88

When the low-order positions in a resultant-identifier are represented by the symbol P in the PICTURE associated with the resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

Example:	value:	5388
	resultant-identifier picture:	99PP
	stored result without ROUNDED option:	53
	stored result with ROUNDED option:	54

5.6.1 The ON SIZE ERROR Option

If, after decimal point alignment, the number of significant digits in the result of an arithmetic operation is greater than the number of integer positions provided in the result-identifier, a size error condition occurs. Division by zero always causes a size error condition. The size error condition applies to both the intermediate results and the final result of an arithmetic operation. If the ROUNDED option is specified, rounding takes place before checking for size error. When such a size error does occur, the subsequent action depends upon whether or not the ON SIZE ERROR option is specified.

If the ON SIZE ERROR is not specified and a size error condition occurs, the value of the resultant-identifier is unpredictable, and no additional action is taken.

If ON SIZE ERROR is specified, and a size error condition occurs, then the values of the resultant-identifier(s) affected by the size errors are not altered. Values for resultant-identifier(s) for which no size error condition occurs are unaffected by size errors that occur for other resultant-identifier(s). After completion of the execution of the arithmetic operation, the statement(s) after ON SIZE ERROR is executed.

Example:	ADD A TO B ON SIZE ERROR GO TO OVERFLW	
	A:	954
	B:	PICTURE IS 999; VALUE 954.
	Result:	The contents of B are left unchanged and control is transferred to the paragraph or section named OVERFLW

5.7 THE CORRESPONDING OPTION

The CORRESPONDING option is used in the formats of two of the arithmetic verbs (ADD and SUBTRACT) and in the format of the MOVE verb.

For the purpose of this discussion, d(1) and d(2) represent identifiers that refer to group items. A pair of data items, one from d(1) and one from d(2), correspond if the following conditions exist:

1. All possible qualifiers for d(1) up to but not including d(1), must be identical to all possible qualifiers for d(2), up to but not including d(2).

2. Both of the data items are elementary numeric data items in the case of an ADD or SUBTRACT statement with the CORRESPONDING option.

3. Neither d(1) nor d(2) can be data items with level-number 66, 77, or 88.

4. Each data item subordinate to d(1) or d(2) that contains a RENAMES, a REDEFINES or an OCCURS clause is ignored. However, d(1) and d(2) can have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

See ADD, MOVE, and SUBTRACT; Sections 5.9.2, 5.9.23, and 5.9.39 respectively for information on the specific formats and results of the use of the CORRESPONDING option.

5.8 DETERMINATION OF USAGE IN ARITHMETIC COMPUTATIONS

If a programmer describes a numeric field as having USAGE DISPLAY-6, DISPLAY-7, DISPLAY-9, or COMP-3, the compiler converts this data to fixed-point binary when performing arithmetic computations with it. If the field contains 10 or fewer digits, it is converted to single-precision fixed-point binary. Conversion to double-precision fixed-point binary is performed if the field contains more than 10 digits. A field described as COMPUTATIONAL (or INDEX) is fixed-point binary; single-precision for 10 or fewer digits, double-precision for more than 10 digits. A field described as COMPUTATIONAL-1 is single precision floating-point binary.

When any arithmetic computation is performed, the arithmetic usage (single-precision fixed-point, double-precision fixed-point, or floating-point) used for each operation is determined from the usages of the two operands of the computation. If either operand is floating-point, the operation is performed in floating-point arithmetic. If neither operand is floating-point, but one operand is double-precision fixed-point, the operation is performed in double-precision fixed-point arithmetic. Otherwise, the operation is performed in single-precision fixed-point arithmetic. If both operands are constants, the operation is performed in single- or double-precision fixed-point arithmetic, as appropriate.

On KL or KS hardware, ADD, SUBTRACT, MULTIPLY, or DIVIDE is done in four-word fixed point arithmetic, if the size of the intermediate result exceeds 18 digits. In the COMPUTE verb, double-precision floating-point arithmetic is used if division or exponentiation is performed or if the intermediate result exceeds 18 digits.

If any alphanumeric characters appear in the DISPLAY-6, DISPLAY-7, or DISPLAY-9 field that is to be converted, the compiler attempts to convert them to binary; however, in many cases, undefined results can occur. When DISPLAY-6, DISPLAY-7, and DISPLAY-9 characters are converted to binary, the following rules apply.

0 through 9	are converted to 0 through 9.
A through I	are converted to 1 through 9.
?,[,{	are converted to 0.
J through R	are converted to 1 through 9, and the field is made negative if it is found in the low-order digit, unless an explicit sign is present.
:,!,],}	are converted to 0, and the field is made negative if it is found in the low-order digit unless an explicit sign is present.
Nulls	are ignored.
Leading spaces and tabs	are ignored.
+ and -	are treated as sign characters.

Scanning of a field proceeds from left to right, it stops when one of the following conditions is met:

1. The entire field has been scanned.

2. A trailing space, tab, plus, or minus is seen.

If both leading and trailing signs appear in the field, the trailing sign is ignored.

5.9 PROCEDURE DIVISION VERB FORMATS

The format of each PROCEDURE DIVISION verb is given on the following pages. The verbs are presented in alphabetical order.

The word "identifier" is a data-name followed, as required, by any qualification, subscripts, and/or indexes necessary to make the data-name unique.

ACCEPT

5.9.1 ACCEPT

Function

Option 1 of the ACCEPT statement causes low-volume data to be read from the terminal.

Option 2 of the ACCEPT statement causes the MESSAGE COUNT field to be updated to include the number of messages in a queue or sub-queue maintained by MCS-10, DIGITAL'S Message Control System for TOPS-10.

General Format

Option 1.

```
    ACCEPT   identifier-1   [,identifier-2]  ...  [FROM mnemonic-name]
```

Option 2.

```
    ACCEPT   cd-name   MESSAGE COUNT  MR-S-1044-81
```

Technical Notes

1. The ACCEPT statement causes the next set of data available from the terminal to replace the contents of the items named by identifier-1, identifier-2,... .
2. If the FROM option is specified, the mnemonic-name must appear in the CONSOLE IS clause of the SPECIAL-NAMES paragraph.
3. When the data to be read for one or more ACCEPT statements is numeric, a comma (,), space, or tab is used as a delimiter separating the data items.
4. When the data to be read for one or more ACCEPT statements is alphanumeric, each data item is delimited by a line-feed, altmode, form-feed, or vertical tab.
5. The ACCEPT statement reads from left to right a maximum of 1023 characters into each identifier. If the identifier is 1023 or less characters in size, then depending upon how many characters are input from the terminal, the following occurs:
    a. Less than the identifier - the identifier is left justified and the rest is filled with spaces.
    b. Exactly the size of the identifier - the identifier is filled.
    c. More than the identifier - the identifier is left justified and the rest is truncated.

ACCEPT (Cont.)

If the identifier is greater than 1023 characters in size, then the above holds true for the first 1023 characters of the identifier. The remaining characters of the identifier are not changed, no matter how many characters are typed on the terminal.

6. Upon execution of the ACCEPT MESSAGE COUNT statement, the contents of the area specified by a communication description entry must contain at least the name of the symbolic queue to be tested. Testing the condition causes the contents of the data items replaced by data-name-ID (status key) and data-name-2 (MESSAGE COUNT) of the areas associated with the communications entry to be appropriately updated.

ADD

5.9.2 ADD

Function

The ADD statement computes the sum of two more numeric operands and stores the result.

General Format

Option 1:

```
ADD { identifier-1 }  [ , { identifier-2 } ] ... TO identifier-m [ ROUNDED ]
    { literal-1    }  [   { literal-2    } ]

    [ ,identifier-n [ ROUNDED ] ] ...

    [ ON SIZE ERROR statement-1 [ ,statement-2 ] ... . ]
```

Option 2:

```
ADD { identifier-1 } , { identifier-2 }  [ , { identifier-3 } ] ...
    { literal-1    }   { literal-2    }  [   { literal-3    } ]

    GIVING identifier-m [ ROUNDED ] [ ,identifier-n [ ROUNDED ] ] ...

    [ ON SIZE ERROR statement-1 [ ,statement-2 ] ... . ]
```

Option 3:

```
ADD { CORRESPONDING } identifier-1 TO identifier-2
    { CORR          }

    [ ROUNDED ] [ ON SIZE ERROR statement-1 [ ,statement-2 ] ... . ]
```

MR-S-1045-81

ADD (Cont.)

Technical Notes

1. Each ADD statement must contain at least two operands (that is, an addend and an augend). In options 1 and 2, each identifier must refer to an elementary numeric item, except that identifiers appearing to the right of the word GIVING can refer to numeric edited items. In option 3, each identifier must refer to a group item.

   Each literal must be a numeric literal; the figurative constant ZERO is permitted.

2. The composite of all operands (i.e., the data item resulting from the superimposition of all operands aligned by decimal point) must not contain more than 19 decimal digits for the non-BIS compiler and not more than 36 digits for a BIS-compiler. In either case, a maximum of 18 digits can be stored in the receiving field.

   NOTE

   The BIS-compiler is standard on the DECSYSTEM-20 and DECsystem-10. For KI based hardware, the non-BIS compiler is optional on the DECsystem-10. (See the COBOL-68 Installation Procedures.)

3. Option 1 causes the values of the operands preceding the word TO to be algebraically summed. The resultant sum is then added to the current value of identifier-m and this result replaces the current value in identifier-m. If other identifiers follow, the same process is repeated for each of them.

4. Option 2 causes the values of the operands preceding the word GIVING to be algebraically summed. The resultant sum then replaces the current contents of identifier-m. If other identifiers follow, their contents are also replaced by this resultant sum. The current values of identifier-m, identifier-n,... do not enter into the arithmetic computation.

5. Option 3 causes the data items in the group item associated with identifier-1 to be added to the current value of the corresponding data items associated with identifier-2, and each result replaces the value of the corresponding data-items associated with identifier-2. The criteria used to determine whether two items are corresponding are described in Section 5.7, The CORRESPONDING Option.

6. The ROUNDED and ON SIZE ERROR options are described in Section 5.6 Common Options Associated with Arithmetic Verbs.

ALTER

5.9.3 ALTER

Function

The ALTER statement changes the object of one or more GO TO statements.

General Format

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

    [,procedure-name-3 TO PROCEED TO procedure-name-1] ...
```

MR-S-1046-81

Technical Notes

1. During execution of the object program, the ALTER statement modifies the GO TO statement in the paragraph named procedure-name-1, procedure-name-3, ... replacing the object of the GO TO by procedure-name-2, procedure-name-4, ..., respectively.

2. Each procedure-name-1, procedure-name-3,.... must be the name of a paragraph that contains only a single GO TO statement without the DEPENDING option.

3. Each procedure-name-2, procedure-name-4,... must be the name of a paragraph or section within the PROCEDURE DIVISION.

4. A GO TO statement in a section whose priority is greater than or equal to 50 must not be referred to by an ALTER statement in a section with a different priority.

5. An ALTER statement in a procedure not in the DECLARATIVES portion of the program can not reference a procedure name within the DECLARATIVES; conversely, an ALTER statement within the DECLARATIVES can not reference a procedure-name not in the DECLARATIVES.

6. Restrictions similar to those in Note 5 also apply to the INPUT PROCEDUREs and to the OUTPUT PROCEDUREs associated with SORT and MERGE verbs.

7. For program segments with priorities of 50 and greater, the changes made by ALTER statements are lost when segments are overlaid.

5.9.4 CALL

Function

The CALL statement is used to transfer control to a subprogram.

General Format

$$\underline{\text{CALL}}\ \left\{\begin{array}{l}\text{program-name}\\ \text{entry-name}\end{array}\right\}\ \left[\underline{\text{USING}}\ \text{identifier-1}\quad ,\text{identifier-2}\ \ldots\right]$$

$$\left[\text{ON}\ \underline{\text{OVERFLOW}}\ \text{imperative-statement-1}\right]\ \underline{.}$$

MR-S-1047-81

Technical Notes

1. Program-name is a 1-to-6 character name (PROGRAM-ID) of the subprogram to be called. Entry-name is a 1-to-6 character name of an entry point in the subprogram. Either name can be enclosed in quotation marks, but can contain only letters and digits.

2. If the program-name is used, the entry point is at the beginning of the executable code in the subprogram.

3. Called programs can call other subprograms, but a called program cannot call, either directly or indirectly, any part of itself or the program that called it.

4. The number of operands in the USING clause of the CALL statement must be greater than or equal to the number of operands in the ENTRY statement or PROCEDURE DIVISION header in the subprogram.

5. Each of the operands in the USING clause can be any item defined in the FILE, WORKING-STORAGE, or LINKAGE SECTION of the calling program. However, these items must be word-aligned; that is, they must begin on a word boundary. 01- and 77-level items are always word-aligned. Any other item can be word-aligned by means of the SYNCHRONIZED LEFT clause.

6. The identifiers in the USING clause indicate those data items in the calling program that can be referenced (or whose subordinate parts can be referenced) in the called program. The order of the identifiers in the CALL statement in the calling program and in the PROCEDURE DIVISION header or ENTRY statement of the calling program is critical. The items in the USING clause are related by their corresponding positions, not by name. Corresponding identifiers refer to a single set of data that is available to both the calling and called programs.

CALL (Cont.)

7. The first time a called program is entered, its state is that of a fresh copy. Subsequently, if the subprogram is not in a LINK overlay, its state when entered is exactly as it was left after the last exit from it. That is, all internal variables, altered GO TO's, and the like are exactly as they were left. However, external data (that is, data described in the LINKAGE SECTION) may have been changed since the last exit.

   If the subprogram is in a LINK overlay and it is entered again, its state is exactly as it was left after the last exit from it provided that the subprogram has not been cancelled by you or overlaid. If the subprogram has been cancelled or overlaid, its state is that of a fresh copy.

8. The ON OVERFLOW condition cannot happen if it is encountered within the CALL statement. It is merely shown here for ANSI compatability. If ON OVERFLOW is used, it is ignored and your CALL statement exits normally.

9. Refer to Chapter 11 of this manual for more information on subprograms.

5.9.5 CANCEL

Function

The CANCEL statement causes a subprogram to be logically disassociated from the main program and, if possible, causes the return of the memory used by the subprogram to the system.

General Format

```
CANCEL subprogram-1 [,subprogram-2] ... .
```

Technical Notes

1. The CANCEL statement can only be used to cancel a subprogram that has been loaded into an overlay link by LINK. Refer to Chapter 11 of this manual for information on specifying LINK overlays and on subprograms.

2. After a subprogram has been cancelled, a subsequent call to the subprogram causes a freshly-initialized copy to be brought into memory.

3. Cancellation of a subprogram causes the entire link in which it resides and all lower level links to be cancelled.

4. A subprogram in the root link or higher in the current overlay structure cannot be cancelled. If an attempt is made to do so, the CANCEL statement is ignored and a warning message issued at runtime.

5. A subprogram cannot cancel itself or any subprogram that resides in an overlay link with it. An attempt to do either results in the CANCEL statement being ignored and a warning message issued at runtime.

6. Cancellation of a subprogram higher in the current calling sequence is also an illegal operation. But, if the subprogram being cancelled is in a lower-level link and higher in the calling sequence, it could be cancelled without being detected as an error. This would cause the return from the program to be an undefined location.

Example

```
CANCEL SUBA,SUBC.
```

CLOSE

5.9.6 CLOSE

Function

The CLOSE statement terminates the processing of input and output files, reels, or units.

General Format

```
                 ┌┌REEL┐┐ ┌      ┌NO REWIND┐┐
CLOSE  file-name ││    ││ │WITH │LOCK     ││
                 └└UNIT┘┘ └      └DELETE   ┘┘

   ┌              ┌┌REEL┐┐ ┌      ┌NO REWIND┐┐┐
   │,file-name-1  ││    ││ │WITH │LOCK     │││
   └              └└UNIT┘┘ └      └DELETE   ┘┘┘
                                       MR-S-1048-81
```

Technical Notes

1. Each filename must appear as the subject of an FD entry in the FILE SECTION of the DATA DIVISION.

2. The REEL, UNIT, and NO REWIND options apply only to magnetic tape files. UNIT is synonymous with REEL.

3. The DELETE option applies only to disk and DECtape files. If this option is included, the file is deleted from the device.

4. For the purpose of showing the effect of various CLOSE options as applied to the various storage media, all input, output, and input-output files are divided into the following three mutually exclusive categories:

a. NON-REEL	A file whose device is such that the concepts of REWIND, REEL, or UNIT have no meaning. This category includes files residing on disk, punched cards, paper tape, line printer, and terminal.
b. SINGLE REEL	A file that is entirely contained on one reel or unit.
c. MULTI-REEL	A file that can be contained on more than one reel or unit.

The results of each CLOSE option for each of the above types of files are summarized in Table 5-3. The definitions for the symbols used in this table are given below. Where the definition depends upon whether the file is an input or output file, alternate definitions are given; otherwise, the single definition given applies to both input and output files.

CLOSE (Cont.)

Codes Used in Table 5-3

A =	Any subsequent reels of this file are not processed.
B =	The current reel is not rewound.

C = Standard CLOSE File Procedure

INPUT and I-O Files (see "OPEN")

If the file is positioned at its end, your ENDING FILE LABEL PROCEDUREs are performed, if you have specified any by a USE statement. An input file is considered to be at the end-of-file if the imperative-statement in the AT END clause of a READ for the file has been executed, and no CLOSE statement for the file has been executed.

OUTPUT Files

If LABEL RECORDS are STANDARD, an ending label is created and written on the output medium. Then, your ENDING FILE LABEL PROCEDUREs are performed.

D = The current reel is rewound and unloaded.

If you are using TOPS-20, the tape drive must be made unavailable to MOUNTR and ASSIGNed to your job in order for the reel to be unloaded.

E = Any attempt to subsequently OPEN this file results in an error message being typed and the run terminated.

F = Standard CLOSE REEL Procedure

INPUT Files

1. If the file is assigned to more than one device, the next device specified in the ASSIGN clause becomes the current device. If no other device is specified, the first device mentioned becomes the current device.

2. The standard beginning reel label procedure and your BEGINNING REEL LABEL PROCEDURE (specified in a USE statement) are performed for the new reel.

OUTPUT and I-O Files

1. The standard ending reel label procedure and your ENDING REEL LABEL PROCEDURE are performed.

2. If the file is assigned to more than one device, the devices are swapped. A halt occurs to allow you to mount an available reel.

3. The standard beginning reel label procedure and your BEGINNING REEL LABEL PROCEDURE (specified in a USE statement) are performed.

G = The tape is rewound.

CLOSE (Cont.)

Codes Used in Table 5-3

H =	The file is deleted from the device. However, if the file is a sequential file on disk that is open for output in supersede mode, the original file remains intact (that is, the original file is not superseded nor deleted).
X =	Illegal. This is an illegal combination of a CLOSE option and a file type.

5. If a file is OPENed but not CLOSEd before the STOP RUN statement is executed, the file is automatically CLOSEd. Any records still retained by a RETAIN statement are automatically freed by a CLOSE statement.

6. If the file has been specified with an OPTIONAL clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION and the file was not present for this run, the CLOSE has no effect.

7. If a CLOSE statement without the REEL or UNIT option has been executed for a file, a READ, WRITE, or CLOSE statement for that file must not be executed until another OPEN for that file has been executed.

Table 5-3
CLOSE Options and File Types

		File Type		
		NON-REEL	SINGLE REEL/UNIT	MULTI-REEL
CLOSE Option	CLOSE	C	C,G	C,G,A
	CLOSE WITH LOCK	C,E	C,G,E	C,G,E,A
	CLOSE WITH NO REWIND	X	C,B	C,B,A
	CLOSE REEL	X	X	F,G
	CLOSE REEL WITH LOCK	X	X	F,D
	CLOSE REEL WITH NO REWIND	X	X	F,B
	CLOSE WITH DELETE	C,H	X	X

5.9.7 COMPUTE

Function

The COMPUTE statement assigns to a data item the value of a numeric data item, literal, or arithmetic expression.

General Format

```
                                   { EQUALS   } { identifier-2          }
COMPUTE  identifier-1  [ROUNDED]   { EQUAL TO } { literal-1             }
                                   { =        } { arithmetic-expression }

     [ON  SIZE ERROR [statement-1  ,statement-2]  ...  .]
```

MR-S-1049-81

Technical Notes

1. The COMPUTE statement allows you to combine arithmetic operations without the restrictions on the composite of operands and/or receiving data items imposed by the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE. Division and exponentiation are always done using double-precision floating-point internal representations. This can cause problems for you if your result has seventeen or eighteen digits of precision; your answers could be low by a very small amount. If this is causing a problem for you, you should use the ROUNDED option.

   Exponentiation can only be done by using the COMPUTE statement.

2. Identifier-1 must be an elementary numeric or numeric edited item.

3. Identifier-2 must be an elementary numeric item. Literal-2 must be a numeric literal.

   The identifier-2 and literal-1 options provide a method for setting the value of identifier-1 equal to identifier-2 or literal-1.

4. The rules for forming arithmetic expressions and the order of evaluation are given earlier in this chapter under "Arithmetic Expressions".

5. The ROUNDED and SIZE ERROR options are described earlier in this chapter under "Common Options Associated with the Arithmetic Verbs".

5.9.8 DELETE

Function

The DELETE statement removes a specified record from a file whose access mode is INDEXED.

General Format

<u>DELETE</u> record-name <u>INVALID</u> KEY statement-1 [, statement-2] ... <u>.</u>

Technical Notes

1. Record-name must be a record associated with a file whose access mode is INDEXED.
2. When the DELETE statement is executed, the record in the file that has a key equal in value to the SYMBOLIC KEY for the file is removed from the file. If no such record exists, the statement(s) associated with the INVALID KEY clause is executed.
3. At the time that the DELETE statement is executed, the file must be open for OUTPUT or INPUT-OUTPUT.

5.9.9 DISPLAY

Function

The DISPLAY statement causes low-volume data to be written on your terminal.

General Format

```
DISPLAY {literal-1   } [ {literal-2   }] ...
        {identifier-1} [,{identifier-2}]

     [UPON mnemonic-name] [WITH NO ADVANCING]
```

MR-S-1050-81

Technical Notes

1. The contents of each operand are written on your terminal in the order listed.

2. Each of the literals can be numeric or alphanumeric, or one of the figurative constants. If a figurative constant is specified as one of the operands, only a single occurrence of that constant is written on the device.

3. The mnemonic-name must appear in the CONSOLE clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

4. If WITH NO ADVANCING is specified, the terminal does not advance to the next line. Thus, printing or type-in can continue on the same line.

5. If the identifier being displayed is numeric, commas are inserted automatically from the right. for example, if you code DISPLAY NUM-ONE and NUM-ONE contains 1234567890, then this appears on your terminal as 1,234,567,890.

DIVIDE

5.9.10 DIVIDE

Function

The DIVIDE statement divides one numeric item into another and sets the value of a data item equal to the result.

General Format

Option 1:

```
DIVIDE {identifier-1} INTO identifier-2 [ROUNDED] [REMAINDER identifier-4]
       {literal-1   }

    [ON SIZE ERROR statement-1 [,statement-2] ... .]
```

Option 2:

```
DIVIDE {identifier-2} BY identifier-1 [ROUNDED] [REMAINDER identifier-4]
       {literal-2   }

    [ON SIZE ERROR statement-1 [,statement-2] ... .]
```

Option 3:

```
DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3
       {literal-1   }      {literal-2   }

    [ROUNDED] [REMAINDER identifier-4]

    [ON SIZE ERROR statement-1 [,statement-2] ... .]
```

MR-S-1051-81

DIVIDE (Cont.)

Option 4:

```
DIVIDE {identifier-2} BY {identifier-1} GIVING identifier-3
       {literal-2   }    {literal-1   }

       [ROUNDED] [REMAINDER identifier-4]

       [ON SIZE ERROR statement-1 [,statement-2] ...] .
                                                   MR-S-1052-81
```

Technical Notes

1. The value of identifier-1 or literal-1 is divided into the value of identifier-2 or literal-2. In option 1, the resulting quotient replaces the value of identifier-2. In option 2, the resulting quotient replaces the value of identifier-1. In options 3 and 4, the resulting quotient replaces the value of identifier-3.

2. Each DIVIDE statement must contain two operands (that is, a dividend and a divisor). Both of these operands (identifier-1 and identifier-2) must refer to elementary numeric items. Identifier-3 can be an elementary numeric or numeric edited item. Each literal-1 or literal-2 must be a numeric literal. Identifier-4 can be an elementary numeric or numeric edited item.

3. The ROUNDED and SIZE ERROR options are described earlier in this chapter under "Common Options Associated with Arithmetic Verbs".

4. If the REMAINDER clause is used, the resulting remainder replaces the value of identifier-4.

5. The data item resulting from the divide operation (i.e., the sum of the digits in the dividend and the digits in the fractional part of the divisor) must not contain more than 20 decimal digits for the non-BIS compiler and not more than 36 digits for the BIS compiler. In either case, a maximum of 18 digits can be stored in the receiving field.

   NOTE

   The BIS compiler is standard for the DECSYSTEM-20 and DECsystem-10. For KI based hardware, the non-BIS compiler is optional on the DECsystem-10. (See the COBOL-68 Installation Procedures.)

6. The remainder is checked for a size error after the quotient is checked, whether or not the quotient has a size error. If either the quotient or the remainder has a size error, LIBOL follows the procedure described under "Common Options Associated with Arithmetic Verbs".

7. The ROUNDED option does not apply to the remainder; the remainder is always truncated.

ENTER

5.9.11 ENTER

Function

The ENTER statement allows the execution of MACRO and FORTRAN subroutines in conjunction with the COBOL program.

General Format

```
       { MACRO   }
ENTER  { FORTRAN }  program-name
       { COBOL   }

 [       { identifier-1     }   [   { identifier-2     } ]     ]
 [ USING { literal-1        }   [ , { literal-2        } ] ... ]
 [       { procedure-name-1 }   [   { procedure-name-2 } ]     ]

              [ON OVERFLOW imperative-statement-1] .
                                          MR-S-1053-81
```

Technical Notes

1. MACRO refers to MACRO assembly language and FORTRAN refers to the FORTRAN language.
2. The program-name can be enclosed in quotation marks.
3. The ENTER statement generates a subroutine call followed by the address in which the items associated with the USING clause are located. (Refer to Chapter 12 for more information on the ENTER statement.)
4. The ON OVERFLOW condition cannot happen if it is encountered within the ENTER statement. It is merely shown here for ANSI compatability. If ON OVERFLOW is used, it is ignored and your ENTER statement exits normally.
5. ENTER COBOL is equivalent to CALL.

5.9.12 ENTRY

Function

The ENTRY statement establishes an entry point in a subprogram.

General Format

<u>ENTRY</u> entry-name [<u>USING</u> identifier-1 [,identifier-2] ... <u>.</u>

Technical Notes

1. The ENTRY statement can only be used in a subprogram.
2. Control is passed to the entry point by a CALL statement in a calling program.
3. Entry-name is a 1-to-6 character name that can contain only letters and digits. It can, however, be enclosed in quotation marks. This name must not be the same as any other entry-name or PROGRAM-ID in any program with which the subprogram containing it is loaded.
4. The identifiers listed in the USING clause must be defined as 01- or 77-level items in the LINKAGE SECTION of the subprogram containing the ENTRY statement.
5. The number of operands in the USING clause of an ENTRY statement must be less than or equal to the number of operands in any CALL statement referencing that ENTRY statement.
6. The identifiers in the USING clause indicate those data items in the called program that can reference data items in the calling program. The order of identifiers in the CALL statement in the calling program and in the ENTRY statement in the called program is critical. The items in the USING clauses are related by their corresponding positions, not by name. Corresponding identifiers refer to a single set of data that is available to both the calling and called programs.
7. At run-time, additional ENTRY statements are ignored unless there are specific calls made to them. for example, if a subprogram has three ENTRY statements and a call is made to the first ENTRY statement, the remaining two ENTRY statements are ignored.
8. Refer to Chapter 11 of this manual for more information on subprograms.

EXAMINE

5.9.13 EXAMINE

Function

The EXAMINE statement replaces or counts the number of occurrences of a given character in a data item.

General Format

```
EXAMINE identifier

 ⎧             ⎧ALL        ⎫                                        ⎫
 ⎪ TALLYING    ⎨LEADING    ⎬  literal-1  [REPLACING BY literal-2]  ⎪
 ⎪             ⎩UNTIL FIRST⎭                                        ⎪
 ⎨                                                                  ⎬
 ⎪             ⎧ALL          ⎫                                      ⎪
 ⎪ REPLACING   ⎨LEADING      ⎬  literal-1  BY  literal-2            ⎪
 ⎩             ⎩[UNTIL] FIRST⎭                                      ⎭
                                                          MR-S-1054-81
```

Technical Notes

1. The USAGE of identifier must be DISPLAY-6, DISPLAY-7, or DISPLAY-9, implicitly or explicitly.

2. Each literal must consist of a single character belonging to a class consistent with that of the identifier. A literal can be any figurative constant.

3. Examination starts at the leftmost character of the identifier and proceeds to the right.

4. When the TALLYING option is used, a count is kept of:

   a. Occurrences of literal-1 when the ALL option is used.

   b. Occurrences of literal-1 prior to a character other than literal-1 when the LEADING option is used.

   c. Characters prior to the first occurrence of literal-1 when the UNTIL FIRST option is used.

   This count replaces the contents of the special register called TALLY (see "Special Registers", Chapter 1). TALLY has a PICTURE of S9(5), and can be referenced in any statement where an identifier referring to an elementary numeric data item is valid.

   If the REPLACING BY clause is used with the TALLYING option, replacement is performed according to the rules below.

EXAMINE (Cont.)

5. When either of the REPLACING BY options are used, replacement rules are

   a. If the ALL option is used, literal-2 is substituted for each occurrence of literal-1.

   b. If the LEADING option is used, the substitution of literal-2 for literal-1 terminates as soon as a character other than literal-1 is encountered.

   c. If the UNTIL FIRST option is used, literal-2 is substituted for each character prior to the first occurrence of literal-1.

   d. If the FIRST option is used, literal-2 is substituted for only the first occurrence of literal-1.

6. If the identifier is classified as numeric, it must consist solely of numeric characters. It can possess an operational sign, but this sign is ignored by the EXAMINE process.

EXIT

5.9.14 EXIT

Function

The EXIT statement provides a common end point for a series of routines executed by a PERFORM or USE statement.

General Format

paragraph-name. <u>EXIT</u>.

Technical Notes

1. EXIT must be the first sentence in a paragraph. Only NOTE can follow.

2. The EXIT statement can be used to provide an end point for a series of paragraphs that are PERFORMed, or at the end of a section in the DECLARATIVES. By using EXIT at the end of the range of a PERFORM or USE, a variety of exits from the performed procedure can be accomplished by making each point at which an exit is required a transfer to the EXIT paragraph. However, unless EXIT is specified as the end of the range of a PERFORM or USE or is placed as the last paragraph in the range of a PERFORM or USE, it is ignored.

   Example:

   ```
   PERFORM TAX-ROUTINE THROUGH EXIT-RTE.
   .
   .
   .
   TAX-ROUTINE.
       IF TOTAL-TAX IS EQUAL TO OR GREATER THAN TAX-LIMIT
       GO TO EXIT-RTE.
       MULTIPLY.....
       .
       .
   DEDUCTION-RTE.
       IF NO-OF-DEPENDENTS IS EQUAL TO ZERO
       GO TO EXIT-RTE.
       MULTIPLY NO-OF-DEPENDENTS BY DEP-DEDUCT....
       .
       .
       .
   EXIT-RTE. EXIT.
   ```

3. If control reaches an EXIT statement and no associated PERFORM or USE statement is active or if EXIT is not the last paragraph in the range of a PERFORM or USE statement even if the PERFORM or USE statement is active, control passes through the EXIT paragraph to the first statement of the next paragraph.

5.9.15 EXIT PROGRAM

Function

The EXIT PROGRAM statement is used to return control from a subprogram to its calling program.

General Format

<u>EXIT PROGRAM</u>.

Technical Notes

1. EXIT PROGRAM can only appear in a subprogram.
2. When an EXIT PROGRAM statement is executed, control is returned to the calling program at the statement immediately following the CALL statement.
3. If an EXIT PROGRAM statement is encountered in a subprogram that is operating as a main program, it is ignored.
4. Refer to Chapter 11 of this manual for more information on subprograms.

FREE

5.9.16 FREE

Function

The FREE statement explicitly frees records that have been retained in a RETAIN statement.

General Format

```
       ⎧              ⎧ RECORD [KEY {identifier-1}] ⎫                 ⎫
       ⎪ file-name-1  ⎨             {literal-1   }  ⎬                 ⎪
       ⎪              ⎩ EVERY RECORD                ⎭                 ⎪
       ⎪                                                              ⎪
FREE   ⎨   ⎡               ⎧ RECORD [KEY {identifier-2}] ⎫ ⎤          ⎬
       ⎪   ⎢ ,file-name-2  ⎨             {literal-2   }  ⎬ ⎥          ⎪
       ⎪   ⎣               ⎩ EVERY RECORD                ⎭ ⎦          ⎪
       ⎪                                                              ⎪
       ⎩ EVERY RECORD                                                 ⎭

     [NOT RETAINED statement-1 [,statement-2] ...] .
```

(Underlined: FREE, KEY, EVERY, NOT RETAINED, and the final period.)

MR-S-1055-81

Technical Notes

1. Filename-1, filename-2... are the names of files containing records that have been retained. Thus, they are files that have been opened for simultaneous update.

2. Identifier-1, identifier-2... and literal-1, literal-2... specify the value of a key. This key refers to the record to be freed in the file.

3. Statement-1, statement-2... are any valid COBOL statements.

4. The FREE statement is needed to explicitly free records that have not been implicitly freed by an I/O statement. This could occur when the RETAIN statement contains the UNTIL FREED phrase, when an I/O statement is not issued after the RETAIN statement, or when the FOR clause of the RETAIN statement specifies ANY VERB. Refer to the RETAIN statement in this chapter for a description of its function and syntax.

5. The EVERY RECORD phrase is used to free all records retained or to free all records retained in a specific file.

FREE (Cont.)

6. The NOT RETAINED phrase specifies the COBOL statements to be executed when one or more records to be freed are not currently retained. If the NOT RETAINED phrase is not included and the records to be freed are not currently retained, the program proceeds and you are not notified of the possible error.

7. When an EVERY RECORD phrase is used, the statements in the NOT RETAINED phrase are executed only if no records are currently retained or only if no records are currently retained in the specified file.

8. If the FREE statement includes a file that was not opened for simultaneous update, the NOT RETAINED statements, if present, are executed. Otherwise, the program continues and you are not notified of the error.

9. You can mix records from sequential, random, and indexed sequential files in the same FREE statement.

10. All records of a file are freed automatically when the file is closed including those records that were retained with an UNTIL FREED clause in the RETAIN statement.

11. The record to be freed, whether or not the KEY phrase is specified, depends on the access mode of the file. Each access mode is described separately below.

    a. SEQUENTIAL ACCESS FILES

       If the KEY phrase is specified, its value refers to the record with that value in the RETAIN statement. That is, a KEY value of 6 in the FREE statement frees the record defined with a KEY value of 6 in the RETAIN statement.

       If the KEY phrase is not specified, the record freed is that record defined with a KEY value of 0 in the RETAIN statement.

       The value of a key can be specified by any identifier, which can be subscripted and/or qualified, provided that its USAGE is COMPUTATIONAL or INDEX. The value of the key can also be specified by a positive integer numeric literal containing ten or fewer digits.

    b. RANDOM ACCESS FILES

       If the KEY phrase is specified, its value refers to the record with that value in the RETAIN statement. That is, a KEY value of 0 in the FREE statement frees the record defined with a KEY value of 0 in the RETAIN statement.

       If the KEY phrase is not specified, the record freed is that record defined by the ACTUAL KEY of the file.

       The value of a key can be specified by any identifier, which can be subscripted and/or qualified, provided that its USAGE is COMPUTATIONAL or INDEX. The value of a key can also be specified by a positive integer numeric literal containing ten or fewer digits.

FREE (Cont.)

c. INDEXED SEQUENTIAL FILES

If the KEY phrase is specified, its value refers to the record with that value in the RETAIN statement. That is, a key identified with a value of "ABC" in the FREE statement frees the record identified as "ABC" in the RETAIN statement. If LOW-VALUES is used as the value of the key, it refers to the next record after the current record, which is not necessarily the record identified by LOW-VALUES in the RETAIN statement. This is because the current record is changed by an I/O statement and LOW-VALUES always refers to the record following the current record.

The value specified in the KEY phrase must normally be an identifier that specifies a field that agrees with the RECORD KEY defined for the file in size, class, usage, and number of decimal places. However, if the RECORD KEY of the file is USAGE COMPUTATIONAL or INDEX, a positive integer numeric literal of ten or fewer digits can be used as the value in the KEY phrase.

If the KEY phrase is not specified, the record freed is that record defined by the SYMBOLIC KEY of the file. If the SYMBOLIC KEY contains LOW-VALUES, it refers to the next record after the current record, which is not necessarily the record specified by LOW-VALUES in the RETAIN statement. This is because the current record is changed by an I/O statement and LOW-VALUES refers to the record following the current record.

Examples

Sequential File

```
RETAIN HISTORY KEY 0 FOR READ-WRITE UNTIL FREED,
       HISTORY KEY 1 FOR READ-WRITE UNTIL FREED,
       HISTORY KEY 2 FOR READ-WRITE.
READ HISTORY, AT END STOP RUN.
FREE HISTORY EVERY RECORD.
```

Random Access File

```
RETAIN PART KEY 0 FOR ANY VERB.
READ PART, INVALID KEY GO TO ERR.
WRITE PARTREC.
FREE PART KEY 0.
```

Indexed Sequential File

```
MOVE "B" TO SYMBOLIC-KEY.
RETAIN LETTERS FOR READ.
FREE LETTERS.
```

5.9.17 GENERATE

Function

The GENERATE statement causes the Report-Writer to execute all automatic report operations, and, if required, produce one or more report groups.

General Format

GENERATE identifier

Technical Notes

1. If identifier is the name of a TYPE DETAIL report group, the GENERATE statement performs all the automatic report operations, and produces an output detail report group on the output file. This is called detailed reporting.

2. If the identifier is the name of an RD entry, the GENERATE statement performs all the automatic report operations, but does not produce an output detail report group. This is called summary reporting.

3. A GENERATE statement performs the following automatic operations:

   a. Steps and tests the LINE-COUNTER and/or PAGE-COUNTER to produce, if necessary, any PAGE FOOTING and PAGE HEADING report groups.

   b. Recognizes any specified control breaks to produce appropriate CONTROL FOOTING and CONTROL HEADING report groups, and resets appropriate summation counters.

   c. Accumulates into the summation counters all specified identifiers.

   d. Executes any routines defined by a USE statement.

   e. If this is detailed reporting, produces the detailed report group.

4. During the execution of the first GENERATE statement for a report, the following groups, if specified, are produced:

   a. Report Heading

   b. Page Heading

   c. All Control Headings, in the order major to minor.

   d. The detail report group, if this is detailed reporting.

GENERATE (Cont.)

5. Data is moved to the data item in the Report Group Description Entry according to the same rules for movement described for the MOVE statement see Section 5.9.23.

6. A GENERATE statement for a particular report can not be executed until an INITIATE statement has been executed for that report. In addition, if a TERMINATE statement has been executed for that report, a GENERATE statement can not be executed until an intervening INITIATE statement is executed for the report.

5.9.18 GO TO

Function

The GO TO statement causes control to be transferred from one part of the PROCEDURE DIVISION to another.

General Format

Option 1:

```
    GO TO [ procedure-name-1 ]
```

Option 2:

```
    GO TO procedure-name-1,procedure-name-2 [ ,procedure-name-3 ] ...

          DEPENDING ON identifier
```

MR-S-1056-81

Technical Notes

1. Each procedure-name is the name of a paragraph or section in the PROCEDURE DIVISION of the program.

2. Option 1 causes transfer of control to the specified procedure-name, or to some other procedure-name if the GO TO has been previously ALTERed.

   In order to be alterable, Option 1 must appear as the first sentence in a paragraph; only NOTE can follow.

   If procedure-name-1 is not specified, the GO TO must be alterable and an associated ALTER statement must be executed prior to executing this GO TO.

   When this form of GO TO appears in an imperative sentence, it must appear as the last or only statement in the sentence, except for NOTE.

3. Option 2 causes transfer of control to procedure-name-1, procedure-name-2,... or procedure-name-n depending on whether the value of the identifier is 1, 2, .. or n, respectively.

   The identifier must refer to an elementary numeric item having no positions to the right of the decimal point. The item can not be USAGE COMPUTATIONAL-1.

   If the value of the identifier is other than the positive integers 1, 2, ... or n, the GO TO statement is by-passed.

GOBACK

5.9.19 GOBACK

Function

The GOBACK statement is used in a subprogram to return control to the calling program.

General Format

```
GOBACK.
```

Technical Notes

1. The GOBACK statement can only be used in subprograms.
2. When control reaches a GOBACK statement, control is returned to the calling program at the statement immediately following the CALL statement.
3. If a GOBACK statement is encountered in a subprogram that is operating as a main program, it is treated as a STOP RUN statement.
4. Refer to Chapter 11 of this manual for more information on subprograms.

5.9.20 IF

Function

The IF statement causes a conditional expression to be evaluated and the subsequent operations to be performed to be determined as a result of this evaluation.

General Format

$$\underline{\text{IF}}\ \text{conditional-expression}$$

$$\left\{\begin{array}{l}\text{statement-1 [,statement-2]} \ \ldots \\ \underline{\text{NEXT SENTENCE}}\end{array}\right\} \left[\underline{\text{ELSE}} \left\{\begin{array}{l}\text{statement-3 [,statement-4]} \ \ldots \\ \underline{\text{NEXT SENTENCE}}\end{array}\right\}\right] \underline{.}$$

MR-S-1057-81

Technical Notes

1. Conditional expressions are discussed in Section 5.5 in this chapter.

2. The subsequent action of the program is determined by whether the conditional expression is true or false.

   a. If the conditional expression is true and statement-1 and any following statements are given, statement-1 and any following statements are executed and, provided that they do not contain a GO TO or STOP RUN, control passes to the next sentence. If the conditional expression is true and NEXT SENTENCE is given, control passes to the next sentence.

   b. If the conditional expression is false and statement-3 and any following statements are given, statement-3 and any following statements are executed and, provided that they do not contain a GO TO or STOP RUN, control passes to the next sentence.

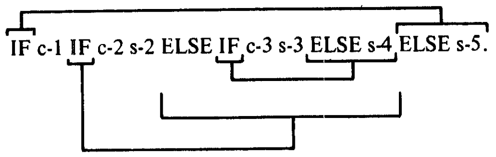
      If the conditional expression is false and either ELSE NEXT SENTENCE is given or the entire ELSE clause is omitted, control passes to the next sentence.

3. The length of compared data-items in the conditional expression of an IF statement is limited to 2047 characters.

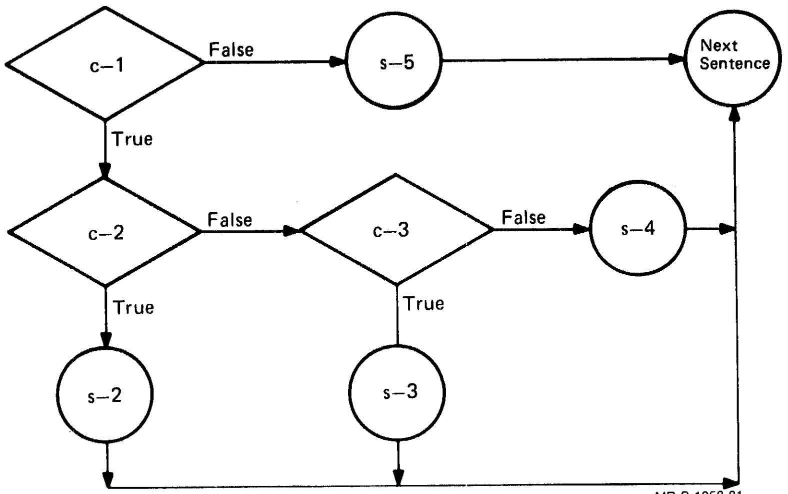
4. Statement-1, statement-2, statement-3, and statement-4 can include any statement or sequence of statements, including other IF statements. IF statements included within other IF statements are nested. Nested IF statements are paired IF and ELSE combinations and can continue up to 12 levels deep. Each ELSE encountered is paired with the nearest preceding IF not already paired with an ELSE. The pairing process begins with the innermost IF ... ELSE pair and proceeds outwards.

IF (Cont.)

Example: (c=condition;s=statement)



```
IF c-1 IF c-2 s-2 ELSE IF c-3 s-3 ELSE s-4 ELSE s-5.
```



MR-S-1058-81

5.9.21 INITIATE

Function

The INITIATE statement is used to initialize all counters before a report is produced.

General Format

<u>INITIATE</u> report-name-1 [, report-name-2] ... <u>.</u>

Technical Notes

1. Each report-name must be defined by an RD entry in the REPORT SECTION of the DATA DIVISION.

2. The INITIATE statement resets all data-name entries that contain SUM clauses associated with a report.

3. The PAGE-COUNTER is set to 1 during the execution of an INITIATE statement. If a different starting value for the PAGE-COUNTER is desired, it can be reset following the INITIATE statement before the execution of the first GENERATE statement.

4. The LINE-COUNTER is set to 0 during execution of the INITIATE statement.

5. The INITIATE statement does not open the file with which the report is associated. An OPEN statement must be executed prior to the execution of the INITIATE statement.

6. A second INITIATE statement for a particular report-name can not be executed until a TERMINATE statement for that report-name is executed.

MERGE

5.9.22 MERGE

Function

The MERGE statement combines two or more identically sequenced files on a set of specified keys. During the MERGE process records are made available, in merged order, to an output procedure or to an output file.

General Format

```
                                               { ASCENDING  }
MERGE [WITH SEQUENCE CHECK] file-name-1 ON     { DESCENDING } KEY data-name-1 [data-name-2] ...

      [    { ASCENDING  }                                 ]
      [ ON { DESCENDING } KEY data-name-3 [data-name-4] ...] ...

   USING file-name-2, file-name-3 [,file-name-4] ...

   {                                        [ { THROUGH }                  ] }
   { OUTPUT PROCEDURE IS section-name-1     [ { THRU    }  section-name-2  ] }
   {                                                                          }
   { GIVING file-name-5                                                       }
```

MR-S-1059-81

Technical Notes

1. File-name-1 must be described in an SD file description entry in the DATA DIVISION. Each data-name must represent data items described in records associated with file-name-1.

2. File-name-2, file-name-3, file-name-4, and file-name-5 must be described in an FD file description, not an SD file description. All records associated with file-name-2, file-name-3, and file-name-4 must be large enough to contain all the KEY data-names.

3. The data-names following the word KEY are listed in order of decreasing significance without regard to how they are divided into KEY clauses.

4. The data-names can be qualified but not subscripted.

5. MERGE statements can appear anywhere in the PROCEDURE DIVISION except in the DECLARATIVES portion or in an INPUT or OUTPUT PROCEDURE associated with a SORT, or an OUTPUT PROCEDURE associated with another MERGE.

6. When the ASCENDING clause is used, the input files must be in sequence from the lowest values to the highest values; when the DESCENDING clause is used, the input files must be in sequence from the highest values to the lowest values.

7. The OUTPUT PROCEDURE, if present, must consist of one or more sections or paragraphs that appear contiguously in the source program and do not form a part of any INPUT PROCEDURE. The OUTPUT PROCEDURE must contain at least one RETURN statement in order to make MERGEd records available for processing.

MERGE (Cont.)

8. ALTER, GO, and PERFORM statements in the OUTPUT PROCEDURE can not refer to procedure-names outside the OUTPUT PROCEDURE in which they appear.

9. If you specify an OUTPUT PROCEDURE, it is PERFORMed by the MERGE statement. You must observe all rules relating to the range of a PERFORM.

10. If WITH SEQUENCE CHECK is present then the input files are checked to make sure that the records are in sequence with respect to the merge keys (that is, that the files were presorted.) A warning message is given for each record out of order.

11. If you specify the GIVING option, all the merged records in file-name-1 are automatically transferred to file-name-5. File-name-5 must not be open when the MERGE statement is executed. Any USE PROCEDURES associated with file-name-5 are executed as appropriate. The GIVING option is equivalent to the following OUTPUT PROCEDURE:

```
     L4.  OPEN OUTPUT file-name-5.
     L5.  RETURN sort-file INTO record-name-5;  AT END GO  TO
          L6.
          WRITE record-name-5.
          GO TO L5.
     L6.  CLOSE file-name-5.
```

MOVE

5.9.23 MOVE

Function

The MOVE statement transfers data in accordance with the rules of editing, from one data area to one or more data areas.

General Format

Option 1:

$$\underline{\text{MOVE}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \underline{\text{TO}} \text{ identifier-2 } [\text{,identifier-3}] \ldots$$

Option 2:

$$\underline{\text{MOVE}} \left\{ \begin{array}{l} \underline{\text{CORRESPONDING}} \\ \underline{\text{CORR}} \end{array} \right\} \text{identifier-1 } \underline{\text{TO}} \text{ identifier-2}$$

MR-S-1060-81

Technical Notes

1. Identifier-1 (or literal-1) represents the data to be moved and is called the sending item. Identifier-2, identifier-3, ... represent the receiving data items.

2. In option 1, the data contained in identifier-1 or literal-1 is moved first to identifier-2, then identifier-3, etc.

   In option 2, data items within the group item associated with identifier-1 are moved to corresponding data items within the group item associated with identifier-2. The results are the same as if you had referred to each pair of corresponding identifiers in separate MOVE statements. The criteria used to determine whether two items are corresponding are described under "The CORRESPONDING Option" at the beginning of this chapter.

3. The following rules apply to both group and elementary items; a group item is treated as a single field.

   a. A numeric edited, alphanumeric edited, or alphabetic data item must not be moved to a numeric or numeric edited data item.

   b. A numeric or numeric edited item must not be moved to an alphabetic data item.

c. A numeric item whose implicit decimal point is not immediately to the right of the least significant digit must not be moved to an alphanumeric or alphanumeric edited item.

All other moves are legal.

4. The following rules apply to legal moves.

a. When an alphanumeric, alphanumeric edited, or alphabetic item is the receiving item,

1. If the size of the sending field is greater than the size of the receiving field, the least significant (rightmost) characters are truncated if the receiving field is not described by a JUSTIFIED RIGHT clause; the most significant (leftmost) characters are truncated if the receiving field is described as JUSTIFIED RIGHT.

2. If the size of the sending field is less than the size of the receiving field, spaces are placed in the remaining rightmost characters of the receiving field if the receiving field is not described by a JUSTIFIED RIGHT clause; spaces are placed in the remaining leftmost characters of the receiving field if the receiving field is described by a JUSTIFIED RIGHT clause.

3. If the sizes of the sending and receiving field are equal, no truncation or filling with spaces takes place.

b. When a numeric or numeric edited item is the receiving item, the sending and receiving fields are aligned by decimal point. If the sending field is not numeric, the decimal point is assumed to be on the right. Any necessary zero filling takes place before editing. If the receiving item has no operational sign, the absolute value of the sending item is stored. If the receiving item has fewer digits to the left or right of the decimal point than does the sending item, the excess digits are truncated. If the sending item contains any nonnumeric characters, the result is unpredictable.

c. Any necessary conversion of data from one form of internal representation to another is performed automatically during the move, along with any editing specified by the PICTURE of the receiving item.

5. Any move that is not an elementary move (that is, both the sending and receiving items are not elementary items) is called a group move. A group move is treated as if it were an alphanumeric to alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In other words, the individual data descriptions of the items within the sending group item and the receiving group item are completely ignored and both items are treated as though they were described by a PICTURE IS X(n) clause, where n is the number of character positions in the particular item.

MULTIPLY

5.9.24 MULTIPLY

Function

The MULTIPLY statement causes one data item to be multiplied by another data item and the resulting product to be stored in a data item.

General Format

Option 1:

```
MULTIPLY {identifier-1} BY identifier-2 [ROUNDED]
         {literal-1   }

    [ON SIZE ERROR statement-1 [,statement-2] ...  .]
```

Option 2:

```
MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3
         {literal-1   }    {literal-2   }

    [ROUNDED] [ON SIZE ERROR statement-1 [,statement-2] ...  .]
                                                        MR-S-1061-81
```

Technical Notes

1. Each MULTIPLY statement must contain at least two operands (a multiplicand and a multiplier). Each identifier must refer to an elementary numeric item, except that identifier-3 can refer to either a numeric or a numeric edited item. Each literal must be a numeric literal; the figurative constants ZERO and TALLY are permitted.

2. Option 1 causes the value of identifier-1 or literal-1 to be multiplied by the value of identifier-2. The resultant product replaces the value of identifier-2.

3. Option 2 causes the value of identifier-1 or literal-1 to be multiplied by the value of identifier-2 or literal-2. The resultant product replaces the value of identifier-3.

MULTIPLY (Cont.)

4. The ROUNDED and SIZE ERROR options are described earlier in this chapter under "Common Options Associated with Arithmetic Verbs".

5. The data item resulting from the multiply operation (that is, the sum of all digits in both operands) must not contain more than 20 decimal digits for the non-BIS compiler and not more than 36 digits for the BIS compiler. In either case, a maximum of 18 digits can be stored in the receiving field.

NOTE

The BIS compiler is standard on the DECSYSTEM-20 and DECsystem-10. For KI based hardware, the non-BIS compiler is optional on the DECsystem-10 (See the COBOL-68 Installation Procedures.)

NOTE

5.9.25 NOTE

Function

The NOTE statement allows the programmer to insert comments in the PROCEDURE DIVISION.

General Format

NOTE character-string.

Technical Notes

1. Any combination of characters from the ASCII character set can be included in the character-string.
2. If the NOTE sentence appears as the first sentence in a paragraph, the entire paragraph is considered to be part of the character-string. The paragraph is ended when a new paragraph is begun. A new paragraph has its first word starting in Area A.
3. If the NOTE statement appears as other than the first sentence in a paragraph, the character-string ends at the first period followed by a space or carriage return.
4. The contents of the character-string appear on the compilation listing, but are not compiled.

5.9.26 OPEN

Function

The OPEN statement initiates the processing of files and, where necessary, performs the checking and writing of labels. It also specifies your covenants for opening a file for simultaneous update.

General Format

```
      ( {INPUT }  file-name-1 [WITH NO REWIND] [,file-name-2 [WITH NO REWIND]] ...
      ( {OUTPUT}
      (
      (                        [     ( READ     ) [     ( READ     ) ]     ]
      ( {I-O         }         [     ( REWRITE  ) [     ( REWRITE  ) ]     ]
      ( {INPUT-OUTPUT} file-name-3 [ FOR < WRITE    > [ AND < WRITE    > ] ... ]
      (                        [     ( DELETE   ) [     ( DELETE   ) ]     ]
      (                        [     ( ANY VERB ) [     ( ANY VERB ) ]     ]
      (
      (                          [                 ( NONE     ) [     ( NONE     ) ]     ]
      (                          [                 ( READ     ) [     ( READ     ) ]     ]
      (                          [ ALLOWING OTHERS < REWRITE  > [ AND < REWRITE  > ] ... ]
      (                          [                 ( WRITE    ) [     ( WRITE    ) ]     ]
      (                          [                 ( DELETE   ) [     ( DELETE   ) ]     ]
      (                          [                 ( ANY VERB ) [     ( ANY VERB ) ]     ]
      (
OPEN <              [             [     ( READ     ) [     ( READ     ) ]     ]
      (             [             [     ( REWRITE  ) [     ( REWRITE  ) ]     ]
      (             [ ,file-name-4 [ FOR < WRITE    > [ AND < WRITE    > ] ... ]
      (             [             [     ( DELETE   ) [     ( DELETE   ) ]     ]
      (             [             [     ( ANY VERB ) [     ( ANY VERB ) ]     ]
      (             [
      (             [              [                 ( NONE     ) [     ( NONE     ) ]     ]
      (             [              [                 ( READ     ) [     ( READ     ) ]     ]
      (             [              [ ALLOWING OTHERS < REWRITE  > [ AND < REWRITE  > ] ... ] ... ] ...
      (             [              [                 ( WRITE    ) [     ( WRITE    ) ]     ]
      (             [              [                 ( DELETE   ) [     ( DELETE   ) ]     ]
      (             [              [                 ( ANY VERB ) [     ( ANY VERB ) ]     ]
      (
      ( [EXTEND] file-name-5 [file-name-6] ...
      (
      ( [UNAVAILABLE statement-1 [,statement-2] ...] .
```

MR-S-1062-81

OPEN (Cont.)

Technical Notes

1. The OPEN statement must be executed for a file prior to the execution of any SEEK, READ, WRITE, REWRITE, DELETE, or CLOSE for that file.

2. A second OPEN statement for a file cannot be executed prior to the execution of a CLOSE statement for that file.

3. When your program executes an OPEN verb, the record area for that file is cleared.

4. An OPEN statement does not obtain or release the first record of a file. A READ statement must be executed to obtain the first record (or a WRITE statement must be executed to release the first record).

5. The maximum number of files that can be opened at a time is 16. When indexed sequential files are being used, each indexed sequential file is treated as two files: the index file and the data file. If the program is segmented, one less file can be open; similarly, if the RERUN option is being used, one less file can be open. The key word INPUT, OUTPUT, INPUT-OUTPUT, or I-O applies to each subsequent filename until another such key word is encountered or until the end of the OPEN statement is reached.

6. When you OPEN an indexed sequential file, the OPEN statement initializes the keys to LOW-VALUES. Thus, you cannot load a key with a value prior to opening the ISAM file and expect the key to have the value you specify.

7. The NO REWIND option has meaning only for magtape files and is ignored for all other devices. If the NO REWIND clause is not specified for a tape file, the tape is rewound to the beginning of tape.

8. If labels exist, the label is read into the record area to make it available to the USE routines. The record area is then filled with spaces. If a file has been described as LABEL RECORDS ARE STANDARD, standard label checking or label writing is performed; your BEGINNING LABEL (USE) routines are executed if provided. If a file has been described as LABEL RECORDS ARE data-name-1, your BEGINNING LABEL (USE) routines are executed. If a file has been described as LABEL RECORDS ARE OMITTED, no label checking or writing is performed.

9. If an INPUT file is described as OPTIONAL (in the FILE-CONTROL paragraph), the object-time system types the message

   IS file-name PRESENT?

   and wait for you to type "YES" or "NO". If you type "NO", the first READ statement for this file causes the imperative-statement at the AT END or INVALID KEY clause to be executed.

OPEN (Cont.)

10. The I-O or INPUT-OUTPUT options permit the opening of a file on a random-access device for both input and output processing. When the I-O option is specified, the execution of the OPEN statement causes the standard beginning label procedures and your BEGINNING LABEL routines, if specified by a USE statement, to be executed. If the file does not exist when it is opened for INPUT-OUTPUT, an empty file is created.

11. A file is opened for simultaneous update if the ALLOWING OTHERS clause is present in the OPEN statement. It must be opened in I-O mode and cannot have a recording mode of V (variable-length EBCDIC).

12. If the first user of a file opens it for simultaneous update, all subsequent users of the file must also open it for simultaneous update or for input only. If the file is currently open for simultaneous update, any subsequent users attempting to open the file for output or I-O are denied access to the file. If the first user of a file opens it for output or I-O only and subsequent users attempt to open that file for simultaneous update, the simultaneous update users are denied access to the file until the first user closes it.

13. After the keyword FOR, you can give one or more verbs that you intend to execute while you have the file open. You can only execute those verbs that you have specified. Following the keywords ALLOWING OTHERS, you give one or more verbs that you allow other users to execute when they open the file. You can also specify that others not be allowed to execute any verbs when they open the file. Specification of ANY VERB means that all verbs legal for the file are permissible. If the ALLOWING OTHERS clause is not present, the file is not opened for simultaneous update.

14. Once you have opened at least one file for simultaneous update, you cannot open any other files for simultaneous update until all files you previously opened for simultaneous update are closed. Thus, all files that must be open concurrently for simultaneous update must be opened in the same OPEN statement. However, files that are not to be opened for simultaneous update can be opened at any time.

15. Files can be opened for INPUT, OUPUT, and just INPUT-OUTPUT (that is, not for simultaneous update) in the same OPEN statement as files opened for simultaneous update.

OPEN (Cont.)

16. When more than one file is to be opened in one OPEN statement and at least one of the files is to be opened for simultaneous update, no files are opened if the simultaneous update file cannot be opened. Simultaneous update files cannot be opened if they are not available in the modes specified by both the FOR and ALLOWING clauses. If the files cannot be opened for this reason, your program is suspended until all files are available, unless the UNAVAILABLE clause is specified. If the UNAVAILABLE clause is specified and one or more simultaneous update files are unavailable, control passes to the UNAVAILABLE clause. Note that the availability of the simultaneous update files is always checked before any files are opened. After the simultaneous update files are checked for availability the files are opened. A failure during the actual opening process on any of the files does not cause the UNAVAILABLE path to be taken, but an error to be returned. You can choose to ignore the error by using the FILE STATUS clause in the ENVIRONMENT DIVISION (see FILE STATUS in Chapter 3 of this manual).

17. Any valid COBOL statements (including OPEN) can be used in the UNAVAILABLE clause.

18. If you wish to open a file in your program for simultaneous update and the file is not available to it, the open request is queued for the file on a first-come/first-served basis. However, if your program wishes to open more than one file for simultaneous update and at least one of the files is not available, the program is queued for those files that are available as well as the ones that are not available. This is because the program cannot open one file without opening all files in the same OPEN request. The requests for files remain in the queue for the files until all of the files are available to you.

19. If your program violates its simultaneous update covenants, it is aborted. That is, if the program opens a file for READ and then issues a WRITE statement for that file, the program is aborted.

20. Once a file is open for simultaneous update, you must issue a RETAIN statement before you execute any I/O on that file. Refer to the RETAIN statement, described further ahead in this chapter.

21. When you specify the EXTEND phrase, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file add records to the file as though the file had been opened with the OUTPUT phrase.

22. If you wish to open a file using the EXTEND option, you cannot open the same file for input-output in the same program. You can, however, supply two FD's for the same file. This allows you to open the file for input-output using the file name supplied with one FD, and at a different time you can open the file using the EXTEND option and the file name supplied in the other FD.

OPEN (Cont.)

23. You cannot use the EXTEND option with system-labeled tapes.

24. When you specify the EXTEND phrase and the LABEL RECORDS clause indicates label records are present, the execution of the OPEN statement includes the following steps:

    a. The beginning file labels are processed only in the case of a single reel file.

    b. The beginning reel labels on the last existing reel are processed as though the file was being opened with the INPUT phrase.

    c. The existing ending file labels are processed as though the file is being opened with the INPUT phrase. These labels are then deleted.

    d. Processing then proceeds as though the file had been opened with the OUTPUT phrase.

Examples

```
OPEN INPUT INFIL.

OPEN I-O TRANSACTION FOR READ AND WRITE,
     ALLOWING OTHERS READ AND WRITE.

OPEN OUTPUT LOG, LIST,
     INPUT-OUTPUT MASTER FOR READ AND REWRITE,
                     OTHERS ANY
                 DET FOR READ,
                     OTHERS READ AND WRITE,
                 ACCOUNT FOR ANY
                     OTHERS NONE,
     INPUT DAILY WITH NO REWIND,
     I-O SKILLS
                 NAMES FOR WRITE.
```

PERFORM

5.9.27 PERFORM

Function

The PERFORM statement is used to depart from the normal sequence of execution to execute one or more procedures and then return control to the normal sequence.

General Format

Option 1:

$\underline{\text{PERFORM}}$ procedure-name-1 $[\underline{\text{THRU}}$ procedure-name-2$]$

Option 2:

$\underline{\text{PERFORM}}$ procedure-name-1 $[\underline{\text{THRU}}$ procedure-name-2$]$ $\left\{\begin{matrix}\text{identifier-1}\\\text{integer-1}\end{matrix}\right\}$ $\underline{\text{TIMES}}$

Option 3:

$\underline{\text{PERFORM}}$ procedure-name-1 $[\underline{\text{THRU}}$ procedure-name-2$]$ $\underline{\text{UNTIL}}$ condition-1

Option 4:

$\underline{\text{PERFORM}}$ procedure-name-1 $[\underline{\text{THRU}}$ procedure-name-2$]$

$\underline{\text{VARYING}}$ identifier-1 $\underline{\text{FROM}}$ $\left\{\begin{matrix}\text{literal-1}\\\text{identifier-2}\end{matrix}\right\}$

$\underline{\text{BY}}$ $\left\{\begin{matrix}\text{literal-2}\\\text{identifier-3}\end{matrix}\right\}$ $\underline{\text{UNTIL}}$ condition-1

MR-S-1063-81

PERFORM (Cont.)

```
[ AFTER VARYING identifier-4 FROM { literal-3    }
                                  { identifier-5 }

    BY { literal-4    } UNTIL condition-2
       { identifier-6 }

  [ AFTER VARYING identifier-7 FROM { literal-5    }
                                    { identifier-8 }

    BY { literal-6    } UNTIL condition-3 ]]
       { identifier-9 }
                                          MR-S-1064-81
```

Technical Notes

1. Each procedure-name is the name of a section or paragraph in the PROCEDURE DIVISION. Each identifier must refer to a numeric elementary item described in the DATA DIVISION. Each literal must be a numeric literal or the figurative constants ZERO and TALLY.

2. When the PERFORM statement is executed, control is transferred to the first statement of procedure-name-1. An automatic return to the statement following the PERFORM statement is established as follows. The procedures executed constitute the range of the PERFORM.

   a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, the return is after the last statement of procedure-name-1.

   b. If procedure-name-1 is a section-name and procedure-name-2 is not specified, the return is after the last statement in the last paragraph in procedure-name-1.

   c. If procedure-name-2 is a paragraph-name, the return is after the last statement in that paragraph.

   d. If procedure-name-2 is a section-name, the return is after the last statement in the last paragraph of that section.

3. There is no relationship between procedure-name-1 and procedure-name-2, except that the sequence of operations beginning at procedure-name-1 must eventually end with the execution of procedure-name-2 in order to effect the return at the end of procedure-name-2. Any number of GO TO and/or PERFORM statements can occur between procedure-name-1 and procedure-name-2.

PERFORM (Cont.)

4. If control passes to these procedures by means other than a PERFORM statement, control passes through the return point to the following statement as though no return mechanism were present.

5. No PERFORM statement can terminate until all PERFORM statements that it has executed have terminated. A PERFORM statement can be executed which terminates at the same procedure-name as another active PERFORM.

6. Option 1 causes the PERFORM range to be executed once, followed by a return to the statement immediately following the PERFORM.

7. Option 2 causes the PERFORM range to be executed the number of times specified by identifier-1 or integer-1. The value of identifier-1 or integer-1 must not be negative; it can be zero. Once the PERFORM statement has been initialized, any modification to the contents of identifier-1 has no effect on the number of times the range is executed.

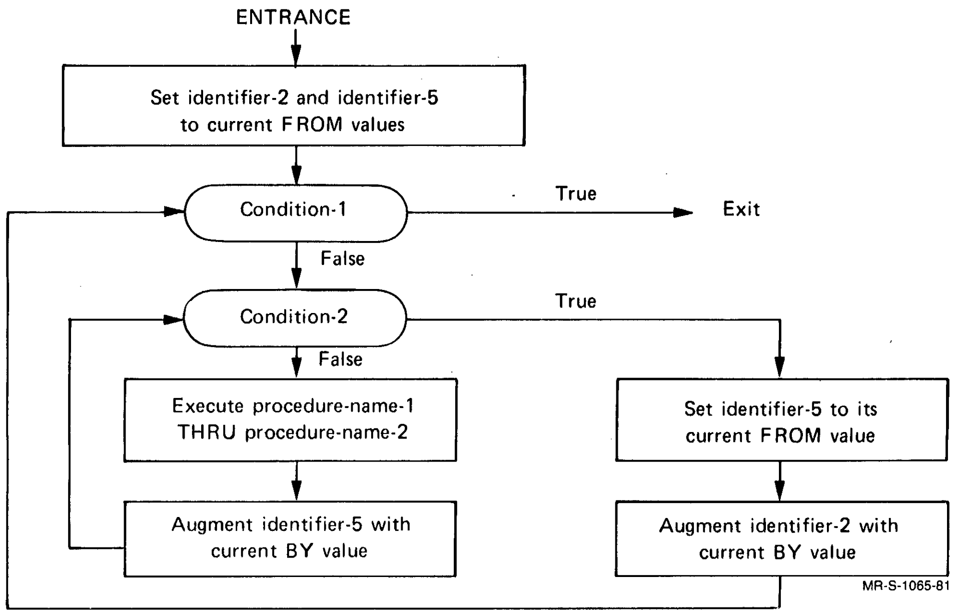
8. Option 3 causes the PERFORM range to be executed until the condition specified in the UNTIL clause is true. If this condition is true at the time the PERFORM statement is initialized, the range is not executed. Conditions are explained under "Conditional Expressions" earlier in this chapter.

9. Option 4 is used to augment the value of one or more identifiers during the execution of a PERFORM statement.

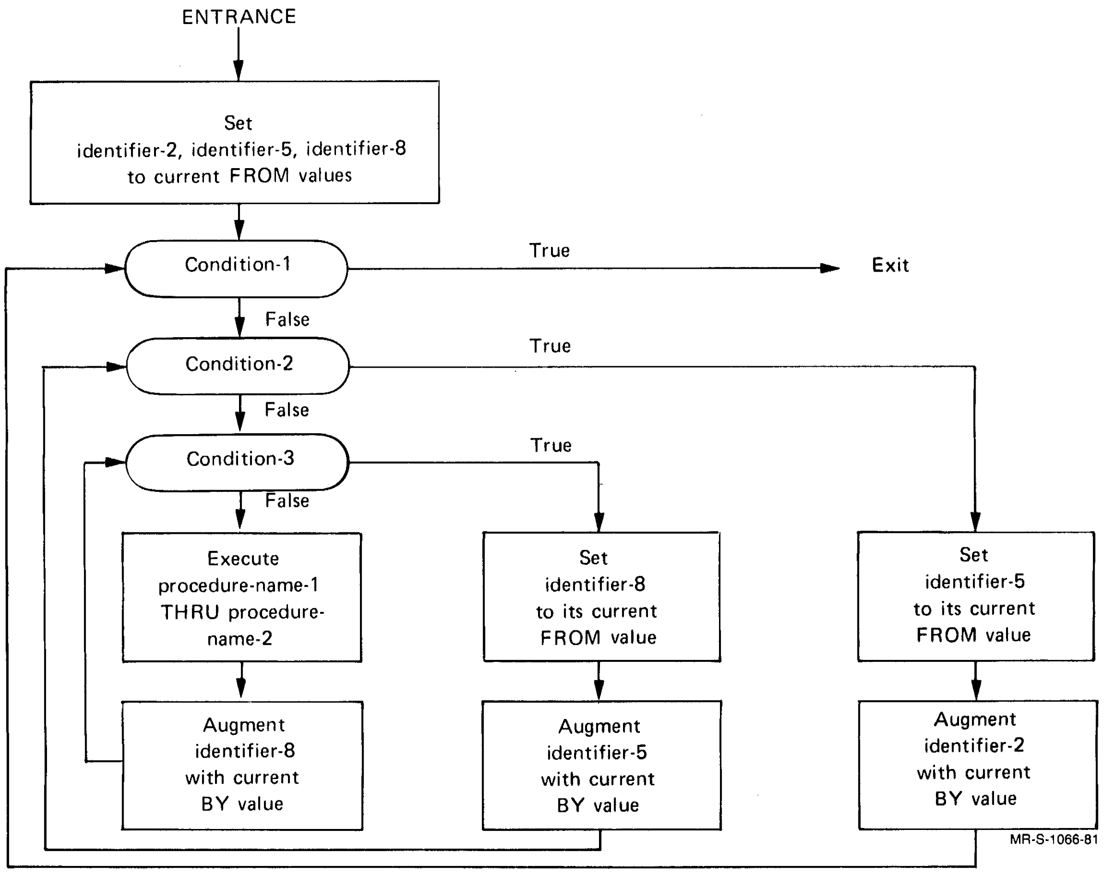
   In option 4, when only one identifier is varied, identifier-1 is set equal to identifier-2 or literal-2 when the PERFORM statement is initialized. If the condition specified is determined to be false at this point, the PERFORM range is executed once. Then the value of identifier-1 is augmented by identifier-3 or literal-3 and the rest of the condition is done again. This cycle continues until condition-1 is true; at this point, control passes to the statement following the PERFORM statement. If condition-1 is true at the beginning of the execution of the PERFORM, control immediately passes to the statement following the PERFORM.

PERFORM (Cont.)

The flow chart below illustrates the logic of the PERFORM cycle when two identifiers are varied.


ENTRANCE
Set identifier-2 and identifier-5 to current FROM values
Condition-1
True
Exit
False
Condition-2
True
False
Execute procedure-name-1 THRU procedure-name-2
Set identifier-5 to its current FROM value
Augment identifier-5 with current BY value
Augment identifier-2 with current BY value
MR-S-1065-81

The following flow chart illustrates the logic of the PERFORM cycle when three identifiers are varied.


ENTRANCE
Set identifier-2, identifier-5, identifier-8 to current FROM values
Condition-1
True
Exit
False
Condition-2
True
False
Condition-3
True
False
Execute procedure-name-1 THRU procedure-name-2
Set identifier-8 to its current FROM value
Set identifier-5 to its current FROM value
Augment identifier-8 with current BY value
Augment identifier-5 with current BY value
Augment identifier-2 with current BY value
MR-S-1066-81

PERFORM (Cont.)

10. When a procedure-name in a segment with a priority number greater than 49 is referred to by a PERFORM statement contained in a segment with a different priority number, the segment referred to is made available in its initial state (that is, with all ALTERable GO TOs set to their initial setting) for each execution of the PERFORM statement.

11. A PERFORM statement in a section not in the DECLARATIVES can have as its range, procedures wholly contained within the DECLARATIVES; however, a PERFORM statement in a section within the DECLARATIVES can not have any non-DECLARATIVE procedures within its range.

12. A PERFORM statement within an INPUT or OUTPUT PROCEDURE associated with a SORT or MERGE verb can not have within its range any procedures outside of that INPUT or OUTPUT procedure.

5.9.28 READ

Function

The READ statement makes available a logical record from an input file and allows performance of a specified imperative statement when end-of-file or invalid key is detected.

General Format

```
READ file-name RECORD

   [INTO identifier] {AT END     } statement-1 [,statement-2] ... .
                     {INVALID KEY}
                                                         MR-S-1067-81
```

Technical Notes

1. An OPEN INPUT or OPEN I-O statement must be executed for the file prior to execution of the first READ statement for that file.

2. The AT END clause is valid only for those files whose access mode is SEQUENTIAL (explicitly or implicitly).

   The AT END clause should be used if there is the possibility that your program will encounter "End-of-Data", in order to avoid program failure.

   The INVALID KEY clause is valid only for those files whose access mode is RANDOM or INDEXED.

   If an end-of-file condition is encountered during the execution of a READ statement for a sequential file, the statement(s) specified in the AT END clause is executed, and no logical record is made available.

   The logical end-of-file depends upon the type of device on which the file resides (see the Monitor Calls manual).

   After execution of the imperative-statement(s) in the AT END clause, no further READ statements can be executed for that file without first executing a CLOSE statement followed by an OPEN statement for the file.

   If, during the execution of a READ statement for a file whose access mode is RANDOM, the ACTUAL KEY is found to contain a value not within the range specified by the FILE-LIMITS clause for that file or a value for a record that has not been written (that is, a zero-length record), the statement(s) specified in the INVALID KEY clause is executed and no logical record is made available.

READ (Cont.)

When a READ statement is executed for a file whose access mode is RANDOM and the ACTUAL KEY contains a value of 0, the first nonzero-length record having a key higher than the last record processed (by a READ or WRITE statement) is made available. If no such record exists (that is, end-of-file), the INVALID KEY statement(s) is executed and no record is made available. If the file has been opened but no READ or WRITE statement has been executed, the first nonzero-length record is made available. You can use this method to sequentially read a file whose access mode is RANDOM.

When a READ statement is executed for a file whose access mode is INDEXED and the SYMBOLIC KEY contains a value other than LOW-VALUES, a search of the file is made to find the record that has a key equal to the contents of the SYMBOLIC KEY associated with the file. If that record is found, it is moved to the record area for the file; if it is not found, the statement(s) associated with the INVALID KEY clause is executed, and no record is made available. When a READ statement is executed for a file whose access mode is INDEXED and the SYMBOLIC KEY contains a value of LOW-VALUES, the first logical record having a key higher than the last record processed (by a READ, WRITE, REWRITE, or DELETE statement) is made available. The next higher key is used regardless of whether or not the previous I/O operation caused the INVALID KEY path to be taken. If no such record exists (i.e., end-of-file), the INVALID KEY statement(s) is executed, and no record is made available. If the file has been opened but no READ, WRITE, REWRITE, or DELETE statement has been executed, the first record of the file is made available.

3. If a file described by an OPTIONAL clause is not present, the imperative-statement(s) in the AT END or INVALID KEY clause is executed on the first READ for that file. Any specified USE procedures are not performed.

4. If logical end-of-reel is recognized during execution of a READ statement, the following operations are carried out.

   a. The reel is rewound and your ENDING REEL LABEL PROCEDUREs are executed, if specified in a USE statement.

   b. If the file is assigned to more than one device, the devices are advanced. The previous reel is rewound and the next reel is initialized.

   c. The standard beginning label procedure and your BEGINNING REEL LABEL PROCEDURE are executed, if specified in a USE statement.

   d. The first data record on the new reel is made available.

5. If a file consists of more than one type of logical record, these records automatically share the same storage area. This is equivalent to an implied REDEFINE for the record area. Only information in the current record is accessible.

6. If the INTO identifier option is specified, the READ statement is then equivalent to a READ without the INTO option, followed by a MOVE of the record area associated with the filename to identifier.

5.9.29 RELEASE

Function

The RELEASE statement transfers records to the initial phase of the sort operation.

General Format

RELEASE record-name [FROM identifier]

Technical Notes

1. A RELEASE statement can be used only in an input procedure associated with a SORT or MERGE statement for a file whose SD description contains record-name.

2. If the FROM option is used, the contents of identifier are moved to record-name, then the contents of record-name are released to the sort subroutines.

3. After the RELEASE statement is executed, the contents of record-name are not available.

4. Refer to the description of the SORT or MERGE verb for examples.

RETAIN

5.9.30 RETAIN

Function

The RETAIN statement specifies your intent to access one or more records in files that are open for simultaneous update.

General Format

```
RETAIN file-name-1 RECORD [ KEY { identifier-1 } ]
                                { literal-1    }

     { READ          [     { READ          }     ] }
     { REWRITE       [     { REWRITE       }     ] }
     { READ-REWRITE  [     { READ-REWRITE  }     ] }
 FOR { DELETE        [ AND { DELETE        } ... ] } [ UNTIL FREED ]
     { WRITE         [     { WRITE         }     ] }
     { READ-WRITE    [     { READ-WRITE    }     ] }
     { ANY VERB      [     { ANY VERB      }     ] }

[
[ ,file-name-2 RECORD [ KEY { identifier-2 } ]
[                           { literal-2    }
[
     { READ          [     { READ          }     ] }               ]
     { REWRITE       [     { REWRITE       }     ] }               ]
     { READ-REWRITE  [     { READ-REWRITE  }     ] }               ]
 FOR { DELETE        [ AND { DELETE        } ... ] } [ UNTIL FREED ] ]
     { WRITE         [     { WRITE         }     ] }               ]
     { READ-WRITE    [     { READ-WRITE    }     ] }               ]
     { ANY VERB      [     { ANY VERB      }     ] }               ]

[ UNAVAILABLE statement-1 [,statement-2] ... ] .
                                              MR-S-1068-81
```

Technical Notes

1. Filename-1, filename-2... must be the names of files previously opened for simultaneous update.

RETAIN (Cont.)

2. Identifier-1, identifier-2... and literal-1, literal-2... specify keys that refer to records in the file.

3. Statement-1, statement-2... are any valid COBOL statements.

4. The RETAIN statement must be given before any record is accessed in a file opened for simultaneous update. If it is given for a file not open for simultaneous update, the program is terminated.

5. The RETAIN statement does not cause any change in the record area or any change in the positioning in the file. You must explicitly issue I/O statements for these changes to be performed. Thus, the RETAIN statement does not cause an end-of-file condition.

6. The actions performed by any I/O operation is logically the same as if the file were not opened for simultaneous update. That is, a sequential file is always read/written sequentially; the ACTUAL KEY is examined to determine the record to be read/written in a random access file; and the SYMBOLIC KEY is examined to determine the record to be read/written/rewritten/deleted in an indexed sequential file. The only difference is that a check is made to ascertain that the record has been retained. Thus, retaining a record does not cause that record to become the current record of the file. Only I/O operations can cause a record to become the current record of the file.

7. You can retain nonexistent records in a file, but you will receive an error if you attempt to perform I/O, other than a WRITE, on these nonexistent records. You can perform a WRITE operation on nonexistent records.

8. It is possible to mix requests for records from sequential, random, and indexed sequential files in the same RETAIN statement.

9. Using the RETAIN for WRITE, DELETE, or ANY VERB statement with indexed-sequential files locks the entire file, not just the record.

10. When you retain a record for READ, other users are also allowed to read that record, but cannot perform any other form of I/O on that record (WRITE, REWRITE, or DELETE). In addition, any other record in that logical-block, where a logical-block is that set of records grouped in a block as defined by the BLOCK CONTAINS clause or as calculated by the compiler for sequential access files, cannot be accessed by other users for any form of I/O. When you retain a record for any use other than READ, all other users are banned completely from accessing that record or block of records.

11. The statement included in the FOR clause in the RETAIN statement must agree with at least one statement in the FOR clause in the OPEN statement for the file. If ANY VERB is specified in the FOR clause in the RETAIN statement, the file must have been explicitly opened for ANY VERB.

RETAIN (Cont.)

12. The record or records named in the RETAIN statement are automatically freed upon execution of the statement or statements (except ANY VERB) in the FOR clause of the RETAIN statement. If you do not issue an I/O statement for the record, or if the UNTIL FREED phrase is used, you must explicitly free the record with the FREE statement. If a record is not freed, you cannot retain any more records in any of your files open for simultaneous update.

13. The UNTIL FREED phrase allows you to retain several logically related records for processing without their being freed automatically by the I/O statements. Instead, the records are retained until they are explicitly freed by means of the FREE statement.

14. The KEY phrase allows you to specify a particular record or to specify more than one record in a file.

15. All records to be retained concurrently, whether in one or several files, must be retained in the same RETAIN statement. Once records in any file have been retained, no other records in any open file can be retained until the currently retained records have all been freed. This rule prevents a deadly embrace situation.

NOTE

Deadly embrace occurs when two users make conflicting demands upon a file resource and neither is willing or able to yield to the other, with the result that both programs hang or stall waiting for the resource to become available.

16. When attempting to retain records, the program is suspended if any one of the records is not available. If you wish the program to perform other processing, rather than be suspended, you can include an UNAVAILABLE phrase in the RETAIN statement. Any valid COBOL statement can be used in the UNAVAILABLE phrase.

17. Use of the RETAIN statement differs according to the access mode of the file. Each type of file is described separately below.

18. SEQUENTIAL ACCESS FILES

    a. Records in a sequential access file only can be retained for READ, WRITE, READ-WRITE, or ANY VERB. For sequential access files, ANY VERB means READ, WRITE, and READ-WRITE.

b. When the KEY phrase is specified, KEY 0 refers to the next record in the file. The next record in the file depends on the last I/O operation performed (READ or WRITE) and the I/O operation for which the record is to be retained. If the last record was written, the next record to be retained for READ, WRITE, or READ-WRITE is defined to be the one following the record just written. Similarly, if the last record was read, the next record to be retained for READ is defined to be the one following the one just read. However, the next record to be retained for WRITE is defined to be the record just read.

c. Subsequent KEY values (1, 2, 3...), refer to records relative to the record designated by a KEY value of 0.

d. If the KEY phrase is not included, the record retained is always the record designated by a KEY value of 0.

e. The value of a key can be specified by any identifier, which can be subscripted or qualified, or both, provided that its USAGE is COMPUTATIONAL or INDEX. The value of the key can also be specified by a positive integer numeric literal containing ten or fewer digits.

f. It is recommended that you, when performing simultaneous updating on sequential access files, retain several records at a time so that the input/output overhead is reduced. If records are retained singly, each record must be brought into memory from the device (even if it is already in memory) so that you have the latest copy of the record. Also when you free a record (either implicitly or explicitly), after writing it, the record must be written out to the device so that other users have access to the latest copy of that record.

g. Example:

```
OPEN INPUT-OUTPUT HISTORY FOR READ AND WRITE
    ALLOWING OTHERS READ AND WRITE
                .
                .
                .
RETAIN HISTORY KEY 0 FOR READ-WRITE UNTIL FREED,
    HISTORY KEY 1 FOR READ-WRITE UNTIL FREED,
    HISTORY KEY 2 FOR READ-WRITE;
READ HISTORY, AT END STOP RUN.
                .
                .
                .
```

19. RANDOM ACCESS FILES

a. Records in a random access file can only be retained for READ, WRITE, READ-WRITE, or ANY VERB. For random access files, ANY VERB means READ, WRITE, and READ-WRITE.

RETAIN (Cont.)

b. When the KEY phrase is specified, the value of the key designates a specific record in the file, just as the ACTUAL KEY of the file does. Thus, record 1 is always the first record in the file. If the value of the key is 0, however, the record retained is the next sequential record in the file. The next record in the file depends on the last I/O operation performed (READ or WRITE) and the I/O operation for which the record is to be retained. If the last record was written, the next record to be retained for READ, WRITE, or READ-WRITE is defined to be the one following the record just written. Similarly, if the last record was read, the next record to be retained for READ is defined to be the one following the record just read. However, the next record to be retained for WRITE is defined to be the record just read. Note that the next record actually read or written depends on the value of the ACTUAL KEY, not on the record specified in the RETAIN statement.

c. If you wish to read/write the file sequentially, you should set the KEY to 0 in the RETAIN statement and set the ACTUAL KEY to 0 so that you are performing I/O on the same records that you are retaining. If you wish to read/write the file randomly, you should set the ACTUAL KEY to the desired record and either use the same value in the KEY in the RETAIN statement or use no KEY value in the RETAIN statement.

d. If the KEY phrase is not specified, the value used for the key is taken from the ACTUAL KEY specified for the file.

e. The value of a key can be specified by any identifier, which can be subscripted or qualified, or both, provided that its USAGE is COMPUTATIONAL or INDEX. The value of the key can also be specified by a positive integer numeric literal containing ten or fewer digits.

f. Example:

```
OPEN I-O PART FOR READ AND WRITE ALLOWING OTHERS
NONE.
MOVE 64 TO PART-ACTUAL-KEY
RETAIN PART FOR READ.
READ PART, INVALID KEY GO TO ERR.
                        .
                        .
                        .
RETAIN PART KEY 0 FOR WRITE,
      PART KEY 35 FOR READ AND WRITE.
WRITE PARTREC.
MOVE 35 TO PART-ACTUAL-KEY.
READ PART, INVALID KEY GO TO ERR.
WRITE PARTREC.
```

RETAIN (Cont.)

20. INDEXED SEQUENTIAL ACCESS FILES

a. Records in an indexed sequential file can be retained for READ, WRITE, REWRITE, DELETE, READ-REWRITE, and ANY VERB. For indexed sequential files, ANY VERB means READ, WRITE, REWRITE, DELETE, and READ-REWRITE. Records in an indexed sequential file cannot be retained for READ-WRITE.

b. When the KEY phrase is specified, the value of the key refers to a specific record in the file, just as the SYMBOLIC KEY does.

c. The value specified in the KEY phrase must normally be an identifier that specifies a field that agrees with the RECORD KEY defined for the file in size, class, usage, and number of decimal places. However, if the RECORD KEY of the file is USAGE COMPUTATIONAL or INDEX, a positive numeric literal of ten or fewer digits can be used as the value in the KEY phrase.

d. If the KEY phrase is not specified, the value used for the key is taken from the current SYMBOLIC KEY for the file.

e. If the value of the key is LOW-VALUES (which must be specified as the contents of an identifier or as the SYMBOLIC KEY), the record retained is that following the last record referenced in the same RETAIN statement or by a READ, WRITE, REWRITE, or DELETE statement.

f. If other users are allowed to write or delete records in an indexed sequential file, LOW-VALUES cannot be used as the value of the first key specified in the RETAIN statement. Similarly, the first I/O statement following the RETAIN statement cannot reference a record specified with a SYMBOLIC KEY value of LOW-VALUES. These restrictions are applied because the next record in the file could be undefined because another user has written or deleted that record.

g. Example:

```
OPEN I-O LETTERS FOR READ ALLOWING OTHERS READ AND
WRITE.
MOVE "B" TO SYMBOLIC KEY.
RETAIN LETTERS FOR READ.
READ LETTERS INVALID KEY GO TO ERRS.
```

RETURN

5.9.31 RETURN

Function

The RETURN statement obtains sorted records from the final phase of a SORT or MERGE operation.

General Format

```
RETURN file-name RECORD [INTO identifier] AT END statement-1 [,statement-2] ... .
```

MR-S-1069-81

Technical Notes

1. File-name must be described by an SD file descriptor.

2. A RETURN statement can be used only in an output procedure associated with a SORT or MERGE statement for file-name.

3. If the INTO phrase is specified, the current record is moved from the record area associated with file-name to identifier.

4. The AT END path is automatically taken when there are no more records to be returned. After executing the statement(s) in the AT END clause, no RETURN statements can be executed until another SORT or MERGE is executed.

5. Refer to the description of the SORT or MERGE verbs for examples.

5.9.32 REWRITE

Function

The REWRITE statement replaces an already existing record in a file whose access mode is INDEXED.

General Format

```
REWRITE record-name [FROM identifier] INVALID KEY statement-1 [,statement-2] ... .
```

MR-S-1070-81

Technical Notes

1. Record-name must be a record associated with a file whose access mode is INDEXED.

2. When the REWRITE statement is executed, a record in the file is located whose key value is equal to the contents of the SYMBOLIC KEY associated with the file. The contents of the SYMBOLIC KEY item are moved to the RECORD KEY item and the contents of the record are then replaced with the contents of record-name. If no such record exists in the file, the statement(s) associated with the INVALID KEY clause is executed.

3. At the time the REWRITE statement is executed, the file must be open for OUTPUT or INPUT-OUTPUT.

4. If the FROM option is used, the statement is equivalent to:

```
MOVE identifier TO record-name
REWRITE record-name (without the FROM option)
```

SEARCH

5.9.33 SEARCH

Function

The SEARCH statement is used to search a table until a specified condition exists.

General Format

Option 1:

```
SEARCH identifier-1 [VARYING identifier-2][AT END imperative-statement-1 [,imperative-statement-2] ...]

    WHEN condition-1 { imperative-statement-3 [,imperative-statement-4]  ...}
                     { NEXT SENTENCE                                        }

    [,WHEN condition-2 { imperative-statement-5 [,imperative-statement-6]  ...}] ...  .
                       { NEXT SENTENCE                                        }
```

Option 2:

```
SEARCH ALL identifier-1 [AT END imperative-statement-1 [,imperative-statement-2] ...]

    WHEN condition-1 { imperative-statement-3 [,imperative-statement-4]   ...}  .
                     { NEXT SENTENCE                                         }
                                                                        MR-S-1071-81
```

Technical Notes

1. If any of the optional clauses are present, they must appear in the order shown.

2. Identifier-1 must not be subscripted or indexed, but its description must contain an OCCURS clause with an INDEXED BY option. In option 2, identifier-1 must also contain a KEY option in its OCCURS clause.

3. Identifier-2 must be an index, or an elementary numeric item with no places to the right of the decimal point.

4. In option 1, condition-1, condition-2, etc., can be any condition described in Section 5.5.

SEARCH (Cont.)

5. In option 2, condition-1 must consist of a relation condition incorporating the EQUAL TO or equal sign, or a condition-name condition where the VALUE clause contains only a single literal, or a compound condition consisting of two or more such simple conditions connected by AND.

   A data-name that appears in the KEY clause of identifier-1 must appear as the subject or object of a test, or be the name of the data-item with which the tested condition-name is associated. However, all preceding data-names in the KEY clause must also be included within condition-1.

6. If the AT END clause is not present, AT END NEXT SENTENCE is assumed.

7. If the VARYING option is not specified, the first index specified in the INDEXED BY option for identifier-1 is used.

   If the VARYING option is used, and identifier-2 is the name of an item specified in the INDEXED BY option for identifier-1, then identifier-2 is used as the index. If identifier-2 is not specified in the INDEXED BY option for identifier-1, the first index-name in the INDEXED BY option is used as the index, and identifier-2 contains the value of the index at each step of the search.

8. If option 1 of the SEARCH verb is used, a serial search takes place, starting with the current index setting.

   If, at the start of execution of the SEARCH statement, the index contains a value that is not positive or is greater than allowed (greater than the number of occurrences or greater than any DEPENDING item), the statement(s) specified in the AT END statement is executed.

   If, at the start of execution of the SEARCH statement, the index is within the allowed range of values, the WHEN conditions are evaluated one at a time. If any condition is true, the associated statement(s) is executed. If no condition is true, the index is incremented by 1, and the search operation is executed again.

   The contents of the index are always left as they were when the search is terminated, either by a WHEN condition, or the AT END condition.

9. If option 2 of the SEARCH verb is used, a binary search takes place. All the keys in the table must be in order (ascending or descending) and all the elements in the table must be filled. It is up to you to ensure that the keys associated with the table are in order and the table filled. If the keys are not in order, or if there are empty elements in the table being searched, the SEARCH can take the AT END path even if the key being searched for is there. If the table is not going to be filled, using the DEPENDING ON clause with OCCURS effectively shortens the table.

SEARCH (Cont.)

The initial contents of the index are ignored; instead, the table is examined until the WHEN condition is satisfied (in which case imperative-statement-3 and any following statements are executed) or until the entire table is examined (in which case the AT END statement(s) is executed).

When the search is terminated, the contents of the index reflect the occurrence number of the entry that satisfied the WHEN condition if it was satisfied, or is arbitrary if it was not satisfied.

Conditional statements (for example, IF) are not allowed in the WHEN clause of a SEARCH verb.

10. In either option, after any WHEN or AT END statement(s) is executed, control is transferred to NEXT SENTENCE unless that statement contained a GO TO.

11. If identifier-1 is a data item subordinate to a data item that contains an OCCURS clause (that is, this is a multidimensional table), only the index associated with identifier-1 is modified during the search. To search an entire multidimensional table, the SEARCH statement must be executed several times.

Example

```
01  TABLE.
    02  TABL1 OCCURS 200 TIMES INDEXED BY I,
        ASCENDING KEYS A, B.
        03 A       PICTURE XXX.
        03 FOO     PICTURE X(20).
        03 B       PICTURE 9(4).
        03 DES     PICTURE X(40).
        03 AM      PICTURE S9(5)V99.

SEARCH ALL TABL1, AT END GO TO WHAT-HAPPENDED;
    WHEN A(I) = "XYZ" AND B(I) = 350 GO TO GO-ONE.
```

5.9.34 SEEK

Function

The SEEK statement initiates the accessing of a mass storage data record in a random access file for subsequent reading or writing.

General Format

```
SEEK file-name RECORD
```

Technical Notes

1. The SEEK statement uses the contents of the ACTUAL KEY item to position the read-write arms on a mass storage device. If the key is invalid, no action is taken; however, if the contents of ACTUAL KEY are not changed before the next READ or WRITE statement is executed, that READ or WRITE statement then takes the INVALID KEY path.

2. The file must be assigned to a mass-storage device, and the ACCESS MODE must be RANDOM.

3. The statement cannot be used for files whose access modes are sequential or indexed.

4. Under timesharing, a SEEK to a public file structure is generally a waste of time, because many users are competing for positioning.

5.9.35 SET

Function

The SET statement allows a data-item to be incremented, decremented, or set to a value.

General Format

```
                                      ( TO      )   ( identifier-3 )
SET identifier-1 [,identifier-2] ...  { UP BY   }   { literal-1    }
                                      ( DOWN BY )
                                                         MR-S-1072-81
```

Technical Notes

1. All identifiers must be numeric elementary items described without any positions to the right of the assumed decimal point.

   All literals must be integers, or the figurative constant ZERO.

2. The SET statement causes identifier-1, identifier-2,... to be set (TO), incremented (UP BY), or decremented (DOWN BY) the value of identifier-3 or literal-1.

5.9.36 SORT

Function

The SORT statement creates a sort file containing the contents of one or more files that have been ordered according to user-specified keys.

General Format

```
SORT file-name-1 ON ASCENDING  KEY data-name-1
                    DESCENDING

    [,data-name-2] ... ON ASCENDING  KEY data-name-3
                          DESCENDING


SORT file-name-1 ON {ASCENDING }  KEY data-name-1
                    {DESCENDING}

    [,data-name-2] ... [ON {ASCENDING }  KEY data-name-3 [,data-name-4] ...] ...
                           {DESCENDING}

    {INPUT PROCEDURE IS procedure-name-1 [THRU procedure-name-2]}
    {USING file-name-2 [,file-name-3] ...                        }

    {OUTPUT PROCEDURE IS procedure-name-3 [THRU procedure-name-4]}
    {GIVING file-name-4                                          }
                                                        MR-S-1073-81
```

Technical Notes

1. File-name-1 must be described in an SD file description entry in the Data Division. Each data-name must represent data items described in records associated with file-name-1.

2. File-name-2, file-name-3, and file-name-4 must be described in an FD file description. All records associated with these files must be large enough to contain all of the KEY data-names. If multiple files must be read, you can use an input procedure that reads each file in turn. If you use SORT, you can use any number of input files with a SORT statement.

3. The data-names following the word KEY are listed in order of decreasing significance without regard to how they are organized in the SD record description.

4. The data-names can be qualified but not subscripted.

SORT (Cont.)

5. With SORT, the maximum record size is 4095 characters for ASCII, EBCDIC, and SIXBIT representations.

6. SORT statements can appear anywhere in the PROCEDURE DIVISION except in the DECLARATIVES portion or in an input or output procedure associated with a sort, or an output procedure associated with a merge.

7. When the ASCENDING clause is used, the sorted sequence is from the lowest value to the highest value; when a DESCENDING clause is used, the sorted sequence is from the highest value to the lowest value.

8. The input procedure, if present, must consist of one or more sections or paragraphs that appear contiguously in the program and do not form a part of any output procedure. The input procedure must contain at least one RELEASE statement to transfer records to the sort subroutine.

9. The output procedure, if present, must consist of one or more sections or paragraphs that appear contiguously in a source program and do not form a part of any input procedure. The output procedure must contain at least one RETURN statement to make sorted records available for processing.

10. ALTER, GO and PERFORM statements in the input procedure are not permitted to refer to procedure-names outside the input procedure; similarly, ALTER, GO and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure.

11. If an input or output procedure is specified, those procedures are PERFORMED by the SORT statement, and all rules relating to the range of a PERFORM must be observed.

12. If the USING option is specified, all records in file-name-2, file-name-3,..., are automatically transferred to the SORT subroutine. File-name-2, file-name-3,..., must not be open when the SORT statement is executed. Any USE PROCEDUREs associated with file-name-2, file-name-3,..., are executed as appropriate. The USING option is equivalent to the following INPUT PROCEDURE:

```
L1.  OPEN INPUT file-name-2
L2.  READ file-name-2 INTO sort-record; AT END GO TO
     L3.  RELEASE sort-record.
     GO TO L2.
L3.  CLOSE file-name-2.
```

SORT (Cont.)

13. If the GIVING option is specified, all the sorted records in file-name-1 are automatically transferred to file-name-4. File-name-4 must not be open when the SORT statement is executed. Any USE PROCEDURES associated with file-name-4 are executed as appropriate. The GIVING option is equivalent to the following OUTPUT PROCEDURE:

```
L4.  OPEN OUTPUT file-name-4.
L5.  RETURN sort-file INTO record-name-4;  AT END GO TO L6.
     WRITE record-name-4.
     GO TO L5.
L6.  CLOSE file-name-4.
```

14. An ISAM file can be sorted with INPUT and OUTPUT procedures. ISAM files cannot be sorted with the USING and GIVING options.

    ISAM files are by definition a sorted set. In designing the file you should use the order in which the file is most often accessed. If you wish to access it in a different order, write a program with an input procedure that reads the ISAM file sequentially using LOW VALUES. The input procedure can release records to the sort. If you wish to use an ISAM file as output, you must have an empty ISAM file for output, return records from the sort and write them into the new ISAM file.

STOP

5.9.37 STOP

Function

The STOP statement halts the object program.

General Format

$$\underline{\text{STOP}} \left\{ \begin{array}{l} \text{literal} \\ \underline{\text{RUN}} \end{array} \right\} \underline{.}$$

Technical Notes

1. If the literal option is used, the literal is displayed on your terminal, and the program waits for you to type

   ```
   CONTINUE
   ```

   Following receipt of this message, the program continues execution at the statement following the STOP.

   The literal can be a figurative constant; in this case, a single character is displayed.

2. If the RUN option is used, all files currently open are closed, and execution of the program is terminated.

5.9.38 STRING

Function

The STRING statement is used to concatenate the partial or complete contents of several data items into a single data item.

General Format

```
         { identifier-1 }  [ ,identifier-2 ]               { identifier-3 }
STRING   { literal-1    }  [ ,literal-2    ] ... DELIMITED BY { literal-3    }
                                                           { SIZE         }

   [    { identifier-4 }  [ ,identifier-5 ]               { identifier-6 } ]
   [ ,  { literal-4    }  [ ,literal-5    ] ... DELIMITED BY { literal-6    } ] ...
   [                                                      { SIZE         } ]

   INTO identifier-7 [ WITH POINTER identifier-8 ]

   [ ;ON OVERFLOW statement-1 ]
```

MR-S-1074-81

Technical Notes

1. Source Items

   a. The data items referenced by identifier-1, identifier-2,... are called source data items.

   b. A numeric source item is moved to an intermediate unsigned numeric data item of the same size as the source and whose USAGE is the same as that of identifier-7 according to the rules for numeric transfers, and then treated as alphanumeric.

   c. If subscripting or indexing is needed to identify a source data item, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the transfer of that particular source item are used.

   d. If a source item is defined in the Data Division using the OCCURS...DEPENDING option, STRING deals with the source item as if it were just long enough to contain the existing characters.

   e. Literal-1; literal-2... are called source literals. Source literals must be alphanumeric literals or alphanumeric figurative constants without the ALL modifier.

   f. If a source literal is a figurative constant, it refers to a single-character literal of the specified type.

STRING (Cont.)

2. Delimiter Items

   a. Each series of source items specified in the STRING statement must be followed by a DELIMITED BY phrase. This phrase specifies the delimiter condition to be associated with each source item in that series.

   b. The data items referenced by identifier-3 and identifier-6 are called delimiter data items.

   c. A numeric delimiter item is moved to an intermediate unsigned numeric data item of the same size as the delimiter and whose USAGE is the same as that of identifier-7 according to the rules for numeric transfers and then treated as alphanumeric.

   d. If subscripting or indexing is needed to identify a delimiter data item, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the transfer of the source item corresponding to that particular delimiter item are used.

   e. Literal-3 and literal-6 are called delimiter literals. Delimiter literals must be alphanumeric literals or alphanumeric figurative constants without the ALL modifier.

   f. If a delimiter literal is a figurative constant, it refers to a single-character literal of the specified type.

   g. If a delimiter data item or a delimiter literal is specified, the content of the data item during the execution of the STRING statement, or the value of the literal is the delimiter string for each source item corresponding to that delimiter item.

      In this case, the delimiter condition for each of the corresponding source items is the first occurrence in the source item of a character string that matches the delimiter string. If there is not such character string in the source item, the delimiter condition is the rightmost boundary of that source item.

      NOTE

      Two character strings match if, and only if, they are of equal length and each character of the first string is equivalent, according to the rules for code conversion, to the corresponding character of the second string.

   h. If the DELIMITED BY SIZE phrase is specified, the only delimiter condition for each of the corresponding source items is the rightmost boundary of the source item.

3. Destination

   a. The data item referenced by identifier-7 is called the destination. The destination must be an unedited alphanumeric data item. It cannot be justified (that is, the JUSTIFIED clause cannot be used for this item).

   b. If subscripting or indexing is needed to identify the destination, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the execution of the STRING statement are used.

4. Pointer

   a. The data item referenced by identifier-8 is called the pointer. The pointer must be an unedited integer data item of sufficient size to contain a value one greater that the size of the destination.

   b. The pointer serves as a character index for the destination.

   c. If subscripting or indexing is needed to identify the pointer, the values of the required subscripts and/or indexes and the depending items, if any, prior to the execution of the STRING statement are used.

   d. If the POINTER phrase is specified, the pointer is directly available to you. It must be initialized before the execution of the STRING statement to a value greater than zero and not greater than the size of the destination.

   e. If the POINTER phrase is not specified, the STRING statement is always executed as if you have specified a pointer and set the initial value to 1. In this case, the pointer is not directly available to you.

   f. The STRING statement is executed as if the initial value of the pointer were stored in a temporary location. This temporary location is used as the pointer during the execution of the STRING statement. The value in this temporary location is stored in the real pointer item before any subscripting is done and at the end of execution of the STRING statement.

5. Execution

   a. When the STRING statement is executed, each source item in turn, starting with the first source item specified, is transferred to the destination character-by-character, beginning at the leftmost character position of the source item and continuing to the right, until the delimiter condition corresponding to that source item has been encountered or the destination has been filled.

STRING (Cont.)

b. If a delimiter item was specified for a source item and a string of characters is found in the source item matching the delimiter string, all characters of the source item preceding the matching string are used in the transfer to the destination, but none of the characters that are in the matching string and no characters following it in the source item are used in the transfer.

c. If no delimiter item was specified for a source item or no string of characters is found in the source item matching the delimiter string, all characters of the source item are used in the transfer to the destination.

d. During the execution of the STRING statement, characters are transferred to the destination from the source items as if the destination were a table of single character data items indexed by the pointer, which is automatically incremented after each character transfer.

e. The first character transferred is stored in the character position of the destination indicated by the initial value of the pointer. The nth character transferred is stored in the character position indicated by the initial value of the pointer plus n-1.

f. The transfer of characters ends when one of the following conditions occur.

   These conditions are specifically checked for in the order stated:

   1. The initial value of the pointer is not a positive integer less than or equal to the size of the destination.

   2. All appropriate characters of all source items have been transferred to the destination.

   3. A character has been transferred to the last character position of the destination, though not all appropriate characters of all source items have been transferred.

g. If the transfer of characters to the destination is terminated because of condition 2 of note f, those character positions of the destination to which characters were not transferred, if any, retain the values they contained before the execution of the STRING statement. That is, remaining character positions in the destination are not space-filled.

h. After the transfer of characters to the destination has ended, the pointer is set to a value one greater than the ordinal number of the last character position of the destination to which data was transferred. If no data was transferred to the destination, the pointer is unchanged.

STRING (Cont.)

6. Overflow

   a. If the transfer of characters to the destination is terminated because of either condition 1 or condition 3 of note 5.f, the STRING statement is considered to have caused an overflow.

   b. IF THE ON OVERFLOW phrase is not specified, after the execution of the STRING statement, regardless of whether or not there was an overflow, control passes to the point in the program immediately following the STRING statement.

   c. If the ON OVERFLOW phrase is specified, after the transfer of characters has ended and the pointer set to the appropriate value, the flow of program control depends on whether or not there was an overflow.

      1. If an overflow did not occur, control passes to the point in the program corresponding to the end of the sentence containing the STRING statement (following all the statements in the ON OVERFLOW phrase).

      2. If an overflow did occur, control passes to the point in the program corresponding to the beginning of statement-1.

Example

```
DATA DIVISION.
01          INPUT-DATE.
            03  IN-MO PIC 99.
            03  IN-DA PIC 99.
            03  IN-YR PIC 99.
01          MONTH-TABLE.
            03  MON-TABLE PIC X(36) VALUE
                "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC".
            03  MON-ARRAY REDEFINES MON-TABLE.
                05  MO-NAME OCCURS 12 TIMES PIC XXX.
01          OUTPUT-DATE PIC X(12).
PROCEDURE DIVISION.
      .
      .
            STRING IN-DA "-" MO-NAME (IN-MO) "-" IN-YR
             DELIMITED BY SIZE INTO OUTPUT-DATE.
      .
      .
            STOP RUN.
```

NOTE

The DELIMITED BY clause specifies a character of data which is to serve as the rightmost terminator of the input field, unless SIZE is specified.

SUBTRACT

5.9.39 SUBTRACT

Function

The SUBTRACT statement is used to subtract one, or the sum of two or more, numeric items from one or more numeric items and set the values of one or more items to the result.

General Format

Option 1:

```
SUBTRACT  { identifier-1 }  [ , { identifier-2 } ] ...
          { literal-1    }  [   { literal-2    } ]

    FROM  identifier-m  [ ROUNDED ] [ ,identifier-n  [ ROUNDED ] ] ...

    [ ON  SIZE  ERROR  statement-1  [ ,statement-2 ] ...  . ]
```

Option 2:

```
SUBTRACT  { identifier-1 }  [ , { identifier-2 } ] ...
          { literal-1    }  [   { literal-2    } ]

    FROM  { identifier-m }   GIVING  identifier-n  [ ROUNDED ]
          { literal-m    }

    [ identifier-p  [ ROUNDED ] ]

    [ ON  SIZE  ERROR  statement-1  [ ,statement-2 ] ...  . ]
                                                     MR-S-1075-81
```

SUBTRACT (Cont.)

Option 3:

```
SUBTRACT { CORRESPONDING }  identifier-1  FROM  identifier-2
         { CORR          }

    [ ROUNDED ][ ON  SIZE  ERROR  statement-1  [ ,statement-2 ] ... . ]
                                                           MR-S-1076-81
```

Technical Notes

1. Each SUBTRACT statement must contain at least two operands (that is, a subtrahend and a minuend). In options 1 and 2, each identifier must refer to an elementary numeric item, except that identifiers to the right of the word GIVING can refer to numeric edited items. In option 3, each identifier must refer to a group item.

   Each literal must be a numeric literal or the figurative constant ZERO.

2. The composite of all operands (that is, the data item resulting from the superimposition of all operands aligned by decimal point) must not contain more than 18 decimal digits for the non-BIS compiler or more than 36 for the BIS compiler. In either case a maximum of 18 digits can be stored in a receiving field.

   NOTE

   The BIS compiler is standard on the DECSYSTEM-20 and DECsystem-10. For KI based hardware, the non-BIS compiler is optional on the DECsystem-10. (See the COBOL-68 Installation Procedures.)

3. Option 1 causes the values of the operands preceding the word FROM to be added together, and this sum to be subtracted from the values of identifier-m, identifier-n, and so forth.

4. Option 2 causes the values of the operands preceding the word FROM to be added together, the sum subtracted from identifier-m or literal-m, and the result stored as the new values of identifier-n, identifier-p, and so forth. The current values of identifier-n, identifier-p, and so forth, do not enter into the computation.

5. Option 3 causes the data items in the group item associated with identifier-1 to be subtracted from and stored into corresponding data items in the group item associated with identifier-2. The criteria used to determine whether two items are corresponding are described under "The CORRESPONDING Option" at the beginning of this chapter.

6. The ROUNDED and SIZE ERROR options are described earlier in this chapter under "Common Options Associated with Arithmetic Verbs".

SUPPRESS

5.9.40 SUPPRESS

Function

The SUPPRESS statement inhibits the presentation of a report group.

General Format

```
SUPPRESS PRINTING
```

MR-S-1077-81

Technical Notes

1. The SUPPRESS statement can appear only in a USE BEFORE REPORTING procedure.

2. The SUPPRESS statement inhibits presentation of the report group named in the USE BEFORE REPORTING statement only.

3. Each time you wish your program to inhibit presentation of a report group you must make sure that your program executes the SUPPRESS statement; you cannot execute it only once if you wish the group to be suppressed all the time.

4. Execution of the SUPPRESS statement causes the following report group functions to be inhibited:

   - The presentation of the print lines of the group,
   - The processing of all LINE clauses in the group,
   - The processing of the NEXT GROUP clause in the group, and
   - The adjustment of LINE-COUNTER.

5.9.41 TERMINATE

Function

The TERMINATE statement ends the processing of a report.

General Format

<u>TERMINATE</u> report-name-1 [, report-name-2] ... <u>.</u>

Technical Notes

1. Each report-name must be defined by an RD entry in the REPORT SECTION of the DATA DIVISION.

2. All control footings associated with the report are produced as if a control break had occurred at the highest level. In addition, the last PAGE FOOTING and any REPORT FOOTING report groups are produced.

3. A second TERMINATE statement for a particular report can not be executed until another INITIATE statement is executed for that report.

4. The TERMINATE statement does not close the file associated with the report; a CLOSE statement must be executed after the TERMINATE statement is executed.

TRACE

5.9.42 TRACE

Function

The TRACE statement causes the compiler to generate calls to COBDDT. This allows you to trace paragraphs or to stop tracing paragraphs at run time. When a paragraph is traced, its name, enclosed in angle brackets (<>), is typed each time that the paragraph is entered.

General Format

$$\underline{\text{TRACE}} \left\{ \begin{array}{l} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{array} \right\}$$

MR-S-1078-81

Technical Notes

1. You must load COBDDT with your program to be able to use the trace facility. (Refer to Section 7.3.1, Loading and Starting COBDDT, for more information.)

2. The compiler generates trace calls for each paragraph in the program if the /P switch is not included in the command string. If the /P switch is included in the command string, the trace calls are not generated.

3. Although the compiler generates trace calls when the /P switch is not present, tracing is not performed unless you include the TRACE ON statement in your program (or specify the TRACE ON statement to COBDDT).

4. The TRACE ON statement causes all ensuing paragraphs to be traced; that is, their names, enclosed in angle brackets (<>), are typed each time they are entered. Tracing continues until either the end of program is reached or a TRACE OFF statement is encountered (or is specified to COBDDT).

5. The TRACE OFF statement stops tracing of all ensuing paragraphs until either the end of program is reached or a TRACE ON statement is encountered (or is specified to COBDDT).

TRACE (Cont.)

6. When compiling for a production run, you should include the /P switch in the command string so that trace calls are not generated and TRACE statements in the program are ignored. The following example shows paragraphs with TRACE OFF and TRACE ON statements included.

```
PROCEDURE DIVISION.
PARA.
     .
     .
     .
TRACE ON.
PARB.
     .
     .
     .
TRACE OFF.
PARC.
     .
     .
     .
TRACE ON.
PARD.
     .
     .
     .
```

Paragraphs PARB and PARD are traced. Paragraph PARC is not traced because the TRACE OFF statement is included immediately before it. If the /P switch is included in the command string when this program is compiled, the TRACE statements are ignored and trace calls are not generated.

5.9.43 UNSTRING

Function

The UNSTRING statement is used to split a single data item (for example, text string) into several parts, depending on the occurrence of specified delimiters, and to store the parts into separate data items where they can be more easily accessed by the COBOL program.

General Format

```
UNSTRING identifier-1

  [ DELIMITED BY [ALL] { identifier-2 } [ ,OR [ALL] { identifier-3 } ] ... ]
                       { literal-2    }              { literal-3    }

  INTO identifier-4 [ ,DELIMITER IN identifier-5 ] [ ,COUNT IN identifier-6 ]

     [ ,identifier-7 [ ,DELIMITER IN identifier-8 ] [ ,COUNT IN identifier-9 ] ] ...

     [ WITH POINTER identifier-10 ] [ TALLYING IN identifier-11 ]

     [ ,ON OVERFLOW statement-1 ]
                                  MR-S-1079-81
```

Technical Notes

1. Source Items

   a. The data item referenced by identifier-1 is called the source item. The source item must be a DISPLAY-6, DISPLAY-7, or DISPLAY-9 data item. A numeric source item is moved to an intermediate unsigned numeric data item of the same size according to the rules for numeric transfers and then treated as alphanumeric.

   b. If subscripting or indexing is needed to identify the source, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the execution of the UNSTRING statement are used.

2. Destination Items

   a. The data items referenced by identifier-4, identifier-7,..., are called destination items. Destination items can be any kind of data items.

UNSTRING (Cont.)

b. If subscripting or indexing is needed to identify a destination item, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the transfer of data to that destination item are used.

3. Delimiter Items

a. The data items referenced by identifier-2, identifier-3,..., are called delimiter data items.

b. A numeric delimiter item is moved to an intermediate unsigned numeric data item of the same size as the delimiter and whose USAGE is the same as that of identifier-1 according to the rules for numeric transfers and then treated as alphanumeric.

c. If subscripting or indexing is needed to identify a delimiter data item, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the transfer of data to each successive destination item are used.

d. Literal-1, literal-2,..., are called delimiter literals. Delimiter literals must be alphanumeric literals or alphanumeric figurative constants without the ALL modifier.

e. If a delimiter literal is a figurative constant, it refers to a single-character literal of the specified type.

f. If a delimiter data item or a delimiter literal is specified, the contents of the data item or the value of the literal is a delimiter string for the source.

g. If more than one delimiter item is specified, the delimiter items are separated by the connective OR. In this case, the several delimiter strings are ordered by the order in which the delimiter items specifying them occur in the UNSTRING statement.

h. If the ALL phrase is specified with a delimiter item, the delimiter string that that item specifies is considered to consist of as many occurrences of that simple delimiter string as can be found contiguously stored in the source.

i. A delimiter condition is an occurrence in the source of a character string, not contained in the portion of the source that has already been scanned, that matched one of the delimiter strings, or the rightmost boundary of the source.

UNSTRING (Cont.)

4. Delimiter Storage Items

   a. A DELIMITER IN phrase can be specified only if the DELIMITED BY phrase is specified.

   b. The data items referenced by identifier-5 and identifier-8 are called delimiter storage items.

   c. If subscripting or indexing is needed to identify a delimiter storage item, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the transfer of data to the destination item corresponding to that delimiter storage item are used.

5. Count Storage Items

   a. The data items referenced by identifier-6 and identifier-9 are called count storage items. Count storage items must be unedited integer data items of sufficient length to contain a value equal to the length of the source.

   b. If subscripting or indexing is needed to identify a count storage item, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the transfer of data to the destination item corresponding to that count storage item are used.

   c. A count storage item is used to store the number of characters of the source that were examined during the execution of the UNSTRING statement and approved for transfer to the destination corresponding to that count storage item.

      NOTE

      This is not necessarily the same as the number of characters that were actually transferred, because the destination can be too small to hold all that were approved for transfer.

6. Pointer

   a. The data item referenced by identifier-10 is called the pointer. The pointer must be an unedited integer data item of sufficient size to contain a value one greater than the size of the source.

   b. The pointer serves as a character index for the source.

   c. If subscripting or indexing is needed to identify the pointer, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the execution of the UNSTRING statement are used.

d. If the POINTER phrase is specified, the pointer is directly available to you. It must be initialized before the execution of the UNSTRING statement to a value greater than zero and not greater than the size of the source.

e. If the POINTER phrase is not specified, the UNSTRING statement is always executed as if you have specified a pointer and set the initial value to 1. In this case, the pointer is not directly available to you.

7. Destination Counter

a. The data item referenced by identifier-11 is called the destination counter. The destination counter must be an unedited integer data item of sufficient size to contain a value equal to the number of destination items specified in the UNSTRING statement.

b. The destination counter is used to store the number of destination items to which data was transferred by the execution of the UNSTRING statement.

c. If subscripting or indexing is needed to identify the destination counter, the values of the required subscripts and/or indexes and the depending items, if any, just prior to the execution of the UNSTRING statement are used.

d. If the TALLYING phrase is specified, the destination counter is directly available to you, and it must be initialized before the execution of the UNSTRING statement.

e. If the TALLYING phrase is not specified, the UNSTRING statement is always executed as if you had specified a destination counter and set the initial value to 0. In this case, the destination counter is not directly available to you.

8. Execution

a. The execution of the UNSTRING statement is an iterative process. There is one iteration for each destination item specified in the UNSTRING statement, starting with the first destination item specified and continuing in order through the series of destination items. However, the iteration process is stopped after all the data in the source has been used, even if not all destination items have been used. During execution of the UNSTRING statement, the pointer and an increment to be added to the destination counter are kept in temporary locations. At the start of execution of the UNSTRING statement, the real pointer is stored in the temporary pointer and the temporary destination count is set to zero. When it becomes necessary to move these items to the real pointer and real destination items, the internal pointer is moved into the real pointer, the internal destination counter is added to the real destination counter, and the internal destination counter is set to zero again.

UNSTRING (Cont.)

b. Each iteration of the process involved in the execution of the UNSTRING statement consists of the following steps:

   1. Select a set of characters from the source.

   2. If the destination item, delimiter storage item, or count storage item is subscripted, store the internal pointer into the real pointer item and update the real destination counter.

   3. Move a representation of these characters to the destination item for that iteration.

   4. Move some characters to the delimiter storage item corresponding to that destination item, if one is specified.

   5. Set the count storage item corresponding to that destination item, if one is specified.

   6. Advance the internal pointer to indicate a new position in the source.

   7. Increment the internal destination counter.

c. During the execution of the UNSTRING statement, the source is treated as if it were a table of single character data items indexed by the pointer. The character position of the source indicated by the pointer, during each iteration of the UNSTRING process, is called the pointer-indicated position for that interation. Only the pointer-indicated position for an iteration and those source character positions to its right are used during that iteration. Character positions to the left of that position are not involved in that iteration in any way.

d. During each iteration of the UNSTRING process, a scan of the source is done to determine which characters of the source are selected as the character set to be moved to the appropriate destination item. This scan begins at the pointer-indicated position and continues to the right in the source.

e. When the source is scanned, certain conditions are detected depending on whether or not the DELIMITED BY phrase is specified.

   1. If the DELIMITED BY phrase is specified, the scan ends when either of the following conditions occurs:

      a. A string of contiguous characters in the source that matches one of the delimiter strings is found.

      b. The rightmost boundary of the source is found.

UNSTRING (Cont.)

2. When the DELIMITED BY phrase is not specified, the scan ends when either of the following conditions occurs:

   a. A number of characters sufficient to completely fill the destination is found.

   b. The rightmost boundary of the source is found.

When the scan ends, the set of characters to be moved to the destination item is known.

f. The source scan proceeds in one of two ways depending on whether or not the DELIMITED BY phrase is specified.

1. If the DELIMITED BY phrase is specified, the scan proceeds as follows:

   a. Each character position of the source, starting at the pointer-indicated position and continuing to the right, is first checked to see if any source character-string beginning at that position matches the delimiter-string specified by the first delimiter item in the UNSTRING statement. If such a string is found, condition a of Note e1 is satisfied.

   b. If no such string is found, the same character position is then checked to see if any source character-string beginning at that position matches the second specified delimiter-string. This process is repeated using each successive delimiter-string until either condition a of Note e1 is satisfied or all specified delimiters have been tried.

   c. If condition a of Note e1 is not satisfied for the source character position under consideration and one of the specified delimiter-strings, that character position is then selected as part of the source to be moved to the current destination item.

   d. The above process continues until no more source character positions remain (condition b of Note e1).

2. If the DELIMITED BY phrase is not specified, the source scan proceeds until one of the following conditions occurs:

   a. Enough successive character positions of the source have been selected to entirely fill the destination item (conditon a of Note e2).

   b. No more source character positions remain (Condition b of Note e2).

UNSTRING (Cont.)

g. During each iteration of the UNSTRING process, the set of contiguous source character positions selected by the process described in Note f is considered to be a complete individual data item, and is moved to the current destination item according to the rules for the MOVE statement, including any class of usage conversion that might be necessary. You should note that truncation or fill can occur during the execution of the MOVE. This data item might contain no character positions at all if the pointer-indicated position satisfied condition a of Note el or it can contain as much as the entire source.

h. If a count storage item is specified with the destination item for an iteration of the UNSTRING process, the number of source characters that were examined during the execution of the UNSTRING statement and approved for transfer to the destination item is stored in that count storage item.

i. If there is a delimiter storage item specified for a particular iteration of the UNSTRING process, then:

   1. If the selection of source character positions described in Note f is terminated because condition a of Note el, the string of contiguous source character positions that contain the match for a delimiter string is treated as a complete individual data item and is moved to the delimiter storage item according to the rules for the MOVE statement, including truncation if necessary.

      If, in this case, the delimiter string that was matched is described with the ALL phrase, the set of source character positions containing a match for the simple delimiter string, plus every immediately succeeding set of contiguous source character positions containing a match for the same delimiter string, are used in the data item that is moved to the delimiter storage item.

   2. If the selection of source character positions described in Note f is terminated because of condition b of Note el, spaces are moved to the specified delimiter storage item.

j. In an iteration of the UNSTRING process, after the appropriate data has been stored in the destination item, the delimiter storage item, and the count storage item, the pointer is set to a value one more than the ordinal number of the last source character position that participated in the selection process. This includes all character positions that were selected as part of the source to be moved to the destination item and, if a DELIMITED BY phrase is specified, all character positions that were used in the successful match of a delimiter string.

UNSTRING (Cont.)

k. At the conclusion of execution of the UNSTRING statement, the real destination counter is updated using the internal destination counter and the internal pointer is stored into the real pointer.

9. Overflow

a. If the initial value of the pointer is less than one or greater than the size of the source, execution of the UNSTRING statement is aborted before any data is transferred, the real pointer's value is unchanged, and the UNSTRING statement is considered to have caused an overflow.

b. If, during the execution of an UNSTRING statement, data has been transferred to all of the destination items in accordance with Note g, but the updated pointer still contains a value less than or equal to the size of the source (that is, not all of the source character positions have been used in the UNSTRING process), the UNSTRING statement is considered to have caused an overflow.

c. If the ON OVERFLOW phrase is not specified, after the execution of the UNSTRING statement, regardless of whether or not there was an overflow, control passes to the point in the program immediately following the UNSTRING statement.

d. If the ON OVERFLOW phrase is specified, after the transfer of characters has ended and the pointer and destination counter are set to the appropriate values, the flow of control of the program depends on whether or not there was an overflow.

1. If an overflow did not occur, control passes to the point in the program corresponding to the end of the sentence containing the UNSTRING statement (following all the statements in the ON OVERFLOW phrase).

2. If an overflow did occur, control passes to the point in the program corresponding to the beginning of statement-1.

UNSTRING (Cont.)

Example

```
DATA DIVISION.
          .
          .
          .
01          INPUT-DATE PIC X(12).
01          DATE-FIELDS.
            03  DAY    PIC XX.
            03  MONTH PIC XXX.
            03  YEAR PIC XX.
              .
              .
              .
PROCEDURE DIVISION.
          .
          .
          .
            UNSTRING INPUT-DATE DELIMITED BY ALL "-"
             INTO DAY MONTH YEAR.
          .
          .
          .
            STOP RUN.
```

NOTE

This example assumes that the input date is in the form "01-JAN-81". If the day and year are to be used as numerics, they have to be moved to fields which are justified right. Also, if the month is to be converted to numeric, this must be done by means of a table search.

5.9.44 USE

Function

The USE statement specifies procedures for input-output label and error handling that are in addition to the standard procedures provided.

General Format

Format 1:

$$\underline{\text{USE}}\ \ \underline{\text{AFTER}}\ \ \text{STANDARD}\ \ \underline{\text{ERROR}}\ \ \underline{\text{PROCEDURE}}\ \ \text{ON} \left\{\begin{array}{l} \text{file-name-1}\ \ [\,\underline{\text{OPEN}}\,] \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \\ \underline{\text{INPUT-OUTPUT}} \end{array}\right\} \underline{.}$$

Format 2:

$$\underline{\text{USE}} \left\{\begin{array}{l} \underline{\text{BEFORE}} \\ \underline{\text{AFTER}} \end{array}\right\} \text{STANDARD} \left[\left\{\begin{array}{l} \underline{\text{BEGINNING}} \\ \underline{\text{ENDING}} \end{array}\right\}\right] \left[\left\{\begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{FILE}} \\ \underline{\text{UNIT}} \end{array}\right\}\right]$$

$$\underline{\text{LABEL}}\ \ \underline{\text{PROCEDURE}}\ \ \text{ON} \left\{\begin{array}{l} \text{file-name-1} \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \\ \underline{\text{INPUT-OUTPUT}} \end{array}\right\} \underline{.}$$

Format 3:

$$\underline{\text{USE}}\ \ \underline{\text{BEFORE}}\ \ \underline{\text{REPORTING}}\ \ \text{identifier-1}\ \underline{.}$$

MR-S-1080-81

Technical Notes

1. USE statements can appear only in the DECLARATIVES portion of the PROCEDURE DIVISION. The DECLARATIVES portion follows immediately after the PROCEDURE DIVISION header and begins in A-margin with the word

   DECLARATIVES.

USE (Cont.)

The DECLARATIVES portion ends in A-margin with the words

```
END DECLARATIVES.
```

Following this must be a section-header as the first entry of the main portion of the PROCEDURE DIVISION.

The DECLARATIVES portion itself consists of USE sections, each consisting of a section-header, followed by a USE statement, followed by the associated procedure paragraphs.

The general format for the DECLARATIVES portion is given below:

```
PROCEDURE DIVISION.

DECLARATIVES.

section-name-1 SECTION. USE......
paragraph-name-1a. (statement)
[paragraph-name-1b. (statement)]
.
.
[section-name-2 SECTION. USE......]
.
.
.
END DECLARATIVES.

section-name-m SECTION.
```

2. The USE statement can follow on the same line as the section-header and must be terminated by a period followed by a space. The remainder of the section must consist of one or more procedural paragraphs that define the procedures to be used.

3. The USE statement itself is never executed, rather it defines the conditions calling for the execution of the USE procedures.

4. The designated procedures are executed at the appropriate time as follows:

   a. Format 1 causes the designated procedures to be executed after completing the standard input-output error routine. This provides the ability to continue a program on an I/O error if it would have otherwise failed. Refer to the FILE-STATUS clause in Chapter 3 for details of specifying a FILE-STATUS action code.

   b. Format 2 causes the designated procedures to be executed at one of the following times, depending upon the options used.

      1. Before or after a beginning or ending input label check procedure is executed.

      2. Before a beginning or ending output label is created.

3. After a beginning or ending output label is created, but before it is written on tape.

4. Before or after a beginning or ending input-output label check procedure is executed.

c. Format 3 causes the designated procedures to be executed just prior to the production of the named report group.

5. There must not be any reference to any non-DECLARATIVES procedure within a USE procedure. Conversely, there must be no reference to procedure-names that appear within the DECLARATIVES portion in the non-DECLARATIVES portion, except that PERFORM statements can refer to a USE section or to a procedure contained entirely within such a USE section.

6. No input/output can be performed, other than use of ACCEPT and DISPLAY statements, during execution of a USE procedure.

7. Format 1 causes the associated procedures to be executed after the standard input-output error routine has been executed. If the INPUT option is used, the procedures are executed for all INPUT files; if the OUTPUT option is used, they are executed for all OUTPUT files; if the I-O or the INPUT-OUTPUT option is used, they are executed for all INPUT-OUTPUT files; if the filename-1 option is used, they are executed only for that particular file. If the filename-1 option is used and any other option is used (i.e., INPUT, OUTPUT, INPUT-OUTPUT), the file named in the filename-1 option cannot be opened in the form that is specified in the other error procedure. For example, if you specify two error procedures, one for INPUT and the other for filename-1, filename-1 cannot be opened for INPUT; it can be opened for OUTPUT or INPUT-OUTPUT.

   If the filename-1 OPEN option is used, the system performs the associated procedures only if a "FILE BEING MODIFIED" error occurs when an attempt is made to open an output file. After performing the procedure, the system automatically tries again to open the file, repeating this process until the file is opened. This option allows you to suspend your job until it can access a file that another user is modifying.

8. Format 2 causes the associated procedures to be executed at the appropriate times, as indicated by the options selected (see note 4). If the words BEGINNING or ENDING are not included in Format 2, the designated procedures are executed for both the beginning and ending labels. If neither UNIT, REEL, nor FILE is included, the designated procedures are executed for both REEL (or UNIT) labels and for FILE labels.

   If the INPUT, OUTPUT, INPUT-OUTPUT, or I-O option is specified, the label procedure applies to every file of that type except files described as LABEL RECORDS ARE OMITTED.

   If the file-name-1 option is used, its file description must not contain a LABEL RECORDS ARE OMITTED clause.

USE (Cont.)

9. Within a given format, a file-name must not be referred to, either implicitly (that is, by an INPUT, OUTPUT, INPUT-OUTPUT, or I-O option) or explicitly (that is, by a file-name-1 option), in more than one USE statement.

10. Identifier-1 in Format 3 represents a report group named in the REPORT SECTION of the DATA DIVISION. An identifier must not appear in more than one USE statement. The report group must not be TYPE DETAIL.

5.9.45 WRITE

Function

The WRITE statement transfers a logical record to an output file.

General Format

Format 1:

```
WRITE record-name-1 [FROM  figurative-constant]
                    [      identifier-1       ]

    [                   ( identifier-2 LINES ) ]
    [ { BEFORE }        ) integer-1 LINES    ( ]
    [ { AFTER  } ADVANCING  mnemonic-name    ) ]
    [                   ( PAGE               ) ]
```

Format 2:

```
WRITE record-name-1 [FROM  figurative-constant]
                    [      identifier-1       ]

    AFTER POSITIONING { identifier-2 } LINES
                      { integer-1    }
```

Format 3:

```
WRITE record-name-1 [FROM  figurative-constant]
                    [      identifier-1       ]

    INVALID KEY statement-1 [,statement-2] ... .
```

MR-S-1081-81

Technical Notes

1. An OPEN OUTPUT or OPEN I-O or OPEN INPUT-OUTPUT statement must be executed for the file prior to the execution of the WRITE statement.

2. After the WRITE is executed, the data in record-name-1 is no longer be available.

3. Record-name-1 must be the name of a logical record in a DATA RECORDS clause of the FILE SECTION of the DATA DIVISION.

WRITE (Cont.)

4. Option 1 is valid for any file currently open for output, with ACCESS MODE IS SEQUENTIAL. The ADVANCING clause allows control of the vertical positioning of the paper form for print files as follows:

   a. If the ADVANCING clause is not specified and the recording mode is ASCII, BEFORE ADVANCING 1 LINE is assumed.

   b. If identifier-2 or integer-1 is specified, it must represent a positive integer or zero. The form is advanced the number of lines equal to the value of identifier-2 or integer-1.

   c. If mnemonic-name is specified, the form is advanced until the specified channel is encountered on the paper-tape format control loop. Mnemonic-name must be defined by a CHANNEL clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

   d. If the BEFORE option is used, the record is printed before the form positioning.

   e. If the AFTER option is used, the record is printed after form positioning occurs, and no form positioning takes place after the printing.

   If end-of-reel is encountered while writing on magtape, the WRITE statement performs the following operations.

   a. Your ENDING LABEL PROCEDURE is executed, if specified by a USE statement.

   b. A file mark is written, and the tape is rewound.

   c. If the file was assigned to more than one tape unit, the units are advanced.

   d. A label is written on the new tape, if labels are not OMITTED, and your BEGINNING LABEL PROCEDURE is executed.

5. Option 2 is valid for any file currently open for output, with ACCESS MODE IS SEQUENTIAL.

   The POSITIONING clause allows control of the vertical positioning of the paper form for print files. The record is written after the printer page is advanced according to the following rules.

WRITE (Cont.)

a. If identifier-2 is specified, it must be described as a 1-character alphanumeric item; that is, with PICTURE X. The valid values that identifier-2 can contain and their interpretations are as follows.

blank	Single-spacing
0	Double-spacing
-	Triple-spacing
+	Suppress spacing
1-8	Skip to channels 1 through 8 respectively on the paper-tape format control loop

Note that LIBOL interprets the values in identifier-2, substituting the proper positioning characters into the ASCII file. The character stored in the field named identifier-2 is not stored in the output file.

b. If integer-1 is specified, it must be unsigned, and must be one of the values 0, 1, 2, or 3. The values have the following meanings.

0	Skip to channel 1 of next page (carriage control "eject")
1	Single-spacing
2	Double-spacing
3	Triple-spacing

6. Either ADVANCING or POSITIONING can be specified for a file, but not both. Also, if either is specified, the recording mode of the file is ASCII, regardless of the recording mode specified in the RECORDING MODE clause.

7. Option 3 is valid only for random-access files whose access mode is RANDOM or INDEXED. The imperative-statement(s) in the INVALID KEY clause of a RANDOM file is executed when an attempt is made to execute a WRITE to a segment outside the range of the FILE-LIMITS of the file.

   When a WRITE statement is executed for a file whose access mode is RANDOM and the ACTUAL KEY contains a value of 0, records are written sequentially in the file (that is, no records are left null). If the previous operation performed on the file was by a READ statement, the previous record is replaced (that is, the record is updated).

   When a WRITE statement is executed for a file whose access mode is INDEXED, the contents of the SYMBOLIC KEY item are moved to the RECORD KEY item and the record is written.

   The statement(s) in the INVALID KEY clause is executed when the SYMBOLIC KEY contains a value equal to the key of an already existing record in an INDEXED file (refer to the REWRITE statement).

WRITE (Cont.)

8. When executing a WRITE statement for a SEQUENTIAL file opened for I-O (or INPUT-OUTPUT), the logical record is placed on the file as the next logical record, if the previous input-output operation was a WRITE, or it replaces the previous record, if the previous input-output operation was a READ.

9. If the FROM option is used, the statement is equivalent to:

```
MOVE identifier-1 TO record-name-1
WRITE record-name-1 (without the FROM option)
```

CHAPTER 6

COMPILING COBOL-68 PROGRAMS

Compiling COBOL-68 programs consists of running the COBOL-68 compiler and typing the correct command string in response to the prompt. To run COBOL and compile your program(s), type R COBOL in response to the TOPS-10 prompt (.) or COBOL in response to the TOPS-20 prompt (@). That is,

```
     .R COBOL (RET) for users of TOPS-10

             or

     @COBOL (RET) for users of TOPS-20
```

The general form of the compiler command string is as follows:

```
     relfil,lstfil= libfil/L, src1, libfil/L, src2,...
```

where:

relfil	is the file that is to hold the generated code. If no generated code is desired, the file description for relfil is replaced by a hyphen. Example: -,lstfil=src1,src2...
lstfil	is the file that is to hold the generated listing. If no listing is desired, the file description for lstfil is replaced by a hyphen. Example: relfil,-=src1,src2,...
libfil	is an optional library file referenced by COPY verbs in the source files. For each source file specified, only one library file opens at once. Thus, you can specify more than one library for a source file, but only the last-specified one is used.
src1,src2	are one or more source files required to form one complete input program.

Each file description has the following form:

```
     device: file.ext [project,programmer] /switch/switch
```

where:

device	is the name of a physical or logical device. The name is composed of 6 or fewer letters and/or digits.

file	is the name of a file. The name is composed of 6 or fewer letters and/or digits.
ext	is the filename extension. It is composed of 3 or fewer letters and/or digits.
project	is a user's project number.
programmer	is a user's programmer number.
switch	is any of the switches shown in Table 6-1.

Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative which is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

Certain default assignments are made by the compiler whenever terms are omitted from the command strings or the file descriptions.

1. If the device is omitted in any output file description, DSK is assumed. If the device is omitted in an input file description, either the preceding device or DSK (if no preceding device is specified) is assumed.

2. If the filename for relfil and/or lstfil is omitted, the filename of the first source file is used.

3. If the filename extension is omitted from relfil, .REL is assumed; if it is omitted from lstfil, .LST is assumed. If the extension is omitted from the source file descriptor, the compiler looks in the file area for the named file with the extension .CBL. If that file is not found, the compiler looks for the named file with the extension .COB. If that file is not found, the compiler looks for the named file without an extension. If the extension is omitted from the library file description, .LIB is assumed.

4. If the project-programmer option is omitted on any file, your default path is used. On TOPS-20, the connected directory is used.

Examples:

```
MTA1:RELOUT.A/W,LPT:=DSK:SRCIN.C [27,36]/M/S
```

The compiler compiles the program found in the file SRCIN.C in the area reserved for project-programmer [27,36]. It treats columns 1-6 of the source as a sequence number (/S). The generated code is written on MTA1, after the tape is rewound (/W). The listing, including maps (/M) is put on the LPT.

```
=LIB1/L,PROG/A
```

The compiler compiles the program found in PROG.CBL (CBL is assumed because the filename extension is omitted from the source file descriptor) on the disk, using LIB1.LIB whenever a COPY verb is seen (/L). The generated code goes into the file DSK:PROG.REL, and the listing onto the file DSK:PROG.LST. The generated code is listed (/A).

```
     -=LIB1/L,PROG/A
```

This is identical to the preceding example, with the exception that no generated code is produced because the file descriptor for the file has been replaced by a hyphen. If you wish to produce the .REL file but not the .LST file, you should add a comma to the command string, as follows:

```
     ,-=LIB1/L,PROG/A
```

The following table shows the switches that can be used in compiling COBOL-68 programs.

Table 6-1
COBOL Switch Summary

Switch	Action by Compiler
A	List the MACRO reproduction of the generated code in the lstfil.
C	Produce a cross-reference table of all user-defined symbols.
D:nnnnnn	Increment, in octal words, that is to be added to the object-time push down list size.
E	Check program for errors, but do not generate code.
H	Type description of COBOL-68 command strings and switches.
I	Suppress output of start address (program is to be used only by CALLs).
J	Force output of start address in spite of the presence of subprogram syntax.
L	Use the preceding source file as a library file whenever a copy verb is encountered. If the first source file is not a /L file, LIBARY.LIB is used as the library file until the first /L file is encountered. (The default extension for library files is .LIB.)
M	Include a map of your defined items in the lstfil.
N	Do not type compilation errors on your terminal.
O	Optimize the object code.
P	Production mode. Omit debugging features from relfil.
Q	Quick mode. Do not range check PERFORMs, also turn on /O and /P.

Table 6-1 (Cont.)
COBOL Switch Summary

Switch	Action by Compiler
R	Produce a two-segment object program. The high segment contains the Procedure Division; the low segment all else.
S	The source file is in "conventional" format (with sequence numbers in columns 1-6 and comments starting in column 73).
U	Produce a one-segment object program. This is the default for TOPS-10.
W	Rewind the device before reading or writing (magnetic tape only).
X	Give a usage of DISPLAY-9 to items whose usage is either omitted or declared as DISPLAY.
Z	Zero the directory of the device before writing (DECtape only).

CHAPTER 7

COBOL UTILITY PROGRAMS

COBOL provides several utility programs that allow you to perform certain operations within your COBOL program. These utility programs are:

- ISAM - **Indexed-Sequential File Maintenance Program**

  ISAM provides you with the ability to create and maintain indexed-sequential files (see section 7.1).

- LIBARY - **Source Library Maintenance Program**

  LIBARY provides you with the facility to create, modify, and delete statements or groups of statements in a library file (See Section 7.2).

- COBDDT - **Program For Debugging COBOL Programs**

  COBDDT provides you with the ability to:

  1. Look for areas of error by setting breakpoints
  2. Trace the activity of procedures
  3. Display and, if necessary, change the contents of data-items
  4. Determine time spent in sections of the program by analyzing a histogram (see Section 7.3)

- RERUN - **Program to Restart COBOL Programs**

  RERUN provides you with the ability to restart a COBOL program after an abnormal termination has occurred (See Section 7.4).

NOTE

Many of the examples in this chapter are written for only one operating system - that is, they have either the TOPS-10 prompt (.) or the TOPS-20 prompt (@) alone. However, unless you are told otherwise, the examples apply to both TOPS-10 and TOPS-20. Thus, in this chapter you can substitute

```
        .R (program name)(RET)
```

for

```
        @(program name)(RET)
```

and vice versa.

7.1 ISAM - INDEXED-SEQUENTIAL FILE MAINTENANCE PROGRAM

Indexed-sequential files are created, maintained, and compacted for backup storage by means of the ISAM program. ISAM performs the following functions:

1. Builds an indexed-sequential file from a sequential file
2. Maintains an indexed-sequential file by reorganizing it
3. Packs an indexed-sequential file into a sequential file for backup storage

ISAM has the following switches that you can use to perform these functions:

A	Advance between records according to the mode specified (this switch can be used only with the P switch)
B	Build an indexed file from a sequential one
C	Check an indexed file for errors
I	Ignore errors in packing a file (this switch can be used only with the P switch)
L	Read or write standard tape labels (this switch can be used only with the B or P switches)
M	Maintain an indexed file by reorganizing it
P	Pack an indexed file for backup storage
R	Rename an indexed file
S	Provide statistics on blocking factors (this switch can be used only with the B switch)

Figure 7-1 shows the COBOL ISAM File Environment.

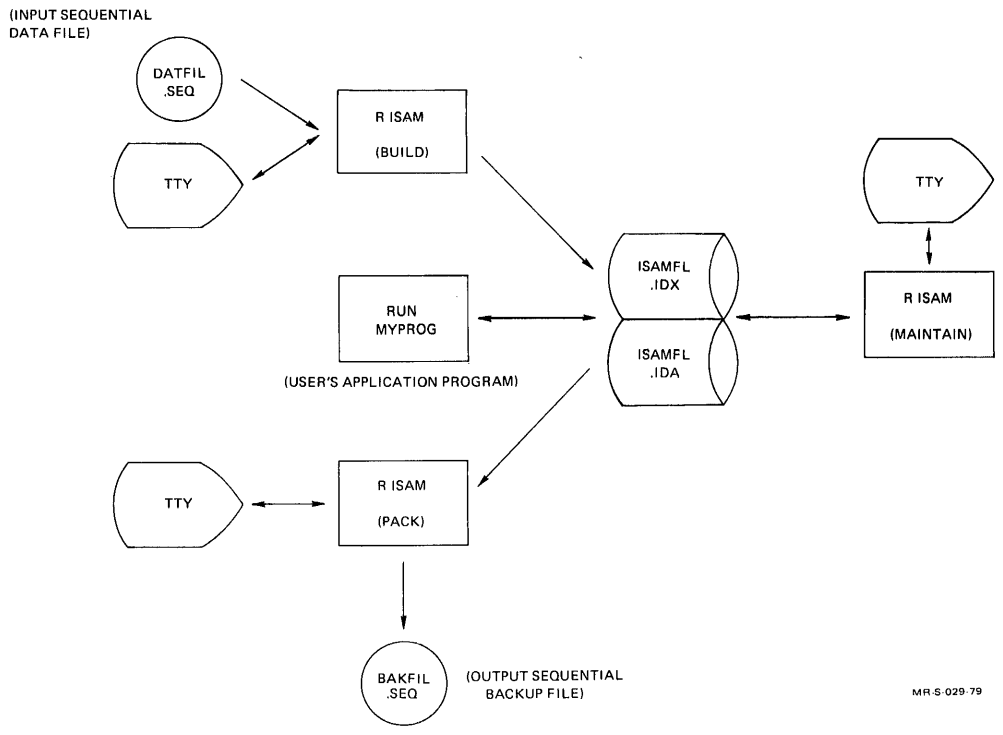


Figure 7-1 COBOL ISAM File Environment

7.1.1 Building An Indexed-Sequential File

To build an indexed-sequential file you provide a sequential file in which the record keys are arranged in ascending order. The ISAM program uses this file to create an indexed-sequential data file with a user-specified number of empty records and blocks. ISAM then creates the index file according to the description of the data file.

To run the ISAM program and select the option for building the indexed-sequential file, type the following:

```
.R ISAM(RET)   for users of TOPS-10

               or

@ISAM(RET)   for users of TOPS-20

*dev1:indfil.ext[ppn1],dev2:datfil.ext=dev3:seqfil.ext[ppn2]/B(RET)
```

where:

dev1, dev2, and dev3 are the devices for the index, data, and input sequential file. dev1 and dev2 must be disk. The default for dev1, dev2, and dev3 is DSK.

indfil.ext is the name and extension of the index file. If the filename is not specified, the name of the input file is assumed. If the extension is omitted, .IDX is assumed.

datfil.ext is the name and extension of the indexed data file. If the filename is omitted, the name of the index file is assumed. If the extension is omitted, .IDA is assumed.

seqfil.ext is the name and extension of the input sequential file. This filename must be specified, but the extension can be omitted. If it is omitted, .SEQ is assumed.

[ppn1], [ppn2] specify directories for the index file and the input file, respectively. If either is omitted, then the directory of the logged-in user is assumed. The data file must reside in the same directory as the index file. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative which is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/B is the switch signifying that ISAM is used to build an indexed-sequential file. If the switch is omitted from the command string, /B is assumed. The equal sign (=) can be omitted if the specifications for the output files are omitted.

Users can build an indexed file without providing a sequential file to the ISAM program by specifying that the device on which the (nonexistent) input sequential file resides is NUL:. Thus, the following command produces an indexed file:

```
TSTFIL,TSTFIL=NUL:TSTFIL/B
```

After reading the command string, ISAM asks a series of questions, which are described below. Every question must be answered.

```
Mode of input file:
```

Reply with S, A, F, V, or ST according to the mode of the input file. S means SIXBIT, A means ASCII, F means fixed-length EBCDIC, V means variable-length EBCDIC, and ST means STANDARD-ASCII.

```
Mode of data file:
```

Specify S, A, F, or V according to the mode in which the ISAM data file is to be recorded. S means SIXBIT, A means ASCII, and both F and V mean EBCDIC, as above. If the mode of the input file differs from that of the data file, characters are converted in the same manner as they are converted in standard COBOL operations.

```
Maximum record size:
```

Specify the size of the largest record in the input file in bytes. For ASCII records you should not count the carriage return and line feed that are appended to each ASCII record.

```
Key descriptor:
```

Describe the key upon which the file is to be indexed. If your records are of variable length, you must be sure that the key field occurs in the fixed portion of the record, or that the key field is filled with characters, in the case of an ASCII text file. If you fail to fill the key field in the ASCII file, your key field might be loaded with a carriage return/line feed pair.

In response to the message from ISAM, you describe the key field using a code that has the form:

```
[s] [x]m.n
```

where:

s designates the sign of the key:

S - the key is signed

U - the key is unsigned

x indicates the key type:

X - the key is nonnumeric

N - the key is numeric display

C - the key is COMPUTATIONAL

F - the key is COMPUTATIONAL-1

P - the key is COMPUTATIONAL-3

m specifies the character position in the record where the key begins, assuming that the position of the first character in the record is 1.

n specifies the size of the key in characters for types X and N or in digits for types C and P. If n is less than or equal to 10 for type C, one word is used. If n is greater than 10, two words are used. n is ignored for type F because it is always one word long.

The following rules apply to the key descriptor:

1. The key type is optional; if S or U are specified, the default is N. Otherwise, the default is X.
2. The key sign is optional; the default is S if the key type is not X.
3. The sign designators S or U cannot be specified in conjunction with type X.
4. m and n must be specified.

```
Records per input block:
```

Give the blocking factor of the input file. If the file is unblocked (that is, the file contains ASCII text), enter 0.

```
Total records per data block: (Recommended = n):
```

Give the total number of records to be contained in each block of the indexed data file. ISAM supplies the blocking factor that is most efficient in terms of disk utilization. If you wish to optimize the amount of core the object-time system requires to process the file, you must calculate the blocking factors yourself.

```
Empty records per data block:
```

Specify the number of records that are to be initially left empty in each block of the data file.

Total entries per index block: (Recommended: = n):

Specify the total number of index entries to be contained in each block of the index file. ISAM supplies the blocking factor that is most efficient in terms of disk utilization. If you wish to optimize the amount of core the object-time system requires to process the file, you must calculate the blocking factors yourself. Usually, a good blocking factor is a page (or less) of memory, since this allows the internal swapping algorithms to operate most efficiently.

Empty entries per index block:

Specify the number of index entries that are to be initially left empty in each index block. Note that at least two entries must be available in each index block, so that the number of total entries minus the number of empty entries equals or exceeds two.

Percentage of data file to leave empty:

Give a figure that corresponds to the amount of empty space you wish ISAM to allocate. This empty space corresponds to completely empty data blocks, over and above the free space that is allocated on "live" data blocks. ISAM figures the total number of data blocks required for the file, then add the number of blocks which equals the given percentage of the required blocks. This procedure helps to prevent "DISK FULL" crashes.

Percentage of index file to leave empty:

Give a figure that corresponds to the amount of empty space you wish ISAM to allocate. ISAM figures the total number of index blocks required for the file, then add the number of blocks which equals the given percentage of the required blocks. As with the data file, this procedure helps to prevent "DISK FULL" crashes.

Maximum number of records file can become:

Reply with the maximum number of records the file can contain before it is next maintained. Note, however, that your number is used only for documentation, since ISAM calculates its own number and uses that number for allocating storage.

At this point, ISAM supplies you with some information on the efficiency of your disk usage. The form of the information is as follows:

```
[ISMLOV m Levels of index ]
[ISMNDR n Data records ]

[ISMWSD Wasted p.  words of q.   r% wasted space in the Data
file.]
[ISMLDE One logical Data block equals s.  physical disk blocks.]
[ISMWSI Wasted t.  words of u.  in the Index file.]
[ISMLIE One logical Index block equals v.  physical disk blocks.]
[ISMIBS LIBOL's I/O buffer will require w.  words (x.  pages) of
core.]
```

Example - Building an indexed-sequential file

```
.R ISAM
*TEST.IDX,TEST.IDA=TEST.SEQ/B
Mode of input file: SIXBIT
Mode of data file: SIXBIT
Maximum record size: 240
Key descriptor: X129.29
Records per input block: 10
Total records per Data block: (Recommended: 53): 25
Empty records per Data block:
Total entries per Index block: (Recommended: 18):
Empty entries per Index block:
Percentage of Data file to leave empty:
Percentage of Index file to leave empty:
Maximum number of records file can become: 10000


[ISMLOV 2 Levels of index ]
[ISMNDR 500 Data records ]

[ISMWSD Wasted 127. words of 1152.  11.0% wasted space in
the Data file.]
[ISMLDE One logical Data block equals 9. physical disk
blocks.]
[ISMWSI Wasted 0. words of 128. in the Index file.]
[ISMLIE One logical Index block equals 1. physical disk
blocks.]
[ISMIBS LIBOL's I/O buffer will require 2816. words (6.
pages) of core.]
```

7.1.2 Maintaining An Indexed-Sequential File

The ISAM program allows you to maintain an existing ISAM file after the file has become crowded. More empty space can be added to the file and the number of index levels can be decreased. That is, the files are rearranged and indexes are streamlined. The input is the indexed-sequential file and the output is a new indexed-sequential data and index file. The command string for the ISAM maintain option is as follows:

```
.R ISAM(RET)  for users of TOPS-10

          or

@ISAM(RET)  for users of TOPS-20

*dev1:indfil.ext[ppn1],dev2:datfil.ext=infil.ext[ppn2]/M(RET)
```

where:

dev1, and dev2, are disk devices on which the files are stored. If any of the devices is omitted, DSK is assumed.

indfil.ext is the name and extension of the new index file. If the name is omitted, the name of the input file is assumed. If the extension is omitted, .IDX is assumed.

datfil.ext is the name and extension of the new indexed data file. If the name is omitted, the name of the new index file is assumed. If the extension is omitted, .IDA is assumed.

infil.ext is the name and extension of the index file of the old indexed-sequential file. The name of the file must be specified, but the extension can be omitted. No extension is assumed if the extension is omitted.

[ppn1], [ppn2] specify directories for the new index file and the old index file, respectively. If either is omitted, the directory of the logged-in user is assumed. The new data file must reside in the same directory as the new index file. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/M is the switch indicating that the maintain option is being requested. The switch must be specified.

If the output file specifications are not included in the command string, the equal sign (=) can be omitted.

After the command string has been scanned, ISAM asks a series of questions about values for the new indexed-sequential file. The mode of the file, the record size, and the key cannot be changed. The values from the old file are given in parentheses with the question. If you wish to change a value, enter the new value; if you do not wish to change a value, press the RETURN key. All questions refer to the output file.

Total records per data block (n):

Specify the total number of records to be contained in each block of the data file.

Empty records per data block (n):

Give the number of data records that are to be initially left empty in each data block.

Total entries per index block (n):

Give the total number of index entries to be contained in each block of the index file. Usually, a good blocking factor is a page (or less) of memory, since this allows the internal swapping algorithms to operate most efficiently.

Empty entries per index block (n):

Specify the number of index entries that are to be initially left empty in each index block.

Percentage of data file to leave empty (n):

Give, as a percentage of the total number of blocks, the number of blocks to be initially left empty in the data file. ISAM figures the total number of data blocks required for the file, then add the number of blocks which equals the given percentage fo the required blocks. This procedure helps to prevent "DISK FULL" crashes.

Percentage of index file to leave empty (n):

Give, as a percentage of the total number of blocks, the number of blocks to be initially left empty in the index file. ISAM uses the procedure described above to figure the amount of space to allocate, thus helping to prevent "DISK FULL" crashes.

Maximum number of record files can become (n):

Specify the maximum number of records that can be contained in the file. This number sets the upper limit on the size of the data file. It is required because storage allocation tables must be set up when the file is created, and ISAM must figure the number of index levels required to handle the specified number of data records.

At this point, ISAM supplies you with some information on the efficiency of your disk usage. The form of the information is the same as that used in building an indexed-sequential file.

Example - Maintaining an indexed-sequential file

```
@ISAM(RET)

*=CALNDR/M
Total records per Data block (64):
Empty records per data block (4):
Total entries per Index block (159):
Empty entries per index block (10):
Percentage of Data file to leave empty (0):
Percentage of Index file to leave empty (0):
Maximum number of records file can become (1000):

[ISMLOV 1 Level of index ]
[ISMNDR 1 Data record ]

[ISMWSD Wasted 0. words of 1408.  in the Data file.]
[ISMLDE One logical Data block equals 11. physical disk blocks.]
[ISMWSI Wasted 2. words of 640. 0.3% wasted space in the Index
file.]
[ISMLIE One logical Index block equals 5. physical disk blocks.]
[ISMIBS LIBOL's I/O buffer will require 3712. words (8. Pages) of
core.]

*
```

7.1.3 Packing An Indexed-Sequential File

Packing an indexed-sequential file is the reverse of building one. An indexed-sequential file is copied into a sequential file in the order specified by the index. This option is used primarily to compact an indexed-sequential file for backup storage, although the resulting sequential file can be treated as any other sequential file. You should always use ISAM to back up your indexed-sequential files. You can pack the files to tape, then rebuild on disk, or you can maintain the file before packing and copying it to tape (or other backup medium). In either case, you have cleaned up the file you are saving for backup and streamlined the actual master file as well. The command string for the packing option of ISAM is as follows:

```
     .R ISAM(RET)  for users of TOPS-10

                or

     @ISAM(RET)  for users of TOPS-20

     *devl:seqfil.ext[ppnl]=dev2:indfil.ext[ppn2] /P(RET)
```

where:

devl and dev2 are the devices on which the sequential file is to be stored and the index file resides, respectively. The input file must be on disk. If neither device is specified, DSK is assumed.

seqfil.ext is the name and extension of the output sequential file. If the name is omitted, the name of the input file is assumed. If the extension is omitted, .SEQ is assumed.

indfil.ext is the name and extension of the index file of the indexed-sequential file. The name must be specified, but the extension can be omitted. If the extension is omitted, no extension is assumed.

[ppnl] [ppn2] are directories for the new sequential file and the old index file, respectively. If either is omitted, the directory of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/P is the switch signifying that the packing option is being requested. It must be included.

If the output file specification is omitted, the equal sign (=) can be omitted.

After the command string has been processed, ISAM asks the following questions.

```
     Mode of output file:
```

Specify SIXBIT (or S), ASCII (or A), F, V, or ST according to the mode in which the sequential file is to be recorded. V is variable-length EBCDIC, and F is fixed-length EBCDIC, and ST is STANDARD-ASCII.

```
     Records per output block:
```

Give the blocking factor that you want for the sequential file (i.e., the number of records per logical block). If the file is to be unblocked, you answer 0.

Example - Packing an indexed-sequential file

```
.R ISAM
*MTA2:TEST.SEQ=TEST.IDX/P
Mode of output file: SIXBIT
Records per output block: 0
```

If you are packing the data from the indexed file into a sequential ASCII file, you can specify the advancing mode you want to be used with the /A switch (/AD and /ADV are also legal). The format of the command string to use with the /A switch is as follows:

```
sequential-file/A:mode=index-file/P
```

The arguments the /A switch takes are:

Mode	**Meaning**
6[8]	use COBOL-68 default (BEFORE)
7[4]	use COBOL-74 default (AFTER)
A[FTER]	use COBOL-74 default
B[EFORE]	use COBOL-68 default

If you do not specify the advancing mode when you pack an ASCII file, your file is written using the COBOL-68 default, which is BEFORE ADVANCING.

7.1.4 Ignoring Errors

When packing an indexed-sequential file into a sequential file, you can include the /I switch in the command string to force ISAM to ignore certain fatal errors. This switch causes ISAM to try to recover as much data as possible from a damaged indexed-sequential file.

Including the /I switch in the command string to ISAM causes the program to make nonfatal those errors that concern duplicate keys or keys out of order. The messages for these errors are preceded by a percent sign (%) rather than a question mark (?) so that ISAM continues the packing operation. The /I switch can be used only with the /P switch. It cannot be used alone.

The command string to use with the /I and /P switches is as follows:

```
.R ISAM(RET)   for users of TOPS-10

          or

@ISAM(RET)   for users of TOPS-20

*dev1:seqfil.ext[ppn1]=dev2:indfil.ext[ppn2]/P/I(RET)
```

where:

dev1 and dev2 are the devices on which the sequential and index files reside, respectively. The input file must be on disk. If neither device is specified, DSK is assumed.

seqfil.ext is the name and extension of the output sequential file. If the name is omitted, the name of the input file is assumed. If the extension is omitted, .SEQ is assumed.

indfil.ext is the name and extension of the index file of the indexed-sequential file. The name must be specified, but the extension can be omitted. If the extension is omitted, no extension is assumed.

[ppn1], [ppn2] are directories for the new sequential file and the old index file, respectively. If either is omitted, the directory of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/P is the switch signifying that the packing option is being requested. It must be included.

/I is the switch signifying that some fatal errors are to be ignored. It can be included only with the /P switch.

The equal sign (=) can be omitted if the output file specification is omitted.

7.1.5 Reading And Writing Magnetic Tape Labels

When you are building or packing an indexed-sequential file, you can include the /L switch to cause ISAM to read or write labels on magnetic tape. The /L switch, when used with the /B switch, causes ISAM to read COBOL standard tape labels on the input magnetic tape. When used with the /P switch, the /L switch causes ISAM to write standard tape labels on the output magnetic tape. The /L switch can only be used on magnetic tape files whose recording mode is not F or V.

NOTE

> The tape labels handled by ISAM are COBOL tape labels, not ANSI labels. COBOL labels are processed by the object-time system, not the operating system.

The command string when using the /L switch with the /B switch is as follows:

```
.R ISAM(RET)  for users of TOPS-10

          or

@ISAM(RET)  for users of TOPS-20

*dev1:indfil.ext[ppn],dev2:datfil.ext=MTAn:seqfil.ext/B/L(RET)
```

where:

devl, dev2, and MTAn are the devices for the index, data, and input sequential file. devl and dev2 must be disk devices. The default disk for devl and dev2 is DSK.

indfil.ext is the name and extension of the index file. If the filename is not specified, the name of the input file is assumed. If the extension is omitted, .IDX is assumed.

datfil.ext is the name and extension of the indexed data file. If the filename is omitted, the name of the index file is assumed. If the extension is omitted, .IDA is assumed.

seqfil.ext is the name and extension of the input sequential file. This filename must be specified, but the extension can be omitted. If it is omitted, .SEQ is assumed.

[ppn] specifies the directory for the index file. If it is omitted, the directory of the logged-in user is assumed. The data file must reside in the same directory as the index file. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/B is the switch signifying that ISAM is used to build an indexed-sequential file. If the switch is omitted from the command string, /B is assumed.

/L is the switch signifying that ISAM reads standard tape labels. It must be included.

The equal sign (=) can be omitted if the file specifications for the output files are also omitted.

The command string when using the /L switch with the /P switch is as follows:

```
.R ISAM(RET)  for users of TOPS-10

          or

@ISAM(RET)  for users of TOPS-20

*MTAn:seqfil.ext=devl:indfil.ext[ppn]/P/L(RET)
```

where:

MTAn: and dev1 are the devices on which the sequential file is to be stored and the index file resides, respectively. The input file must be on disk. If the name of dev1 is not specified, DSK is assumed.

seqfil.ext is the name and extension of the output sequential file. The name and extension can both be omitted because filenames are not used on magnetic tape.

indfil.ext is the name and extension of the index file of the indexed-sequential file. The name must be specified, but the extension can be omitted. If the extension is omitted, no extension is assumed.

[ppn] is a directory for the old index file. If it is omitted, the directory of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/P is the switch signifying that the packing option is being requested. It must be included.

/L is the switch signifying that ISAM writes standard tape labels. It must be included.

7.1.6 Renaming An Indexed-Sequential File

Since indexed-sequential files actually consist of two different files, you cannot rename a particular ISAM file with a single monitor-level command. To allow you to rename ISAM files with greater ease, the ISAM program has the /R switch. The form of the command string you must supply to ISAM is as follows:

```
.R ISAM(RET)  for users of TOPS-10

          or

@ISAM(RET)  for users of TOPS-20

*dev1:indfil.ext[ppn1],dev2:datfil.ext=infil.ext[ppn2]/R(RET)
```

where:

dev1, and dev2, are disk devices on which the files are stored. If any of the devices is omitted, DSK is assumed.

indfil.ext is the name and extension of the new index file. If the name is omitted, the name of the input file is assumed. If the extension is omitted, .IDX is assumed.

datfil.ext is the name and extension of the new indexed data file. If the name is omitted, the name of the new index file is assumed. If the extension is omitted, .IDA is assumed.

infil.ext is the name and extension of the index file of the old indexed-sequential file. The name of the file must be specified, but the extension can be omitted. No extension is assumed if the extension is omitted.

[ppn1], [ppn2] specify directories for the new index file and the old index file, respectively. If either is omitted, the directory of the logged-in user is assumed. The new data file must reside in the same directory as the new index file. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

/R is the switch indicating that the file is to be renamed. The switch must be specified.

The equal sign (=) can be omitted if the output file specification is omitted.

Example - Renaming an indexed-sequential file

```
@DIRECTORY CALNDR

   PS:<DUPREE>
 CALNDR.IDA.1
   .IDX.1

 Total of 7 pages in 2 files
@ISAM
*CLDR=CALNDR/R

*C
@DIRECTORY CLDR

   PS:<DUPREE>
 CLDR.IDA.1
   .IDX.1

 Total of 7 pages in 2 files
@
```

7.1.7 Checking An Indexed-Sequential File

The ISAM program can check the format of an indexed-sequential file to make sure there are no detectable errors. This includes duplicate keys, bad key ordering, version number discrepancies, and record length discrepancies.

When you use the /C switch, ISAM reads the entire file, printing error messages when it encounters errors as it does when you use the /I switch. That is, the messages are printed as warnings rather than fatal errors; the prefix character for these messages is % instead of ?. Since ISAM does not produce an output file, you need not specify an output file specification. The command string you should use to get ISAM to check your indexed-sequential file is as follows:

```
.R ISAM(RET)  for users of TOPS-10

             or

@ISAM(RET)  for users of TOPS-20

*idxfil/C
```

Example

```
*CALNDR/C

%ISMKOO    keys are out of order
   00000009
   is after
   00000010
   Data block 7. version number 21.


*
```

7.1.8 Producing Blocking Data With ISAM

The ISAM program can help you to build indexed-sequential files more efficiently when you use the /S switch to produce blocking factors and statistics. ISAM cannot make the decision of the best blocking factor for you, because you must balance the best blocking factor in terms of disk space against the amount of main memory the file needs to be processed. But ISAM can help you by doing most of the simple calculations. The statistics ISAM can calculate include:

- Percentage of wasted space in the file for each blocking factor
- Number of physical blocks in each logical block for each blocking factor
- Size of logical block in core words for each blocking factor

Example - Producing Blocking Data

```
@ISAM
*=CALNDR/B/S
Mode of input file: A
Mode of data file: A
Maximum record size: 102
Key descriptor: 1.8
Records per input block: 0
Total records per Data block:
     Records  Disk     Wasted   Core
     /block   Blocks   space    (wds)
       5        1       14.1%    128
       11       2       5.5%     256
       17       3       2.6%     384
       23       4       1.2%     512
       29       5       0.3%     640
       34       6       2.6%     768
       40       7       1.8%     896
       46       8       1.2%     1024
       52       9       0.7%     1152
       58       10      0.3%     1280
       64       11      0.0%     1408
```

```
(Recommended = 64):
Empty records per data block: 4
Total entries per Index block:
     Records  Disk     Wasted   Core
     /block   Blocks   space    (wds)
       31       1       1.6%     128
       63       2       0.8%     256
       95       3       0.5%     384
       127      4       0.4%     512
       159      5       0.3%     640
(Recommended = 159):
Empty entries per index block: 10
Percentage of Data file to leave empty: 10
Percentage of Index file to leave empty: 10
Maximum number of records file can become: 1000

[ISMLOV        1 Level of index ]
[ISMNDR        4 Data records ]

[ISMWSD        Wasted 0. words of 1408.  in the Data file.]
[ISMLDE        One logical Data block equals 11. physical disk
blocks.]
[ISMWSI        Wasted 2. words of 640. 0.3% wasted space in the
Index file.]
[ISMLIE        One logical Index block equals 5. physical disk
blocks.]
[ISMIBS        LIBOL's I/O buffer will require 3712. words (8.
Pages) of core.]

*^C
@
```

7.1.9 Indirect Commands

The ISAM program accepts command strings and dialogue responses from indirect command files.

The command string to direct ISAM to read an indirect command file is:

```
.R ISAM(RET)   for users of TOPS-10

            or

@ISAM(RET)   for users of TOPS-20

*@dev:cmdfil.ext[ppn](RET)
```

where:

@ indicates that this is an indirect command file.

dev is the device on which the command file is stored. If it is omitted, DSK is assumed.

cmdfil.ext is the name and extension of the command file. The name must be specified. If you omit the extension, .CMD is assumed.

[ppn] is the directory in which the command file is stored. If it is omitted, the directory of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

After ISAM reads the command string, it reads the command file and performs the processing specified within it. The command file must contain the complete command string and all dialogue responses for a single ISAM operation exactly as they would be typed if you were giving them directly to the ISAM program. Nothing else can be present in the command file.

7.1.10 **Using Indexed-Sequential Files**

Indexed-sequential files can be read and written, and individual records within them can be rewritten or deleted. You can perform any actions on the records in an indexed-sequential file by specifying the desired record key in the RECORD KEY field. Remember that COBOL-68 supports single-key ISAM files only. To use an indexed-sequential file, the following statements are employed:

```
     ENVIRONMENT DIVISION.
     INPUT-OUTPUT SECTION.
     FILE-CONTROL.
1.        SELECT ISAM-FILE ASSIGN TO DSK
2.        ACCESS MODE IS INDEXED
3.        SYMBOLIC KEY IS ISAM-SYM-KEY
4.        RECORD KEY IS ISAM-RECORD-KEY.
                .
                .
     DATA DIVISION.
     FILE SECTION.
     FD ISAM-FILE
5.        BLOCK CONTAINS 13 RECORDS
6.        VALUE OF IDENTIFICATION IS "ISAMFLIDX".
     01 ISAM-RECORD.
          02 FILLER PIC X(12).
4.        02 ISAM-RECORD-KEY PIC X(3).
          02 FILLER PIC X(75).
                .
                .
     WORKING-STORAGE SECTION.
3.   01 ISAM-SYM-KEY PIC S(3).
                .
     PROCEDURE DIVISION.
     BEGIN.
          OPEN INPUT-OUTPUT ISAM-FILE.
                .
                .
7.        READ ISAM-FILE, INVALID KEY GO TO ERRPROC.
                .
                .
8.        WRITE ISAM-RECORD, INVALID KEY GO TO ERRPROC.
                .
                .
9.        DELETE ISAM-RECORD, INVALID KEY GO TO ERRPROC.
                .
                .
```

```
10.       REWRITE ISAM-RECORD, INVALID KEY GO TO ERRPROC.
               .
               .
          MOVE LOW-VALUES TO ISAM-SYM-KEY.
11.       READ ISAM-FILE, INVALID KEY GO TO ENDFILE.
```

The notes in the following list are keyed to the numbers to the left of the lines in the preceding program.

1. The indexed-sequential file must reside on disk.

2. The ACCESS MODE clause is required if you wish to access the file in random fashion, since the ACCESS MODE defaults to sequential.

3. The SYMBOLIC KEY clause is required in the Environment Division and the data item named must be defined in the Working-Storage Section of the Data Division. The SYMBOLIC KEY and the RECORD KEY must have the same size and usage, but they do not need to have the same level number, and the fact that one of them is a group item does not mean that the other must also be a group item.

4. The RECORD KEY clause is required in the Environment Division. It refers to the data-item designated as the record key which appears in the Data Division within the FD.

5. An indexed-sequential file must be blocked.

6. The VALUE OF IDENTIFICATION clause is required. It designates the filename and extension of the index file rather than that of the data file. The name of the related data file is stored within the index file. The VALUE OF IDENTIFICATION must be specified because the name of the file must be present at initialization time so that the buffer and storage space can be allocated.

7. The READ statement reads the indexed-sequential file to find the record whose key as written on the file matches the record key. If no match is found, the INVALID KEY path is taken.

8. The WRITE statement writes the record that has a key that matches the record key. If the record whose key matches the record key is already in the file, the INVALID KEY path is taken.

9. The DELETE statement causes a search to be made of the file to find the record whose key matches the record key. When the record is found, it is deleted. If the record is not found, the INVALID KEY path is taken.

10. The REWRITE statement causes searching of the file to find the record whose key matches the record key. When the record is found, it is replaced with the contents of the record specified in the REWRITE statement. If the record is not found in the file, the INVALID KEY path is taken.

11. This shows the method used to read an indexed-sequential file sequentially. First, LOW-VALUES is moved to the SYMBOLIC KEY. Then a READ is issued, which causes the record following the last record accessed to be read. Thus, if the first I/O operation you execute is a READ with LOW-VALUES in the SYMBOLIC KEY, you read the first record in the file. You could then proceed to issue more READs and you would be reading the indexed file sequentially. This procedure works with all I/O verbs, not just with READ.

7.2 LIBARY - PROGRAM TO CREATE AND MAINTAIN SOURCE LIBRARIES

LIBARY provides a facility for creating or maintaining COBOL library files on disk or DECtape (TOPS-10 only). Library files contain COBOL source-language text organized into statement groups. Specifically, the LIBARY program has the capability of adding source-language text to the library file, replacing and/or deleting statement groups, and providing a listing of the file. It allows you to specify those data descriptions or procedures used in many programs and to place them in a common file for use by the COBOL compiler. The statement groups in the library file are included in a COBOL program through the use of the COPY verb. (See Section 1.4.1 for information on the COPY statement.)

7.2.1 Library File Format

A library file is a collection of COBOL source-language statement groups, each identified by a unique 1- to 8-character library-entry-name. The library file must be on a directory device. Each statement group is a set of ordinary COBOL language statements conforming to the use of the COPY verb. The statement groups are kept in alphabetic order according to their library names. The maximum number of statement groups that can appear in a library is 3869.

The library file is in a binary format that is recognizable only by LIBARY and the COBOL compiler. You, however, need not concern yourself with the format of the actual entries in the file. You enter them as ASCII text; LIBARY stores them in the appropriate format automatically.

7.2.2 Invoking The Library Utility

To invoke the library utility program, enter R LIBARY in response to the TOPS-10 prompt (.) or LIBARY in response to the TOPS-20 prompt (@). That is:

```
.R LIBARY(RET)  for users of TOPS-10

            or

@LIBARY(RET)  for users of TOPS-20
```

When LIBARY is ready to process commands, it issues an asterisk prompting character and waits for you to enter a file specification command line. The file specification command line identifies the library files being either created or used as input. It also identifies the listing file if a listing is required. The file specification command line has the following general format:

```
*output-library,listing=input-library(RET)
```

where:

output-library	is the file specification for the library file being generated.
listing	is the file specification for the file that is to receive the output listing (See Figure 10-1, Sample LIBARY listing.)
input-library	is the file specification for the library file being used as input.

Each file specification has the following format:

```
dev:filename.ext[ppn]/sw
```

where:

dev:	is the logical device name for the unit on which the desired file is mounted. The default assignment is DSK:.
filename	is the name of the file consisting of from one to six SIXBIT characters. Filename must be specified for at least one library file.
.ext	is the filename extension consisting of a period followed by zero to three characters. It is used to indicate the type of information in the file.
[ppn]	is the directory area in which the file is stored. The directory specification, enclosed in brackets, contains the project-programmer number of the file's owner. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative which is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.
/sw	is one ASCII character preceded by a slash specifying a LIBARY switch option. (See Section 7.2.4, LIBARY Switches.)

There are three main forms to the LIBARY command string. One form allows you to read an existing library and make changes, additions, deletions, and extractions as you wish. The second form simply produces a listing of the contents of the exiting library. The third allows you to create a new library by typing in the text with which you wish to load it.

To make changes, additions, and so on, to your existing library, you use the following command form (note that the listing file contains only the changes to your existing library, not a listing of the entire library):

```
    *output-library,listing-file=input-library(RET)
```

If you wish to produce a listing of your input library but do not wish to make any changes to the library, use the following form:

```
    *listing-file=input-library/L(RET)
```

To create a new library by typing in text, use the following form of the command:

```
    *output-library,listing-file=(RET)
```

After you have invoked LIBARY and given it a file specification command line, it automatically creates a scratch file to contain the output file generated by the LIBARY run. When you are through working on your library file and enter the END command (See Section 7.2.6.2, LIBARY Directing Commands), LIBARY renames the scratch file with the proper output name (after any necessary renaming of the input file).

If an error occurs causing the execution of LIBARY to be aborted, the input file, if specified, is unchanged and the scratch file is deleted. If the error occurred after the input file has been renamed, the original input file has an extension of .BAK.

7.2.3 Command String Defaults

The following default values are assumed by LIBARY if any part of any file specification is omitted:

1. If any device is not specified, DSK is assumed.
2. If the file specification for the listing file is omitted, no listing is produced.
3. If the name of the listing file is omitted, the name of the input file is assumed.
4. If the extension of the listing file is omitted, .LST is assumed.
5. If the file specification for the output file is omitted, it is assumed that there is no output file to be produced.

NOTE

If you are omitting the output file because you want to run LIBARY to obtain a listing only, the listing file specification, the input file specification, and the /L switch must be specified.

6. If the name of the output file is omitted, the name of the input file is assumed. In this case (as well as when you specify that the output file is to be named the same as the input file), the input file is written out with the extension .BAK.

7. If the extension of the output file is omitted, .LIB is assumed.

8. If the file specification of the input file is omitted, it is assumed that there is no input file and that a library is being created. Thus, only commands for insertion can be used.

9. The filename for the input file cannot be omitted if the file specification is present.

10. If the extension of the input file is omitted, .LIB is assumed.

11. If any project-programmer number is omitted, it is assumed to be that of the logged-in user. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative which is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

12. If the input and output files have the same name and extension, and are both on disk, the extension of the input file is changed to .BAK at the completion of the operation.

7.2.4 LIBARY Switches

The following switches can be included in the command string to LIBARY:

/D	List on your terminal all of the library-entry-names contained on the input library file.
/H	List on your terminal all of the commands available with LIBARY.
/L	Create only a listing file of the entire input library. The output file specification must be omitted.
/S	Put the input statement group into standard card format.
/W	Rewind (for magnetic tape only).

/Z	Clear an output directory (for DECtape only).

7.2.5 Running LIBARY

Running LIBARY consists of specifying commands in response to the LIBARY asterisk prompting character (*). LIBARY provides you with the means to optionally create new library files and insert or delete statement groups into an existing file. Each command causes LIBARY to move forward in the file. Because LIBARY cannot move backward in the file, you should plan your interaction with LIBARY so that you create or modify your files in alphabetical order by statement group. This keeps you from having to restart LIBARY and reprocess your file. Thus, if you have a library entry that is called CLDESC and another called FLSTAT, you should be sure to deal with CLDESC first and do everything you wish to accomplish at once, if possible, before going on to FLSTAT.

7.2.6 LIBARY Commands

The following sections describe the commands available with LIBARY. LIBARY commands are divided into two classes of commands:

- Group mode (See Section 7.2.6.1)
- LIBARY directing (See Section 7.2.6.2)

These commands can be abbreviated as long as you supply a unique abbreviation.

NOTE

For the remainder of this chapter, the words "line number" refer to the line numbers generated by a system standard editor (such as SOS or EDIT); the words "COBOL line number" refer to the conventional line numbers as described in Section 1.3, Source Program Format.

7.2.6.1 Group Mode Commands

- Group mode commands allow you to insert, replace, extract, and delete entire statement groups. The group mode commands are:

DELETE library-entry-name

Delete the statement group identified by library-entry-name from the library file. The library-entry-name itself is also deleted. LIBARY moves forward through the input library file. It copies each statement it finds onto the output file until it encounters the library entry specified by library-entry-name. When library-entry-name is reached, LIBARY positions itself at the next sequential library entry and waits for another command.

EXTRACT library-entry-name,file-specification

Extract the complete library entry specified by library-entry-name from the input library file and generate a new file named filename. LIBARY searches the input library file for the library entry specified by library-entry-name. When library-entry-name is found, it creates a file or overwrites an existing file with the attributes specified by filename and copies the library entry onto it. The input library file remains unchanged.

INSERT library-entry-name,file-specification

Insert the statement group contained on the file specified by filename into the output library file. The statement group is inserted alphabetically according to the name specified by library-entry-name. The file specified by filename must be an ASCII file. LIBARY assumes that the entire file is to be inserted under library-entry-name. If you want to insert many entries, you must create a separate file for each and execute a separate INSERT command for each. If there are line numbers in the file, they are included when the file is merged. If there are no line numbers, LIBARY generates them starting with 10 and incrementing by 10. If the library entry being inserted contains COBOL line numbers, the /S switch must be specified. (See Section 7.2.4, LIBARY Switches.)

REPLACE library-entry-name, file-specification

Replace the library entry identified by library-entry-name with the statement group contained on the file specified by filename. The file specified by filename must be an ASCII file. LIBARY assumes that the entire file is to replace the statements currently associated with library-entry-name. If you want to replace many library entries, you must create a separate file for each, and execute a separate REPLACE command for each. If there are line numbers in the file, they are included. If there are no line numbers, LIBARY generates them starting with 10 and incrementing by 10. The /S switch must be specified for files having COBOL line numbers. (See Section 7.2.4, LIBARY Switches.)

7.2.6.2 **LIBRARY-Directing Commands** - LIBRARY-directing commands allow you to end or restart library processing. The LIBARY-directing commands are:

END

Copy any remaining statement groups from the input to the output file, close both the input and output files, and rename the input file with the extension .BAK, if necessary.

NOTE

If you neglect to use the END command and attempt to leave LIBARY by hitting CTRL/C, you abort your LIBARY session, and your output library and listing file is not written out.

RESTART

Copy any remaining statement groups from the input to the output files, close both the input and output files, rename the input file with the extension .BAK, and reopen the output file as the new input. Any changes made prior to issuing the RESTART command are in the new input file.

7.2.6.3 **Example of Command Usage** - The following example shows the use of LIBARY commands. Suppose a library on disk contains the routines PAYCOMP, FIND-MP, and MP-DESCR, and you wish to do the following:

1. Insert a new routine called JOB-DESC
2. Correct MP-DESCR
3. Delete PAYCOMP

These tasks must be undertaken in this order because LIBARY deals with code units in alphabetic order only. The MP-DESCR routine contains the following source statements:

```
000010     LABEL RECORDS ARE OMITTED
000020     DATA RECORD IS MP-RECORD.
```

The dialogue at the terminal might appear as follows:

```
.R LIBARY
*LIBARY.NEW=LIBARY.OLD
*INSERT JOB-DESC, JOBDES.TXT
*REPLACE MP-DESCR, MPDESC.TXT
*DELETE         PAYCOMP
*END
```

The file LIBARY.NEW now contains the following:

1. FIND-MP
2. JOB-DESC
3. MP-DESCR

To insert one or more files in a library, you can issue the following commands to LIBARY.

```
.R LIBARY
*ALIB,ALIB=
*INSERT AFIL,AFIL
*INSERT BFIL,BFIL
*END

*^C
```

The file ALIB.LIB contains two statement groups (AFIL and BFIL) and the file ALIB.LST contains the following information.

```
A F I L            COBOL LIBRARY       01-DEC-78        09:52

000010     DISPLAY "A".


B F I L            COBOL LIBRARY       01-DEC-78        09:52

000010     DISPLAY "B".
```

7.3 COBDDT - PROGRAM FOR DEBUGGING COBOL PROGRAMS

COBDDT is an interactive program that is used to debug COBOL programs at run-time. With COBDDT, you can:

1. Change the contents of a data-name
2. Set up to 20 breakpoints in a program
3. Continue from a breakpoint to any other breakpoint
4. Display the contents of a data-name
5. Trace paragraphs and sections
6. Obtain a histogram of paragraphs executed to show program behavior
7. Interrupt a running program

7.3.1 Loading And Starting COBDDT

To run COBDDT, you must first compile the source program. You then load and start the compiled program with COBDDT.

NOTE

Using the /P switch with the COMPILE command suppresses your symbols that are used by COBDDT. Therefore, you must not use the /P switch when compiling your program, if you wish to use COBDDT. However, it is possible to load some programs compiled with the /P switch with some compiled without the /P switch, though those compiled with the /P switch cannot be debugged.

You can load the compiled source program with either the monitor command LOAD or direct commands to LINK. In both cases, LINK loads your symbols along with the program.

After loading the compiled source program, you issue the monitor command START to start the program. You can also issue the monitor command DEBUG to load and start COBDDT with your COBOL program. If you use the DEBUG command, you can specify the file to be debugged by any of the following: the name of the source file, the name of the binary relocatable file, or merely the name of the file without the extension. However, if the extension of the source file is something other than .CBL, you must use the /COBOL switch with the DEBUG command. Otherwise the file is not recognized as a COBOL file. When you load COBDDT with your program, only COBDDT is started; the program itself is not started.

The three methods of loading and starting are shown below. Although all system prompts shown are for TOPS-10, you can use the same syntax on TOPS-20. If you are using TOPS-20, you do not have to specify the /"LOCALS" switch, as TOPS-20 loads local symbols by default.

1.
```
.LOAD %"LOCALS" file spec, SYS:COBDDT
.START
```

2.
```
.DEBUG file spec [/COBOL]
```

3.
```
.R LINK
*/LOCALS file spec, SYS:COBDDT /GO
.START
```

NOTE

On TOPS-20, you do not need to specify load local symbols as this is the default mode.

When the program is started with the START command, COBDDT is entered. This is shown by the message:

```
[Starting COBOL DDT]
COBDDT>
```

You can now issue any COBDDT command (described below). Users of TOPS-20 can get a list of the available commands by typing a question mark. The TOPS-20 version of COBDDT also allows you to use recognition on the commands. If you want to run your program at this time, enter the PROCEED command. This causes your program to run to completion or until a fatal error is encountered. If an error is encountered that would normally cause abortion of execution, COBDDT is entered automatically and the message:

```
?ENTERING COBDDT from:     <paragraph-name>
```

gives the name of the paragraph in which the error occurred. COBDDT can then be used to check data values at the time of the failure. The program cannot proceed after COBDDT has been entered due to an error.

If the COBOL program is in a loop and is not reaching a breakpoint, you can enter COBDDT by typing CTRL/C two times followed by typing the REENTER command. For example:

```
^C^C
REENTER
```

This causes COBDDT to display the following message:

```
Do you want to enter COBDDT (Y or N)
```

If you enter Y, the execution of the object program is resumed where it was interrupted and COBDDT is entered at the next TRACE entry in the program. If you enter N, however, your COBOL program is reentered at its original address.

7.3.2 COBDDT Commands

The commands to COBDDT are described below. Other than for the STOP command, you need only type the first letter of each command for COBDDT to recognize the command. For the STOP command, however, you must type the entire command. Data-names and section-names need not be typed in full as long as each name or portion of the name is unique in the program. Paragraph-names can be qualified by section-names, and data-names can be qualified by higher-level data-names or subscript values or both. The subscripts for a qualified data-name must appear immediately after the first data-name. Subscripts must be numeric integers. Section-names and data-names cannot be qualified by program-names because COBDDT uses the names in the program specified in the MODULE command.

NOTE to TOPS-20 Users

COBDDT uses the COMND JSYS to do its command parsing. You must be aware that if a command line ends with a "-" (hyphen), the COMND JSYS deletes the "-" from the command line and continues parsing on the next line.

If you wish to include a "-" as the last character on a command line, you should type a space following the "-". For example,

```
COBDDT>DISPLAY FILE-<SPACE><RET>
```

ACCEPT

The ACCEPT command allows you to change the contents of a data item. The new contents of the data item are typed on the next line. The ACCEPT command has the format:

```
ACCEPT
ACCEPT data-name
```

If the data-name is not specified, the last name specified in a DISPLAY or another ACCEPT command is assumed.

Example:

```
COBDDT>ACCEPT VAR1
16.25

COBDDT>
```

BREAK

The BREAK command sets a breakpoint (or pause) at the beginning of the specified paragraph or section name. The BREAK command has the format:

```
BREAK paragraph-name
BREAK section-name
```

Up to 20 breakpoints can be set in a program. unless the UNPROTECT command is given.

Breakpoints can be set in nonresident COBOL segments, whether or not the segment is in memory. If more than one module is in memory, the name of the module in which the break occurred is typed with the paragraph and section names.

You can set breakpoints in LINK overlays, but all breaks in the overlay are cleared when the overlay is overlaid or cancelled. To set breakpoints in LINK overlays, you must use the OVERLAY command to specify OVERLAY ON. If you do not specify the OVERLAY ON command, the program executes through the overlay before you can set a breakpoint. This is because you cannot set a breakpoint in an overlay unless the overlay is in memory.

Example:

```
COBDDT>BREAK PAR1

COBDDT>
```

CLEAR

The CLEAR command removes the breakpoint at a specified paragraph. The CLEAR command has the format:

```
CLEAR paragraph-name
CLEAR
```

If the paragraph-name is not specified, all breakpoints that have been set in the program are removed.

Example:

```
COBDDT>CLEAR PAR1

COBDDT>
```

DDT

The COBDDT DDT command allows you to enter regular DDT, the assembly language debugger. COBDDT can supply only certain types of data; the use of the DDT command enables you to look at the data areas or procedure areas of the object program. This allows you to change the compiled code or to put breakpoints in the middle of a paragraph. If COBDDT or LIBOL have been linked with symbols, you can use the DDT command to look at these as well. To use the assembly language debugger, you must first use the LOCATE command or an assembly listing to obtain the addresses of the areas that you want to look at. Once you have these addresses, you can use the DDT command to look at these areas. The DDT command has the format:

```
DDT
```

COBDDT responds to the DDT command by telling you how to exit from the assembly language debugger back to COBDDT. To get back to COBDDT from the assembly language debugger, you type POPJ 17,$X, where the "$" is an ESCape (ESC).

COBDDT tries to load DDT into memory if it is not already there.

This example shows the use of the DDT command on TOPS-10. Although the system prompt differs on TOPS-20, the use of the command is the same on both systems.

Example:

```
.RUN PRGRM

STARTING COBOL DDT

COBDDT>DDT
[Return from DDT by typing "POPJ 17,$X"]
DDT
```

DISPLAY

The DISPLAY command causes the contents of a data item to be displayed on your terminal. The DISPLAY command has the format:

```
DISPLAY
DISPLAY data-name
```

If no data-name is specified, COBDDT uses the last data-name specified in an ACCEPT or DISPLAY command.

Example:

```
COBDDT>DISPLAY ALPHA
0

COBDDT>
```

GO

The GO command causes the program to resume execution of the specified procedure name. The GO command has the format:

```
GO procedure-name
```

The procedure name must be in a module that is currently loaded into core. Execution of the program begins at the designated procedure name immediately after the command is typed.

The procedure name that you specify can be in another module, if that module is in memory. However, the GO command does not set up a return for the EXIT PROGRAM statement, nor does it provide addresses for LINKAGE SECTION items.

The GO command also does not alter the existing stack of PERFORM exits or subprogram exits. If an error is detected in using these return mechanisms following the GO command, control is returned to COBDDT, but the PROCEED and GO commands are disabled. Therefore further execution of the object program is not possible.

Example:

```
COBDDT>GO PARA1
BREAK AT <<PARA4>>

COBDDT>
```

LOCATE

The LOCATE command causes the object-time address of a procedure name or a data item to be typed. The LOCATE command has the format:

```
LOCATE procedure-name

LOCATE data-item
```

If the specified data-item does not start on a word boundary in memory, the bit displacement of the data-item is also displayed.

Example:

```
COBDDT>LOCATE PARA1
401057

COBDDT>
```

MODULE

The MODULE command causes COBDDT to look for data names and procedure names in the specified program. The MODULE command has the format:

```
MODULE [program-name]
```

If the name is omitted, COBDDT types the name of the current module followed by the names of all modules currently in memory.

Normally, within a run unit containing more than one program, COBDDT searches for data names and procedure names in the current program. The MODULE command changes the program in which the search takes place. All subsequent searches for data names and procedure names are within the specified program until another MODULE command is issued. If the current module is cancelled or overlaid, the main program becomes the current module.

Example:

```
COBDDT>MODULE

CURRENT MODULE:  MYPROG

COBDDT>
```

NEXT

The NEXT command causes the contents of a data item to be displayed on your terminal. The NEXT command uses the variable name and the subscript values given for the last ACCEPT, DISPLAY, or NEXT command and adds the numeric value of the signed integer to the rightmost subscript value in the subscript list. The NEXT command has the format:

```
NEXT
NEXT signed integer
```

If the signed integer is omitted, a default of +1 is used. A signed integer can be any integer with plus, minus, or no leading sign. If you specify a subscript that is out of range, an error message is displayed.

Example:

```
COBDDT>NEXT 3
33

COBDDT>
```

OVERLAY

The OVERLAY command either causes a break when an overlay is entered or clears the breakpoint. The OVERLAY command has the format:

```
OVERLAY ON
OVERLAY OFF
```

OVERLAY ON causes COBDDT to break the first time that a LINK overlay is entered each time it is brought into memory. The break only occurs once for each time the overlay is brought into memory. COBDDT types the following message when the break occurs:

```
BREAK UPON ENTRY TO name
```

where name is the name of the entry point. Following the message, COBDDT types the name of the current module and a list of the modules currently in memory.

OVERLAY OFF causes COBDDT not to break when a LINK overlay is entered and not to type the information described above. OVERLAY OFF is the initial default.

PROCEED

The PROCEED command causes the program either to be started or to continue execution after a breakpoint caused it to pause. The PROCEED command has the format:

```
PROCEED
PROCEED n
```

After a PROCEED command is executed, the program runs either to completion or until another breakpoint is reached. If an integer is included with the command, the program runs until the n(th) occurrence of the preceding breakpoint has been reached. Thus PROCEED 1 is equivalent to PROCEED.

Example:

```
COBDDT>PROCEED 3
BREAK AT <<PARA3>>

COBDDT>
```

SHOW SYMBOLS

The SHOW SYMBOLS command prints on the terminal all symbols that match the given mask. The mask consists of letters and the special characters ?, %, and *. The asterisk (*) stands for any number of characters, including zero. The question mark (?) for TOPS-10 and the percent sign (%) for TOPS-20 represent exactly one character.

```
COBDDT>SHOW SYMBOLS *WRITE*

     FIRST-WRITE-PARA
     LAST-WRITE-PARA
     REWRITE-RECORD
     WRITE-RECORD

COBDDT>SHOW SYMBOLS %%WRITE*

     REWRITE-RECORD

COBDDT>
```

STEP

The STEP command causes your program to execute a specified number of steps, each step being a procedure name, section name, or a paragraph name. The default is a single step. The STEP command has the format:

```
STEP
STEP integer
```

If an integer is included with the command, the program runs until the n(th) occurrence of a step has been reached. When the STEP command has completed the specified number of steps, the program is interrupted, and control is returned to COBDDT. The following display then occurs:

```
                         ┌                ┐
                         │ procedure-name │
STEP AT                  │ program-name   │
                         └                ┘
EXIT PROGRAM
```

Modules that have been compiled with the /P switch are invisible to the STEP command. The entry point, the procedure names and the exit programs are not counted as steps.

Example:

```
COBDDT>STEP 2
BREAK AT <<PARA2>>

COBDDT>
```

STOP

The STOP command is equivalent to the COBOL STOP RUN statement. All files that are open are closed and program execution is terminated. The STOP command has the format:

```
STOP
```

You must type the word STOP in full. Typing only the first letter, S, initiates execution of the STEP command.

Example:

```
COBDDT>STOP
EXIT
```

TRACE

The TRACE command starts tracing, stops tracing, or traces backwards, depending on the form of the command. The TRACE command has the format:

```
TRACE ON
TRACE OFF
TRACE BACK
```

TRACE ON causes tracing of all paragraphs and sections as they are executed. Whenever a paragraph or section is entered, its name, enclosed in angle brackets (<>), is typed on your terminal.

For each depth of subprogram, COBDDT types an exclamation point (!) before each paragraph or section name. For each depth of a PERFORM statement, COBDDT also types an asterisk (*) before each paragraph or section name. The maximum length of the string printed is 35 characters. Note that the exclamation point and asterisk are printed for each depth of subprogram or PERFORM.

Example:

```
COBDDT>TRACE ON
!!*!**<PARA>

COBDDT>
```

When a LINK overlay is brought into memory, COBDDT types the names of any modules overlaid and the names of the modules in the new overlay. When a LINK overlay is cancelled, COBDDT types the names of the modules in that overlay.

TRACE OFF causes COBDDT to stop tracing procedures until either execution is terminated or another TRACE ON command is executed.

Example:

```
COBDDT>TRACE OFF

COBDDT>
```

TRACE BACK causes COBDDT to show the sequence of paragraphs and sections that were called to reach this program. When you specify the TRACE BACK command, the name of the currently activated program is displayed, followed by the sequence of programs that were called to reach this program.

Example:

```
COBDDT>TRACE BACK

IN PROGRAM [SAMPLE]

COBDDT>
```

UNPROTECT (TOPS-10 only)

The UNPROTECT command turns off write-protection for the high segment. This command must be typed before you put breakpoints (the BREAK command) in the high segment of the source code to be compiled with the /R switch.

WHERE

The WHERE command causes COBDDT to list the names of all paragraphs at which breakpoints were set. The WHERE command has the format:

```
WHERE
```

If more than one module is in memory, the module name is included with the paragraph name.

Example:

```
COBDDT>WHERE

PROGRAM STOPPED AT <<PARA1>>

BREAKPOINTS:
<<PARA1>>
<<PARA2>>
<<PARA3>>

17 UNUSED BREAKPOINTS

COBDDT>
```

7.3.3 Obtaining Histograms Of Program Behavior

The histogram facility in COBDDT allows you to obtain a report of the number of times each section and paragraph in your COBOL program was entered as well as the total amount of processor time and elapsed time spent in each section and paragraph. The commands for using this feature are described in the following sections.

Both words of the histogram commands can be shortened to their unique abbreviations. None of the commands can be abbreviated to just H; the first letter of the second word of the command must be present; for example, H I, H B, and H E are legal.

7.3.3.1 Initializing the Histogram Table - The HISTORY INITIALIZE command causes COBDDT to set up and initialize the histogram table in which are stored the statistics for the histogram. The form of this command is:

```
HISTORY INITIALIZE [filespec]['title']
```

The file specification is the device, filename, extension, and project-programmer number of the output histogram report (dev:file.ext[p,pn]). If the entire file specification is omitted, your terminal is assumed. If the device is omitted but the filename is included, DSK is assumed. If the extension is omitted, .HIS is assumed. If the project-programmer number is omitted, that of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

The title is the one that is printed as the second line of the histogram report. It must be enclosed in single quotation marks and can have a maximum length of 70 characters.

Once you specify a file specification and/or title, it becomes the default for any subsequent reports until explicitly changed.

It is not necessary to use this command, but it is advisable to do so if only a portion of the program's statistics are to be recorded. The table can also be reinitialized by means of the HISTORY INITIALIZE command to begin a new histogram.

7.3.3.2 **Starting the Histogram** - The HISTORY BEGIN command causes COBDDT to start gathering statistics for each section and paragraph entered after this command is issued. This command has the form:

```
HISTORY BEGIN [filespec]['title']
```

The file specification is the device, filename, extension, and project-programmer number of the output histogram report (dev:file.ext[p,pn]). If the entire file specification is omitted, your terminal is assumed. If the device is omitted but the filename is included, DSK is assumed. If the extension is omitted, .HIS is assumed. If the project-programmer number is omitted, that of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

The title is the one that is printed as the second line of the histogram report. It must be enclosed in single quotation marks and can have a maximum length of 70 characters.

Once you specify a file specification and/or title, it becomes the default for any subsequent reports until explicitly changed.

The HISTORY BEGIN command implies a HISTORY INITIALIZE command if one has not already been issued and if a histogram has not already been started. If a histogram already exists, HISTORY BEGIN adds data to that histogram. The statistics collected are:

- The number of times each paragraph or section is entered
- The CPU time spent within each paragraph or section
- The elapsed time spent within each paragraph or section
- The elapsed time and CPU time for overhead
- The unaccounted elapsed time and CPU time

7.3.3.3 **Stopping the Histogram** - The HISTORY END command causes COBDDT to stop gathering statistics for the histogram. This command has the form:

```
HISTORY END
```

If you wish to gather statistics throughout the entire execution of the program, you need not use the HISTORY END command. However, if you wish to stop gathering statistics for the histogram before the program finishes, you must set a breakpoint at the appropriate paragraph and, when the break occurs, use the HISTORY END command.

7.3.3.4 **Obtaining the Histogram Listing** - The HISTORY REPORT command causes COBDDT to list the available statistics in a report. This command has the form:

```
HISTORY REPORT [file specification]['title']
```

The file specification is the device, filename, extension, and project-programmer number of the output histogram report (dev:file.ext[p,pn]). If the entire file specification is omitted, your terminal is assumed. If the device is omitted but the name is included, DSK is assumed. If the extension is omitted, .HIS is assumed. If the project-programmer number is omitted, that of the logged-in user is assumed. Users of TOPS-20 who wish to specify a directory other than the default can run the TRANSLATE program to determine the correct project-programmer number. (See the TOPS-20 User's Guide for information on how to do this.) For an alternative that is generally more useful, see Appendix C, Defining Logical Names under TOPS-20.

The title is the one that is printed as the second line of the histogram report. It must be enclosed in single quotation marks and can have a maximum length of 70 characters.

Once you specify a file specification and/or title, it becomes the default for any subsequent reports until explicitly changed.

The format for the histogram report is shown below. The heading is printed for each module that is in memory at the time the report is printed, even if the module was never entered. If the report is printed while a module for which statistics were gathered is not in memory, the statistics for that module are not printed.

```
COBDDT HISTOGRAM FOR module-name                REPORT:integer-1
title


PROCEDURE              ENTRIES        CPU                 ELAPSED
-section-name-         integer-2      time-1              time-2
paragraph-name         integer-3      time-3              time-4

OVERHEAD:          ELAPSED:time-5     CPU:time-6
UNACCOUNTED:       ELAPSED:time-7     CPU:time-8
```

module-name	is the name of the module, taken from the PROGRAM ID clause.
integer-1	is the report number. It starts at 1 and is incremented by 1 for each report produced in a run.
title	is the title that you specified in one of the HISTORY commands.
section-name	is the name of a section into which control was transferred or passed. Each paragraph in the section to which control was passed is given with the section.
integer-2	is the number of times control was passed directly to the section.
time-1	is the amount of CPU time spent in the section.
time-2	is the amount of elapsed time spent in the section.
paragraph-name	is the name of a paragraph to which control was transferred or passed.
integer-3	is the number of times control was passed to this paragraph.
time-3	is the amount of CPU time spent in this paragraph.
time-4	is the amount of elapsed time spent in this paragraph.
time-5	is the elapsed time spent entering and exiting from subprograms and PERFORM statements. If this time is 0, the line is not printed.
time-6	is the CPU time spent entering and exiting from subroutines and PERFORM statements.
time-7	is the elapsed time that could not be charged to any section or paragraph. If this time is 0, the line is not printed.
time-8	is the CPU time that could not be charged to any section or paragraph. For example, when a subprogram is entered, the time accrued until the first paragraph or section is seen is charged to unaccounted.

If control is never passed to a particular section or paragraph, nothing is printed for that section or paragraph. When a PERFORM statement or subprogram is entered, the current paragraph or section is saved on a stack so that COBDDT can continue to charge time to the correct section or paragraph when the return is done. The size of the stack is 20 locations. After a depth of twenty calls or PERFORM statements is reached, time is charged to unaccountable.

A sample histogram report is shown below.

```
COBDDT HISTOGRAM FOR CASHX                              REPORT:  1

PROCEDURE                          ENTRIES        CPU       ELAPSED

-GENERATED-SECTION-NAME-              0          1.360      21.707
 START                              721          0.008       2.641
 ST-1                                 1          0.000       0.000
 START-2                            721          0.385       5.616
 INITIAL-SETUP                        1          0.016       0.233
 END-INITIAL-SETUP                    1          0.000       0.017
 CONVERT-RECORDS                    721          0.400       5.575
 END-CONVERT-RECORDS                721          0.167       2.146
 RATE-IT                            721          0.178       2.086
 END-RATE-IT                        721          0.206       3.393
```

7.3.3.5 **Using the Histogram Feature** - To use the histogram feature, issue the following commands upon entering COBDDT for the first time.

```
HISTORY INITIALIZE
HISTORY BEGIN
```

At any time when you are stopped at a breakpoint, you can stop gathering statistics for the histogram by issuing the HISTORY END command. If you issue a HISTORY BEGIN command after a HISTORY END command, the histogram continues from the point where the HISTORY BEGIN command was issued. However, if after a HISTORY END command you issue a HISTORY INITIALIZE and a HISTORY BEGIN command, the previous statistics are lost and a new histogram begun. To get the previous histogram, issue a HISTORY REPORT command before the HISTORY INITIALIZE command.

If a histogram file already exists with the same file specification as the one given, the histogram report is appended to the existing file. If the file specification is different, COBDDT starts a new histogram file.

7.4 RERUN - PROGRAM TO RESTART COBOL PROGRAMS

The RERUN program is used to restart a COBOL program that has been terminated abnormally due to a system failure, a device error, or an exceeded disk quota. RERUN uses checkpoint files, which are similar to memory-image dump files. Checkpoint files are created in one of two ways:

- By including RERUN statement(s) in the COBOL program itself
- By typing CTRL/C twice followed by REENTER during program execution

The COBOL system creates a checkpoint file by writing a memory-image dump file of the program onto disk and adding some other information to allow a later restart of the program. At the same time, the COBOL system closes and reopens all disk and magnetic tape output files. The dump is not performed if a sort is in progress. Each time the checkpoint file is written, the COBOL system types the message DUMP COMPLETED on your terminal.

If the COBOL program is interrupted during execution, you can restart the program by means of the RERUN program. The RERUN program reads the dump file back into memory, restores the files to their state at the time the checkpoint file was written, and then passes control to the COBOL program so that it can continue processing to completion. RERUN assumes that the operating environment at the time the COBOL program was interrupted is the same as the environment at the time the checkpoint file was written. Thus, the files must be associated with the same types of devices, and devices must have the same logical names.

7.4.1 Operating RERUN

To restart a COBOL program from the last checkpoint file written before execution stopped, type R RERUN in response to the operating system prompt (users of TOPS-20 can respond RERUN). For example:

```
.R RERUN(RET)  for users of TOPS-10

              or

@RERUN(RET)  for users of TOPS-20
```

If you are running on TOPS-20, RERUN displays a prompt:

```
RERUN>
```

The first message you see from TOPS-10 RERUN is the second message you see from TOPS-20 RERUN, and both are followed by an asterisk prompt:

```
TYPE CHECKPOINT FILENAME
*
```

At this point you type the name of the checkpoint file in which the core-image dump is stored.

When a checkpoint dump is being written, the COBOL system uses the filename of the program as the name of the checkpoint file and adds the extension .CKP. If the COBOL program does not have a filename because it was not saved, the COBOL system takes the checkpoint filename from the PROGRAM-ID in the program and adds the extension .CKP. If the program has been divided into a 2-segment file, the high-segment filename must be the same as the low-segment filename. Thus, when you respond with the checkpoint filename you are in effect telling RERUN the program name as well.

If TOPS-10 RERUN encounters a logical device name in the program, it types the following message:

```
ASSIGN device-name
TYPE CONTINUE WHEN DONE
```

and exits to monitor command level. If this happens, you should give the appropriate ASSIGN command to assign the logical device to a specific one, then a CONTINUE monitor command to restart RERUN.

The TOPS-20 RERUN works similarly. If it finds a logical name, it prints the following:

```
DEFINE logical-name: (AS) device:
TYPE CONTINUE WHEN DONE
```

TOPS-20 RERUN exits to the monitor. You should then give the appropriate DEFINE command followed by a CONTINUE command to restart RERUN.

7.4.2 Examples Of Using RERUN

In the following example, you have a COBOL program that was terminated by a system failure. Checkpoints had been inserted in the program by means of RERUN statements. The program has a filename of ACCNT; thus, the checkpoint filename is ACCNT.CKP. Instead of running the program again from the beginning, you employ the RERUN program to restart your program from the last checkpoint written before the program stopped. You type:

```
.R RERUN(RET)
```

The RERUN> prompt is not printed above because the example is running on a TOPS-10 system. RERUN responds:

```
TYPE CHECKPOINT FILENAME
*
```

You type:

```
ACCNT.CKP (RET)
```

RERUN loads the checkpoint file into memory, reopens and repositions the magnetic tape and disk files, and passes control to the COBOL program so that it can continue processing to completion.

In the example below, you are running a COBOL program that is notified the system is going down. You do not have any RERUN statements in your program, yet you wish to create a checkpoint file so that the processing done by your COBOL program up to that point is not wasted. You create the checkpoint file by typing CTRL/C twice and then typing REENTER. The checkpoint file is written by the COBOL system onto disk with a filename of PROG13 (taken from the PROGRAM-ID) and an extension of .CKP. After the system is restored, you can restart the program by running the RERUN program. The dialogue is as follows:

```
@RERUN (RET)
RERUN>
TYPE CHECKPOINT FILENAME
*
PROG13.CKP (RET)
```

The program PROG13 is loaded into memory, its files are reopened, and it continues running to completion.

CHAPTER 8

FILE FORMATS

8.1 RECORDING MODES

The recording mode specifies the byte size of the data and, except for binary mode, also specifies the character set used. The four recording modes and their respective byte sizes are:

RECORDING MODE	BYTE SIZE
ASCII	7 bits
SIXBIT	6 bits
EBCDIC	8 bits
Binary	36 bits (1 word)

The following sections describe the recording modes in more detail.

8.1.1 ASCII Recording Mode

An ASCII word consists of 5 characters left-justified in the word. Each character is represented by a 7-bit byte:

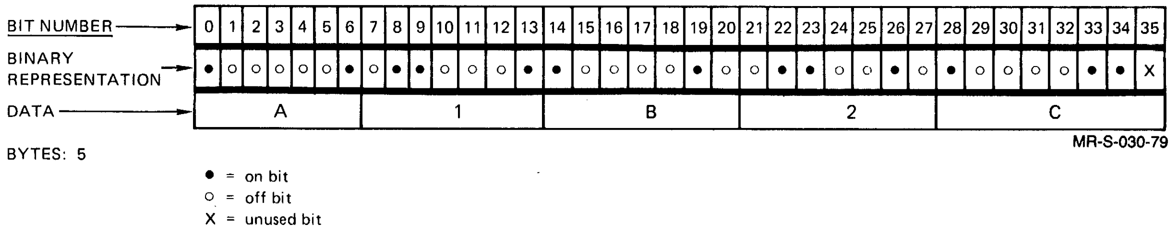


Figure 8-1 ASCII Recording Mode

NOTE

A variant form of ASCII, line-sequence ASCII, sets bit 35 of the line-sequence word to 1.

8.1.2 SIXBIT Recording Mode

SIXBIT is a compressed form of ASCII in which lowercase letters and a few special characters are not used. A SIXBIT word consists of 6 characters per word, with each character represented by a 6-bit byte:

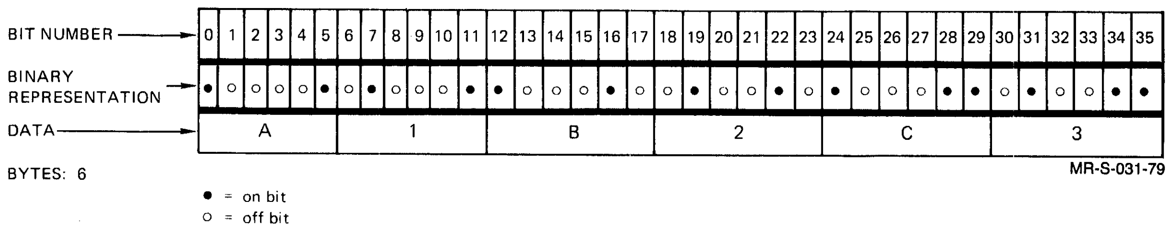


Figure 8-2 SIXBIT Recording Mode

8.1.3 EBCDIC Recording Mode

An EBCDIC word consists of 4 characters per word. Each byte is 9 bits long, but the first bit in each byte is unused. Each character is represented by 8 bits:

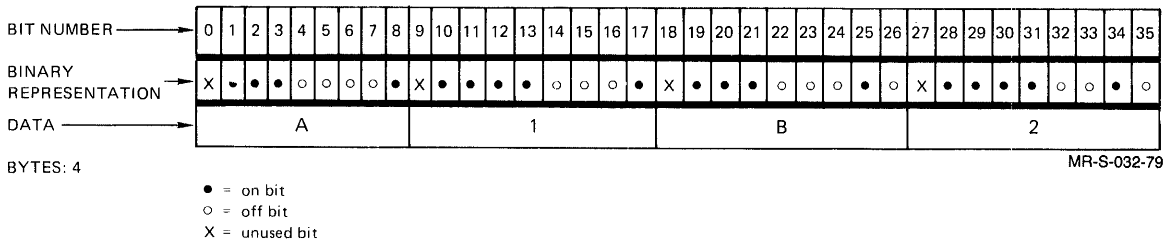


Figure 8-3 EBCDIC Recording Mode

A variant form, used only for magnetic tape, is industry-compatible EBCDIC. In this form of EBCDIC, there are 4 characters per word, left-justified within the word. Each character is represented by an 8-bit byte. The last 4 bits in the word are unused:

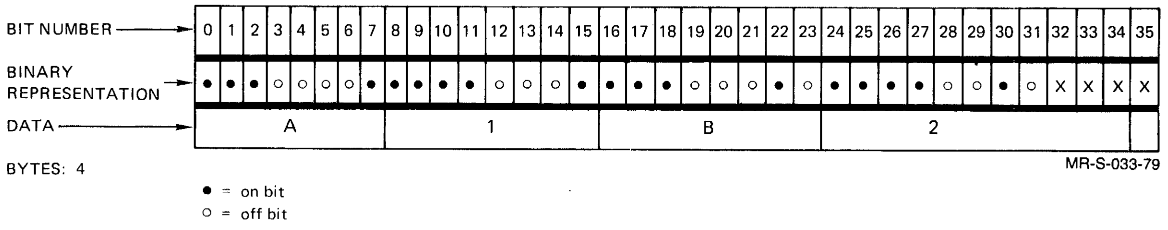


Figure 8-4 EBCDIC Recording Mode - Industry-Compatible

8.1.4 BINARY Recording Mode

Unlike the recording modes previously mentioned, binary mode does not specify a character set for the data. In binary mode, the entire 36-bit word is interpreted as a single byte of binary data:

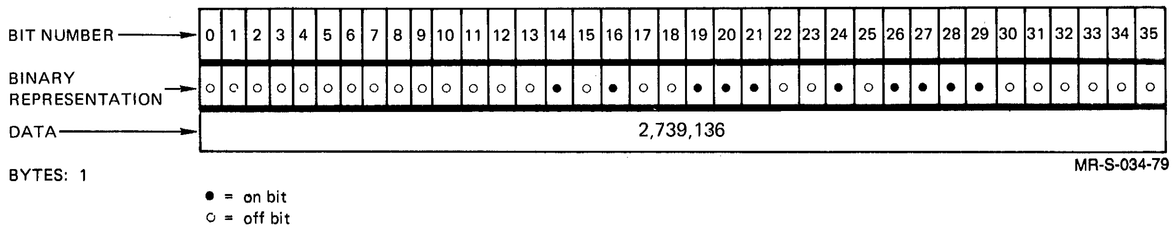


Figure 8-5 Binary Recording Mode

8.2 FILE FORMATS

The file format specifies the structure of the record used to store the data. The following sections describe all major file formats. Each section includes a diagram of the file format and a COBOL code segment that generates the file format.

The following conventions are used in the diagrams:

1. Alphanumeric or numeric character data in a word is shown with each individual character enclosed in a box. The box represents 1 byte. Thus, a word of ASCII data would be shown as follows:

A	B	C	D	E

2. Binary data in a word (fixed- and floating-point numbers) is shown by a number in the word:

32156.10

3. EBCDIC packed-decimal values are shown as two decimal digits per EBCDIC byte. The right half of the rightmost byte contains the sign. Neither the digits nor the sign are EBCDIC characters.

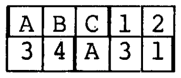
4. COBOL signed numeric data, such as produced by PIC S9(n), is shown with the over-punched character if the sign is negative. For example, -12345 is shown as 1234N, with the N representing both the negative sign and the value 5. DIGITAL's COBOL does not use over-punched characters for positive sign representation, so diagrams depicting positive, signed numeric data do not show a sign.

5. Italicized characters in a diagram do not depict data; they label or clarify parts of the diagram:

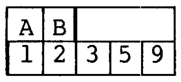
RDW 30	0

   MR-S-034A-81

6. Heavy vertical lines are used to delimit individual fields within a record:



7. Padding, the use of blanks or nulls to force the next record to begin on some boundary (for example, a word or disk-block boundary), is shown by white space in the word:



You cannot consider padding as part of a record field, nor can you use padding as part of a key field. However, the length of any padding must be taken into account when calculating record length and key starting position.

8.2.1 Fixed-Length ASCII

A fixed-length ASCII file consists of records containing five characters per 36-bit word, with each group of five characters left-justified within the word. Fixed-length ASCII records must end with a carriage return/line feed. The following diagram illustrates the format of fixed-length ASCII records:

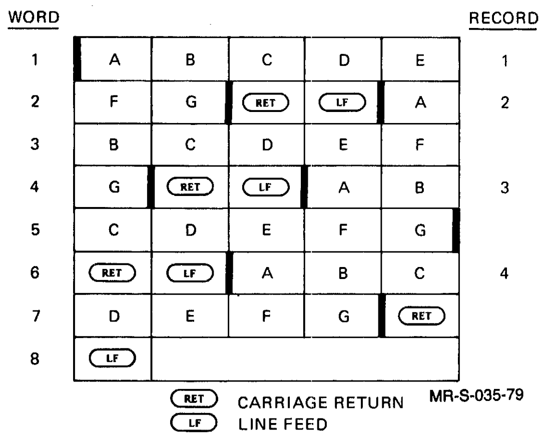


Figure 8-6 Fixed-Length ASCII

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS ASCII.

DATA DIVISION.
FILE SECTION.

FD filename        VALUE OF ID "DATA  FIL".
01 record-1        DISPLAY-7.
   02 field-1      PIC X(6).
   02 field-2      PIC A(3).
   02 field-3      PIC 9(4).
   02 field-4      PIC S9(6).
   02 field-5      PIC S9(6)V9999.

PROCEDURE DIVISION.

LOAD-PARAGRAPH.
        MOVE "AB12EF" TO field-1.
        MOVE "GHI" TO field-2.
        MOVE 3249 TO field-3.
        MOVE -481253 TO field-4.
        MOVE 31458.5012 TO field-5.
        WRITE record-1.
```

Figure 8-7 illustrates the record produced by the code segment shown above:

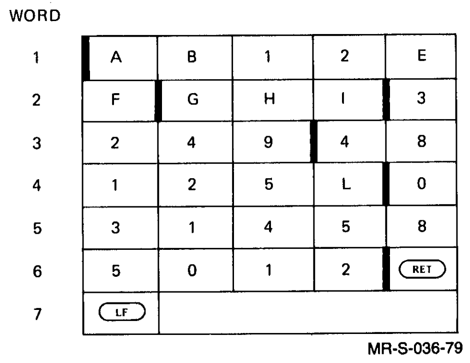


Figure 8-7 COBOL Fixed-Length ASCII

8.2.2 Variable-Length ASCII

Variable-length ASCII consists of records containing five characters per 36-bit word, with each group of five characters left-justified within the word. Variable-length ASCII records must end with some combination of the following control characters:

1. Carriage return

2. Line feed

3. Vertical tab

4. Form feed

The following diagram illustrates the format of variable-length ASCII records:

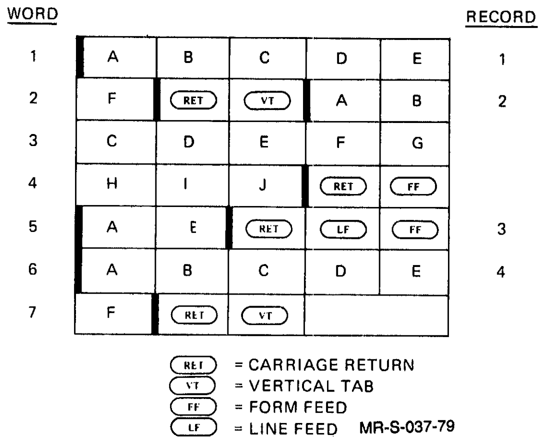


Figure 8-8 Variable-Length ASCII

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS ASCII.

DATA DIVISION.
FILE SECTION.

FD filename        VALUE OF ID "DATA  FIL".
01 record-1        DISPLAY-7.
   02 field-1      PIC X(7).
   02 field-2      PIC S9(7)V99.
   02 field-3      PIC A(3).
   02 field-4      PIC 9(4).

01 record-2        DISPLAY-7.
   02 field-1      PIC X(7).
   02 field-2      PIC S9(7)V99.
   02 field-3      PIC A(3).
   02 field-4      PIC 9(7).

PROCEDURE DIVISION.

LOAD-PARAGRAPH-1.
        MOVE "AB13521" TO field-1 OF record-1.
        MOVE -3269.02 TO field-2 OF record-1.
        MOVE "ILM" TO field-3 OF record-1.
        MOVE 1359 TO field-4 OF record-1.
        WRITE record-1.

LOAD-PARAGRAPH-2.
        MOVE "EFGHI95" TO field-1 OF record-2.
        MOVE 42553.40 TO field-2 OF record-2.
        MOVE "LMN" TO field-3 OF record-2.
        MOVE 3712536 TO field-4 OF record-2.
        WRITE record-2.
```

Figure 8-9 illustrates the record produced by the code segment shown above:

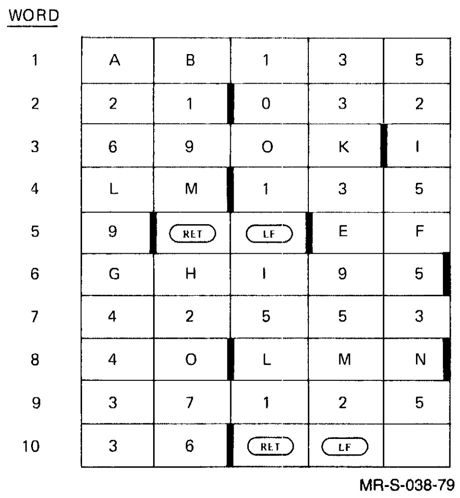


Figure 8-9 COBOL Variable-Length ASCII

8.2.3 Fixed-Length SIXBIT

In a SIXBIT file, characters are stored six per 36-bit word, and a SIXBIT record must start and end on a word boundary. The left half of the first word in the record contains one of the following:

1. The record sequence number of COBOL magnetic tape records

2. Data specific to COBOL ISAM records

3. Binary zeros

The right half of the first word contains the number of characters in the record. To ensure that the record ends on a word boundary, the last word in the record is padded with blanks, if necessary. When determining the size of the record for memory considerations, you must take into account the first word of the record (containing file-access information and a character count) and the possible existence of padding characters (blanks) to enable the record to end on a word boundary.

The following diagram illustrates the format of fixed-length SIXBIT records. Note that the character count is the same for each record:

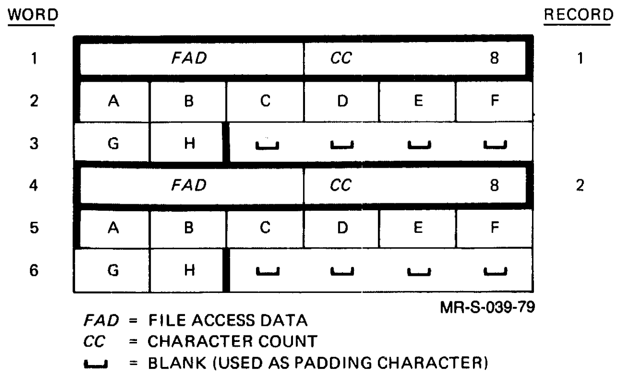


Figure 8-10 Fixed-Length SIXBIT

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS SIXBIT.

DATA DIVISION.
FILE SECTION.

FD filename      VALUE OF ID "DATA  FIL".
01 record-1      DISPLAY-6.
   02 field-1    PIC X(4).
   02 field-2    PIC A(5).
   02 field-3    PIC 9(10) COMP.
   02 field-4    PIC X(2).
   02 field-5    PIC 9(11) COMP.
   02 field-6    PIC 9(4).
   02 field-7    COMP-1.
   02 field-8    PIC 9(11) COMP.

PROCEDURE DIVISION.

LOAD-PARAGRAPH.
        MOVE "A13B" TO field-1.
        MOVE "CDEFG" TO field-2.
        MOVE 9654839218 TO field-3.
        MOVE "HI" TO field-4.
        MOVE 34567982314 TO field-5.
        MOVE 1289 TO field-6.
        MOVE 123.45 TO field-7.
        MOVE 12398756983 TO field-8.
        WRITE record-1.
```

Figure 8-11 illustrates the record produced by the code segment shown above:

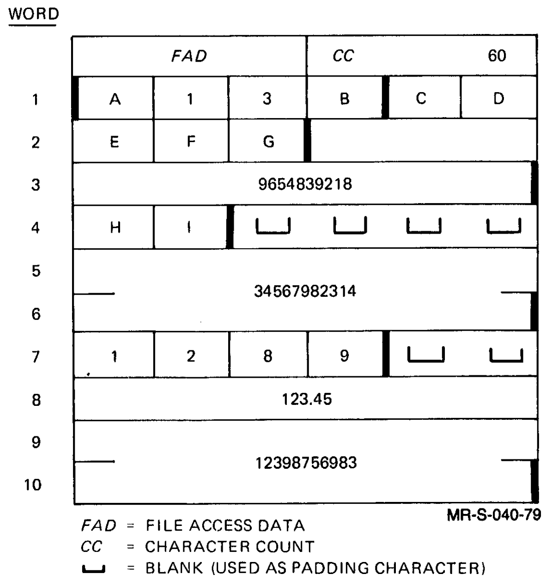


Figure 8-11 COBOL Fixed-Length SIXBIT

8.2.4 Variable-Length SIXBIT

This format is the same as fixed-length SIXBIT, except that the character count can vary from record to record. The following diagram illustrates the format of variable-length SIXBIT records:

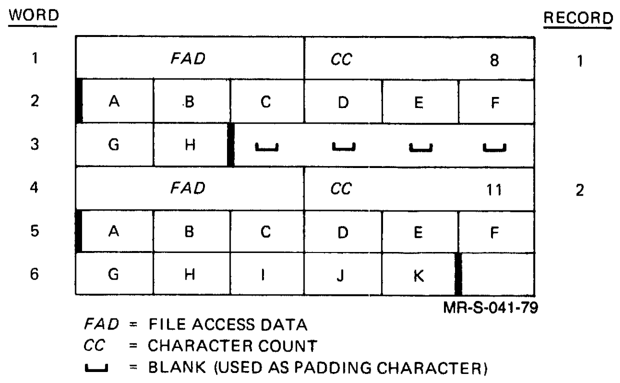


Figure 8-12 Variable-Length SIXBIT

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS SIXBIT.

DATA DIVISION.
FILE SECTION.

FD filename       VALUE OF ID "DATA  FIL".
01 record-1       DISPLAY-6.
   02 field-1     COMP-1.
   02 field-2     PIC X(3).
   02 field-3     PIC A(3).
   02 field-4     PIC 9(3).
   02 field-5     PIC 9(10) COMP.
   02 field-6     PIC 9(11) COMP.
   02 field-7     PIC X(2).
   02 field-8     PIC 9(5) COMP.

01 record-2       DISPLAY-6.
   02 field-1     COMP-1.
   02 field-2     PIC X(3).
   02 field-3     PIC A(3).
   02 field-4     PIC 9(3).
   02 field-5     PIC 9(10) COMP.
   02 field-6     PIC 9(11) COMP.
   02 field-7     PIC X(2).
   02 field-8     PIC 9(11) COMP.

PROCEDURE DIVISION.

LOAD-PARAGRAPH-1.
        MOVE 123.4567 TO field-1 OF record-1.
        MOVE "A3C" TO field-2 OF record-1.
        MOVE "DEF" TO field-3 OF record-1.
        MOVE -55 TO field-4 OF record-1.
        MOVE 1234567809 TO field-5 OF record-1.
        MOVE 98765432108 TO field-6 OF record-1.
        MOVE "A2" TO field-7 OF record-1.
        MOVE 32571 TO field-8 OF record-1.
        WRITE record-1.

LOAD-PARAGRAPH-2.
        MOVE 1395.678 TO field-1 OF record-2.
        MOVE "B5L" TO field-2 OF record-2.
        MOVE "LMN" TO field-3 OF record-2.
        MOVE 79 TO field-4 OF record-2.
        MOVE 8176596821 TO field-5 OF record-2.
        MOVE 18976532150 TO field-6 OF record-2.
        MOVE "M5" TO field-7 OF record-2.
        MOVE 12357986183 TO field-8 OF record-2.
        WRITE record-2.
```

Figure 8-13 illustrates the record produced by the code segment shown above:

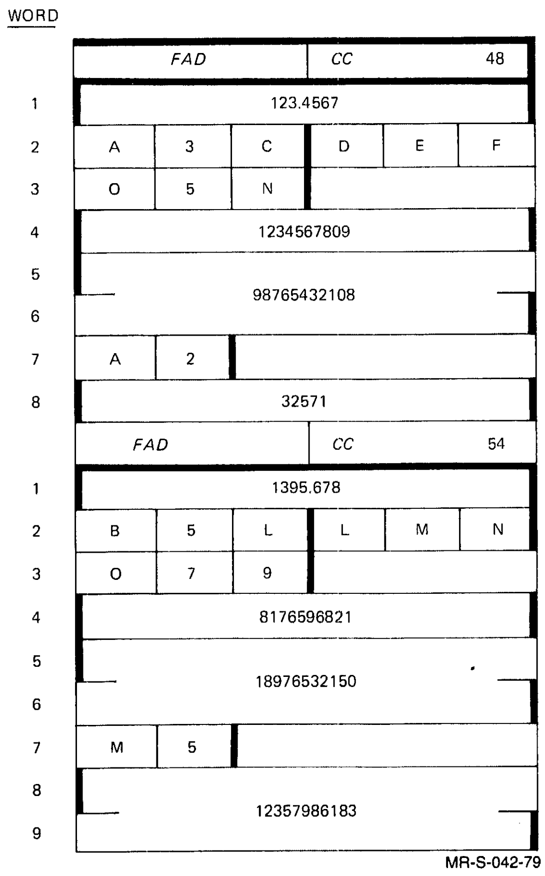


Figure 8-13 COBOL Variable-Length SIXBIT

8.2.5 EBCDIC File Formats

On disk and in memory, the characters in an EBCDIC file are represented by 8 bits right-justified in 9-bit bytes. On tape, the characters in an EBCDIC file are represented by 8-bit bytes, and 4 bytes occur per 36-bit word. Within a given file, records can be either fixed or variable length, and can be either blocked or unblocked. Thus, there are four types of EBCDIC files:

1. Fixed-length
2. Variable-length
3. Blocked fixed-length
4. Blocked variable-length

In a file written in fixed-length EBCDIC, records all have the same record length and the records need not begin or end on a word boundary. The following diagram illustrates the format of fixed-length EBCDIC records in an unblocked file:

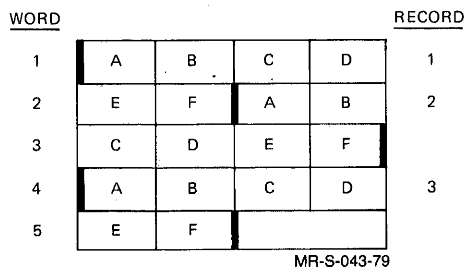


Figure 8-14 Fixed-Length EBCDIC

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS F.

DATA DIVISION.
FILE SECTION.

FD filename       VALUE OF ID "DATA  FIL".
01 record-1       DISPLAY-9.
   02 field-1     PIC 9(3).
   02 field-2     PIC X(5).
   02 field-3     PIC A(2).
   02 field-4     PIC 9(9) COMP-3.
   02 field-5     PIC S9(6) COMP-3.

PROCEDURE DIVISION.

LOAD-PARAGRAPH.
        MOVE 123 TO field-1.
        MOVE "ABCDE" TO field-2.
        MOVE "LM" TO field-3.
        MOVE 137958795 TO field-4.
        MOVE -351235 TO field-5.
        WRITE record-1.
```

Figure 8-15 illustrates the record produced by the code segment shown above:

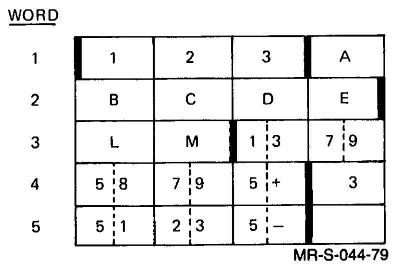


Figure 8-15 COBOL Fixed-Length EBCDIC

In a file written in variable-length EBCDIC format, the record lengths can vary from record to record. Each record contains a 4-byte Record Descriptor Word (RDW) at the head of the record. The left half-word of the RDW specifies a value equal to the number of bytes in the record plus 4 (to allow for the length of the RDW itself). The rightmost 2 bytes of the RDW must be zero; if they are nonzero, they indicate spanned records, which are unsupported. The following diagram illustrates the format of variable-length EBCDIC records in an unblocked file:

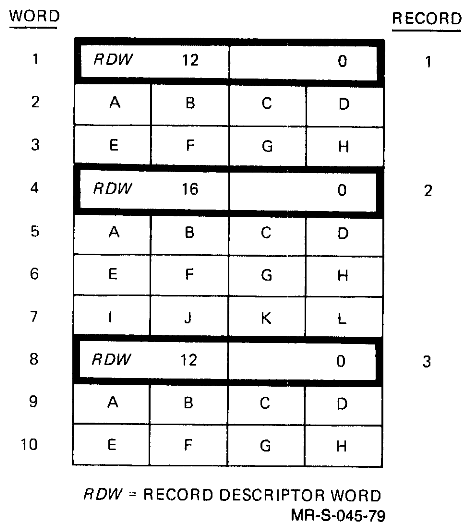


Figure 8-16 Variable-Length EBCDIC

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS V.

DATA DIVISION.
FILE SECTION.

FD filename       VALUE OF ID "DATA  FIL".
01 record-1       DISPLAY-9.
   02 field-1     PIC S9(7) COMP-3.
   02 field-2     PIC S9(8) COMP-3.
   02 field-3     PIC 9(3).
   02 field-4     PIC A(2).
   02 field-5     PIC X(5).

01 record-2       DISPLAY-9.
   02 field-1     PIC S9(7) COMP-3.
   02 field-2     PIC S9(8) COMP-3.
   02 field-3     PIC 9(3).
   02 field-4     PIC A(2).
   02 field-5     PIC X(8).

PROCEDURE DIVISION.

LOAD-PARAGRAPH-1.
        MOVE -1398569 TO field-1 OF record-1.
        MOVE 57635937 TO field-2 OF record-1.
        MOVE 596 TO field-3 OF record-1.
        MOVE "AB" TO field-4 OF record-1.
        MOVE "A13DE" TO field-5 OF record-1.
        WRITE record-1.

LOAD-PARAGRAPH-2.
        MOVE 5369787 TO field-1 OF record-2.
        MOVE -53896156 TO field-2 OF record-2.
        MOVE 593 TO field-3 OF record-2.
        MOVE "MN" TO field-4 OF record-2.
        MOVE "ILH5MLXY" TO field-5 OF record-2.
        WRITE record-2.
```

Figure 8-17 illustrates the record produced by the code segment shown above:

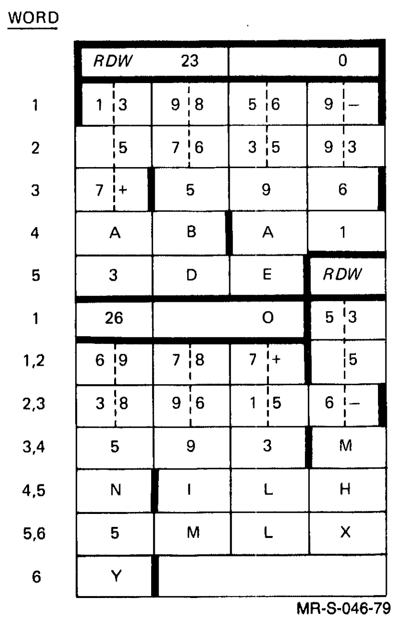


Figure 8-17 COBOL Variable-Length EBCDIC

Fixed-length EBCDIC records can also be blocked. In this file format, fixed-length EBCDIC records are written in groups (or blocks). Each new block begins on a disk-block boundary. For tapes, each block starts a new physical magnetic-tape record.

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS F.

DATA DIVISION.
FILE SECTION.

FD filename     VALUE OF ID "DATA  FIL"
        BLOCK CONTAINS 1 RECORD.
01 record-1     DISPLAY-9.
   02 field-1   PIC 9(3).
   02 field-2   PIC X(5).
   02 field-3   PIC A(2).
   02 field-4   PIC S9(5) COMP-3.
   02 field-5   PIC S9(4) COMP-3.

01 record-2     DISPLAY-9.
   02 field-1   PIC X(3).
   02 field-2   PIC X(5).
   02 field-3   PIC A(2).
   02 field-4   PIC S9(5) COMP-3.
   02 field-5   PIC S9(4) COMP-3.

PROCEDURE DIVISION.

LOAD-PARAGRAPH-1.
        MOVE 194 TO field-1 OF record-1.
        MOVE "BDEFG" TO field-2 OF record-1.
        MOVE "MN" TO field-3 OF record-1.
        MOVE 13796 TO field-4 OF record-1.
        MOVE 1985 TO field-5 OF record-1.
        WRITE record-1.

LOAD-PARAGRAPH-2.
        MOVE "762" TO field-1 OF record-2.
        MOVE "LANBH" TO field-2 OF record-2.
        MOVE "AB" TO field-3 OF record-2.
        MOVE 76543 TO field-4 OF record-2.
        MOVE -9764 TO field-5 OF record-2.
        WRITE record-2.
```

Figure 8-18 illustrates the record produced by the code segment shown above:

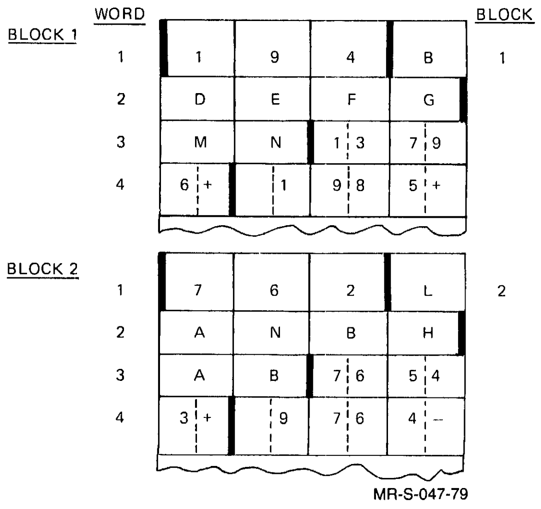


Figure 8-18 COBOL Blocked Fixed-Length EBCDIC

Variable-length EBCDIC records can be blocked as well. In this file format, the record length can vary from record to record. Each record contains a 1-word Record Descriptor Word (RDW) at the head of the record. This word contains (in the left half-word) a count of all bytes in the record and in the RDW itself. The right half of the RDW must be zero. The records are read and written in groups called blocks. The actual number of records in a block depends on the blocking factor specified when the file was created. Each block of records contain a 1-word Block Descriptor Word (BDW) which contains a count (in the left half-word) of the bytes in the block. That is, the bytes of data and the bytes of the RDW for each record in the block and the 4 bytes of the BDW itself are included in the block count. The following illustrates the format of blocked variable-length EBCDIC records:

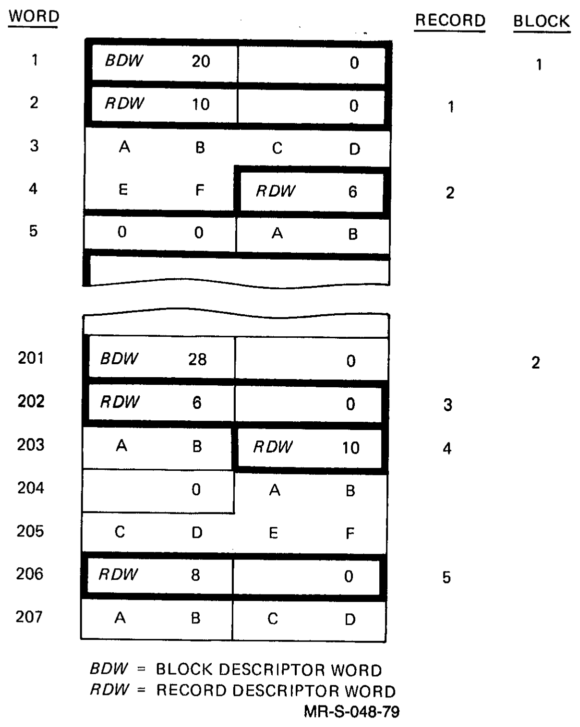


Figure 8-19 Blocked Variable-Length EBCDIC

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS V.

DATA DIVISION.
FILE SECTION.

FD filename     VALUE OF ID "DATA  FIL"
        BLOCK CONTAINS 1 RECORD.
01 record-1     DISPLAY-9.
   02 field-1   PIC S9(7) COMP-3.
   02 field-2   PIC S9(7) COMP-3.
   02 field-3   PIC X(3).
   02 field-4   PIC A(2).
   02 field-5   PIC S9(9) COMP-3.
   02 field-6   PIC X(6).

01 record-2     DISPLAY-9.
   02 field-1   PIC S9(7) COMP-3.
   02 field-2   PIC S9(7) COMP-3.
   02 field-3   PIC X(3).
   02 field-4   PIC A(2).
   02 field-5   PIC 9(9) COMP-3.
   02 field-6   PIC X(9).

PROCEDURE DIVISION.

LOAD-PARAGRAPH-1.
        MOVE +9356127 TO field-1 OF record-1.
        MOVE 3987156 TO field-2 OF record-1.
        MOVE "198" TO field-3 OF record-1.
        MOVE "MN" TO field-4 OF record-1.
        MOVE -569138279 TO field-5 OF record-1.
        MOVE "ABCDEF" TO field-6 OF record-1.
        WRITE record-1.

LOAD-PARAGRAPH-2.
        MOVE -3295865 TO field-1 OF record-2.
        MOVE 9378518  TO field-2 OF record-2.
        MOVE "196" TO field-3 OF record-2.
        MOVE "AL" TO field-4 OF record-2.
        MOVE 569138279 TO field-5 OF record-2.
        MOVE "ABCDEFGHI" TO field-6 OF record-2.
        WRITE record-2.
```

Figure 8-20 illustrates the record produced by the code segment shown above:

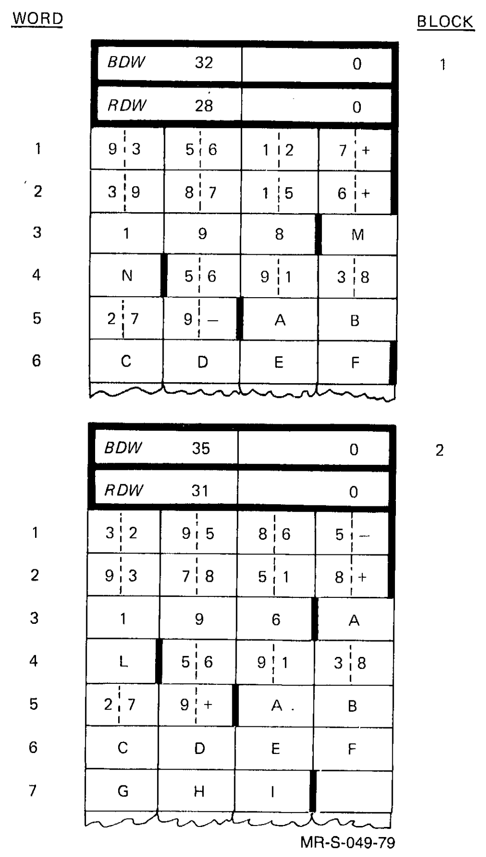


Figure 8-20 COBOL Blocked Variable-Length EBCDIC

8.2.6 BINARY File Formats

Binary records consist of contiguous 36-bit words. Each record starts and ends on a word boundary. Binary is the only recording mode which does not have a character set associated with it, and standard binary records can only be interpreted as COMPUTATIONAL and COMP-1 binary numbers. However, it is possible to associate a character set with binary records by writing mixed-mode records. COBOL programs are capable of writing three mixed-mode binary formats. Each format is shown below:

8.2.6.1 COBOL ASCII Mixed-Mode Binary -

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS BINARY.

DATA DIVISION.
FILE SECTION.

FD filename       VALUE OF ID "DATA  FIL".
01 binary-rec     DISPLAY-7.
   02 field-1     PIC S9(10) COMP.
   02 field-2     COMP-1.
   02 field-3     PIC X(7).
   02 field-4     PIC 9(11) COMP.
   02 field-5     PIC X(3).
   02 field-6     PIC 9(14) COMP.
   02 field-7     PIC A(2).

PROCEDURE DIVISION.

LOAD-PARAGRAPH.
        MOVE 1234568910 TO field-1.
        MOVE 1246.5978 TO field-2.
        MOVE "ABCDE12" TO field-3.
        MOVE 12345678954 TO field-4.
        MOVE "532" TO field-5.
        MOVE 12345678954967 TO field-6.
        MOVE "LM" TO field-7.
        WRITE binary-rec.
```

Figure 8-21 illustrates the record produced by the code segment shown above:

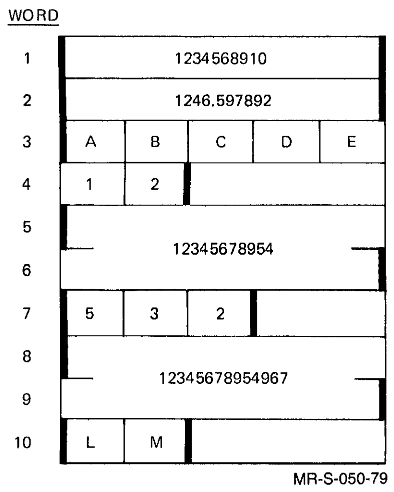


Figure 8-21 COBOL Standard Binary and ASCII Mixed-Mode Binary

8.2.6.2 COBOL SIXBIT Mixed-Mode Binary -

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS BINARY.

DATA DIVISION.
FILE SECTION.

FD filename      VALUE OF ID "DATA  FIL".
01 binary-rec    DISPLAY-6.
   02 field-1    PIC S9(10) COMP.
   02 field-2    COMP-1.
   02 field-3    PIC X(7).
   02 field-4    PIC 9(11) COMP.
   02 field-5    PIC X(3).
   02 field-6    PIC 9(14) COMP.
   02 field-7    PIC A(2).

PROCEDURE DIVISION.

LOAD-PARAGRAPH.
        MOVE 1234567891 TO field-1.
        MOVE 1234.5921 TO field-2.
        MOVE "ABCDE12" TO field-3.
        MOVE 12345678954 TO field-4.
        MOVE "532" TO field-5.
        MOVE 12345678954967 TO field-6.
        MOVE "LM" TO field-7.
        WRITE binary-rec.
```

Figure 8-22 illustrates the record produced by the code segment shown above:

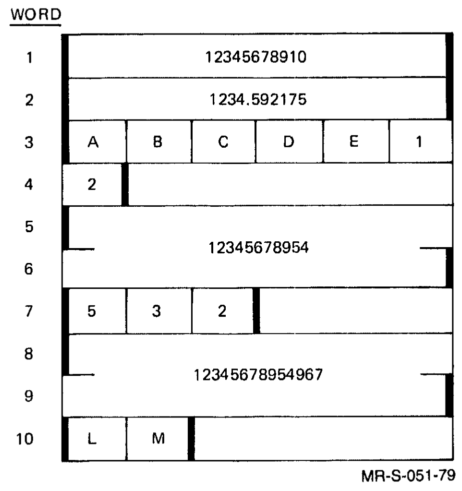


Figure 8-22 COBOL Standard Binary and SIXBIT Mixed-Mode Binary

8.2.6.3 COBOL EBCDIC Mixed-Mode Binary -

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
        RECORDING MODE IS BINARY.

DATA DIVISION.
FILE SECTION.

FD filename       VALUE OF ID "DATA  FIL".
01 binary-rec     DISPLAY-9.
   02 field-1     PIC S9(10) COMP.
   02 field-2     COMP-1.
   02 field-3     PIC X(7).
   02 field-4     PIC 9(11) COMP.
   02 field-5     PIC 9(3).
   02 field-6     PIC 9(14) COMP.
   02 field-7     PIC A(2).
   02 field-8     PIC S9(5) COMP-3.
   02 field-9     PIC 9(8) COMP-3.

PROCEDURE DIVISION.

LOAD-PARAGRAPH.
        MOVE 1234567891 TO field-1.
        MOVE 1246.5978 TO field-2.
        MOVE "ABCDE12" TO field-3.
        MOVE 12345678954 TO field-4.
        MOVE "532" TO field-5.
        MOVE 12345678954967 TO field-6.
        MOVE "LM" TO field-7.
        MOVE -72539 TO field-8.
        MOVE 36193586 TO field-9.
        WRITE binary-rec.
```

Figure 8-23 illustrates the record produced by the code segment shown above:

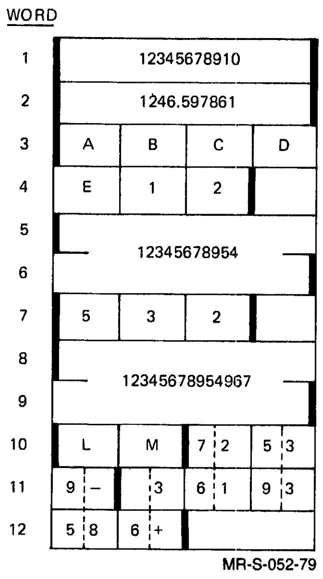


Figure 8-23 COBOL Standard Binary and EBCDIC Mixed-Mode Binary

8.3 FILE ORGANIZATION AND ACCESS

File organization refers to the manner in which the records are arranged in a file. File access refers to the method by which records in the file are read and written. COBOL-68 supports three types of file organization: sequential, random, and indexed-sequential. Each type of file organization has a corresponding method of access. Sequential files are accessed sequentially only, that is, in the order in which they are recorded. Random and indexed-sequential files can be accessed either sequentially or randomly.

You use the ACCESS MODE clause in the Environment Division to specify the method of access which you wish to use. The chart below shows the types of COBOL-68 files and the methods by which they can be accessed.

File Organization	Method of Access	ACCESS MODE
Sequential	Sequential	SEQUENTIAL
Random	Sequential Random	SEQUENTIAL RANDOM
Indexed	Sequential Random	SEQUENTIAL INDEXED

In the following sections, file types are described in greater detail, along with the methods by which they can be accessed and the manner in which these methods are specified.

8.4 SEQUENTIAL FILES

Sequential files are those files that can only be read or written sequentially, that is, starting at the first record in the file and continuing with each subsequent record until the end of the file. Sequential files can reside on any file medium: cards, paper tape, DECtape (DECsystem-10 only), magnetic tape, and disk. If the file contains a large amount of data that is read and written frequently, it should be stored on magnetic tape or disk. Since tape storage is normally less expensive than disk storage, magnetic tape is often used for such files. However, if it is necessary to have rapid access to the data, disk storage can be preferable to tape storage. Sequential files on disk or DECtape should not be blocked unless they are to be open for input/output. When the files are stored on magnetic tape, they should be blocked to reduce wasted space caused by inter-record gaps.

A sequential file can be open for input/output (updating), but it must be blocked for this purpose and must reside on disk. If a sequential file is open for input/output, a write to the file causes writing of either the last record read (if the last operation was a READ) or the record after the last record written (if the last operation was a WRITE).

8.5 RANDOM FILES

Random files are arranged like sequential files, but differ from sequential files in the method by which they are accessed and in the devices on which they must be stored. The following requirements must be fulfilled for a random file:

1. It must be on a random-access device.

2. It must be blocked.

Random files can be accessed sequentially by declaring ACCESS MODE IS SEQUENTIAL in the SELECT statement of the FILE-CONTROL paragraph. This declaration allows you to treat your random file exactly like a sequential one. If you declare this, you must deal with the records in the order in which they are recorded - you can not access records by their relative position in the file.

Random files can be accessed randomly by declaring ACCESS MODE IS RANDOM in the SELECT statement. This declaration allows you to process records in any order you choose. You must specify the data-name of the item which holds the relative record number of the record you wish to access in the random file. This item is called the ACTUAL KEY, and is specified in the SELECT statement in the Environment Division. You must also specify the maximum range of relative record numbers in the FILE-LIMIT clause, which is also in the SELECT statement.

The data-name specified by the ACTUAL KEY must be described in the Working-Storage section of the Environment Division as a COMPUTATIONAL item of 10 or fewer digits. Its picture can only contain the characters S and 9 (or their equivalent, such as S9(4)). The ACTUAL KEY specifies to the object-time system the location of a record relative to the beginning of the file. That is, the ACTUAL KEY of the first record in the file is 1, that of the second is 2, and that of the last is n where n is the number of records in the file.

Some records in a random file can be zero-length; that is, they do not have anything written in them because the file was created randomly. These records have ACTUAL KEYs and can be written but cannot be read until information is placed into them. If an attempt is made to read zero-length records, the INVALID KEY path is taken.

You can create a random file by declaring its ACCESS MODE to be RANDOM and writing out records to the file. You can write the records either randomly or sequentially. To create a file randomly (that is, by writing into scattered or random records), you must open the file for output, move an integer value into the ACTUAL KEY for each record to be written, and write each record. To create a random file sequentially, simply open the file for output and begin writing records. The ACTUAL KEY defaults to the next record in the file, and the records are entered sequentially. No zero-length records are in the file if it is written sequentially.

8.5.1 Sequential Access Of Random Files

A random file can be accessed sequentially if you specify ACCESS MODE IS SEQUENTIAL in the FILE-CONTROL paragraph of the Environment Division. Read operations on such a file retrieves succeeding records, starting with the first non-zero-length record on the file, and continuing with each successive non-zero-length record. Any zero-length records are skipped by the sequential read operation. A successful sequential READ or WRITE updates the file's ACTUAL KEY value to indicate the current record position.

The AT END or INVALID KEY condition occurs if:

1. A READ is made to a non-existent record - this is logical End-of-File

2. A WRITE is made to a location containing a non-zero-length record

8.5.2 Random Access Of Random Files

A random file can be accessed at scattered locations if you specify the clause ACCESS MODE IS RANDOM in the FILE-CONTROL paragraph of the Environment Division. In this case the record accessed is the one indicated by the current value of the ACTUAL KEY. The first record on the file is assigned the key of 1, with succeeding records numbered 2, 3, 4, ... Therefore, before you execute a random I/O operation, you must specify the record by moving the value you desire into the ACTUAL KEY for the file. Non-zero-length records can be updated by the use of the WRITE clause, assuming that the file is open for I/O and that the previous I/O operation was a successful READ of the record in question.

The INVALID KEY condition occurs if:

1. A READ is made to a zero-length record

2. A WRITE is made to a non-zero-length record

A random file can be treated as a sequential file by declaring its ACCESS MODE to be SEQUENTIAL, but the file cannot be read or written randomly when this declaration has been made. However, if you declare that the ACCESS MODE IS RANDOM, you can access the records randomly or sequentially. You can access the records sequentially by moving zero to the ACTUAL KEY and acting as if the ACCESS MODE were SEQUENTIAL.

The following example shows the statements used to update a random file sequentially when the ACCESS MODE has been declared to be RANDOM.

```
ID DIVISION.
PROGRAM-ID. RNDTST.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT RNDOUT ASSIGN TO DSK
     ACCESS MODE IS RANDOM
     FILE LIMIT IS 100
     ACTUAL KEY IS RNDKEY.
DATA DIVISION.
FILE SECTION.
FD RNDOUT
     BLOCK CONTAINS 1 RECORD
     RECORDING MODE IS ASCII
     VALUE OF ID IS "FOO   FIL".
01 RNDREC.
     03 BEGIN-REC     PIC S999 USAGE COMP.
     03 MID-REC       PIC 9(4).
     03 FILLER        PIC X(50).
WORKING-STORAGE SECTION.
77 RNDKEY PIC 9(10) VALUE ZERO USAGE COMP.
PROCEDURE DIVISION.
START.
     OPEN INPUT-OUTPUT RNDOUT.
          .
          .
READ-AND-UPDATE-PARA.
     READ RNDOUT, INVALID KEY GO TO FINISH.
     PERFORM UPDATE-ROUTINE.
          .
          .
     WRITE RNDREC, INVALID KEY GO TO ERROR-ROUTINE.
     GO TO READ-AND-UPDATE-PARA.
FINISH.
     CLOSE RNDOUT, STOP RUN.
UPDATE-ROUTINE.
          .
          .
          .
ERROR-ROUTINE.
     DISPLAY "ERROR REPLACING RECORD", DISPLAY "RNDKEY ="RNDKEY.
     GO TO FINISH.
```

Figure 8-24 Statements Used to Sequentially Access a Random File

8.6 INDEXED-SEQUENTIAL FILES

Indexed-sequential files (also called ISAM files, for indexed-sequential access method) are files in which records are accessed through a hierarchy of indexes according to a key within each data record. This file organization is commonly used for applications in which the programmer wishes to identify and access records by the contents of a data field (the key) rather than the relative location of the record within the file. Some examples of applications for which this file organization is commonly used are:

- Payroll (key is employee number)
- Inventory control (key is part number)
- Production control (key is job or batch number)

An indexed-sequential file consists of two files: the indexed data file containing the actual data and the index file containing pointers to record keys within the indexed data file. The location of the record key within each record is specified when the file is built. To build an indexed-sequential file, you must provide a sequential file and some necessary information to the ISAM program. (See Section 7.1, ISAM, Indexed-Sequential File Maintenance Program.) ISAM then copies the data from the sequential file and creates an indexed data file and an index file to reference the indexed data file.

All reading and writing of the index file is performed by the object-time system; you need not be concerned with this function. When using indexed-sequential files, you need only specify which record is to be read, written, rewritten, or deleted. The object-time system performs all searching, insertion, deletion, and updating of both the index and indexed data files.

Indexed-sequential files must be accessed from disk. Also, because each indexed-sequential file is actually two files, two software I/O channels are required - one for the indexed data file and one for the index file.

8.6.1 Indexed Data File

The indexed data file can be recorded in EBCDIC, SIXBIT or ASCII; in any mode, the file must be blocked. When building an indexed-sequential file (by means of the ISAM utility program), you must provide a sequential file that contains record keys in the same relative location in each record. You are advised to sort the file in advance to insure that the most efficient index is built. Each record must have a unique key and the keys must be arranged in ascending order (numeric, alphabetic, or alphanumeric). You can indicate to the ISAM program that some records in each block are to be left empty and some empty blocks should be added to the file. The empty records and blocks are to allow for insertion or addition of new records in the file.

When a program processes the indexed-sequential file, insertions and additions are made by the object-time system. Records are inserted in a block in ascending order. When there are no empty record slots in the block, the block is split into two more-or-less-equal blocks, and the record is added to the appropriate block. New blocks created by insertions or additions are placed in the empty blocks that were allocated when the file was built. If empty records and blocks were not provided when the file was built, the object-time system requests additional blocks from the monitor as needed. If the monitor cannot allocate additional blocks (that is, because your quota on the file structure is exceeded or the system's limit was reached), an error message is issued.

The format of the indexed data file is similar to that of random and sequential files, with the following exceptions:

1. The right half of the header word contains the size of the record in bytes; the left half contains a version number. Only the version number of the first record of a block has any meaning; it pertains to all records for that block. All records (ASCII, SIXBIT, and EBCDIC) have a header word.

2. All records are line-blocked; they occupy an integral number of words. ASCII records always end with a single carriage return/line feed pair.

3. For ASCII records, the left half of the header word contains a version number, bits 18 through 34 contain the size of the record in bytes, and bit 35 is always 1.

Figure 8-25 shows the structure of an ISAM data file.

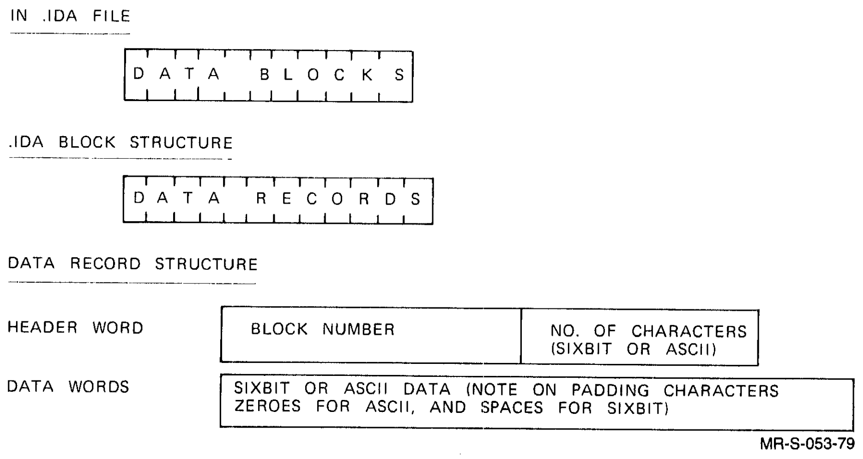

IN .IDA FILE
DATA BLOCKS
.IDA BLOCK STRUCTURE
DATA RECORDS
DATA RECORD STRUCTURE
HEADER WORD
BLOCK NUMBER
NO. OF CHARACTERS (SIXBIT OR ASCII)
DATA WORDS
SIXBIT OR ASCII DATA (NOTE ON PADDING CHARACTERS ZEROES FOR ASCII, AND SPACES FOR SIXBIT)
MR-S-053-79

Figure 8-25 ISAM Data File Structure

8.6.2 **Index File**

The index file is created by the ISAM program from the description of the input sequential data file and your parameters. It contains to ten levels of indexes, the lowest of which contains pointers to the record keys in the data file. Each successive level of index points to all blocks containing the next lower-level index. The highest level index is contained in one block and points to the blocks containing the next lower-level index. Index levels are provided so that the entire index need not be searched each time that a record key is accessed. When a record key is accessed, the object-time system reads the highest-level index to find which lower-level index contains a pointer to the approximate location of that key. The block of the next lower-level index that contains the approximate location of the key is then searched. If this is the lowest-level index, it points to the first record of the data block in which the record is stored. The data block is then searched for the appropriate record key, and the record is made available. If this is not the lowest-level index, the next lower level is searched until the lowest level is reached. Figure 8-26 illustrates the search.

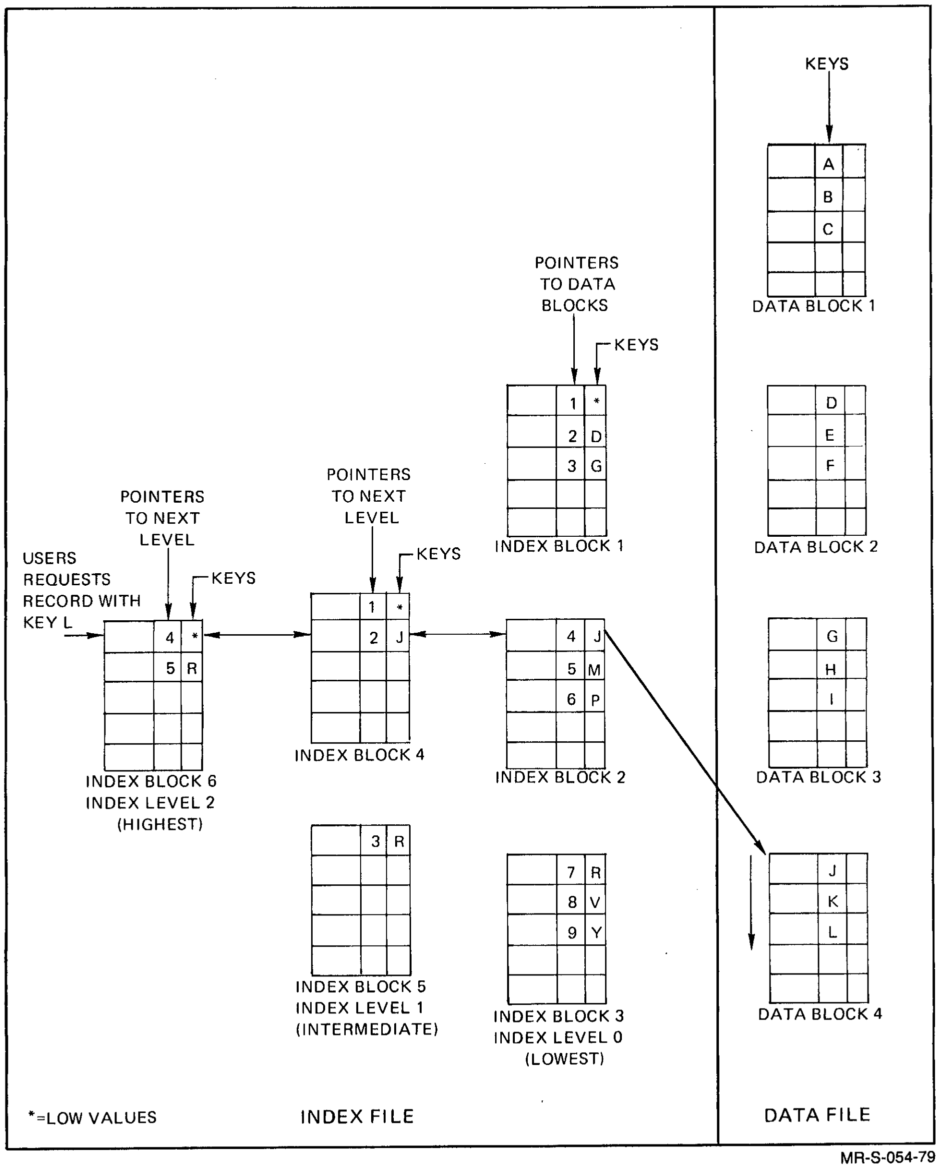


Figure 8-26 Locating a Record in an Indexed-Sequential File

The format of the index file is more complex than that of the indexed data file. Figure 8-27 shows an overview of the structure of the index file.

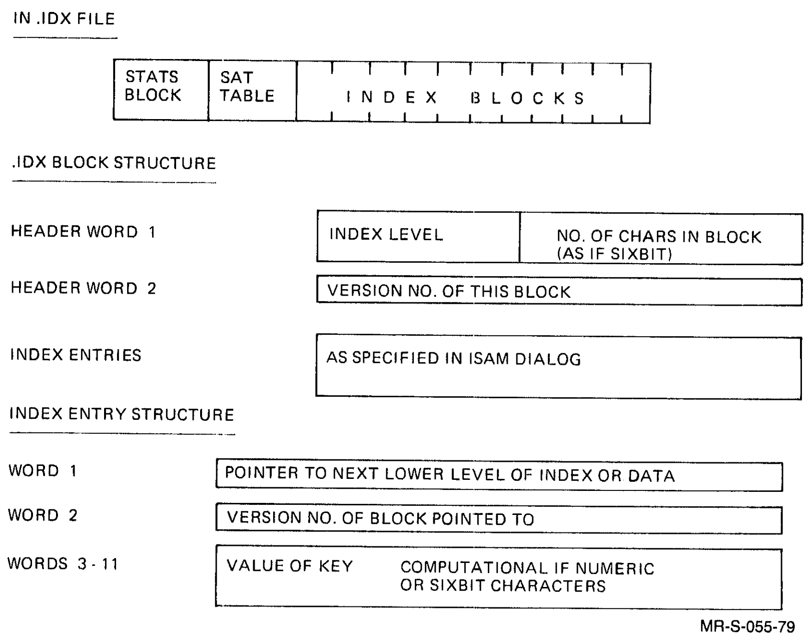

IN .IDX FILE
STATS BLOCK
SAT TABLE
INDEX BLOCKS
.IDX BLOCK STRUCTURE
HEADER WORD 1
INDEX LEVEL
NO. OF CHARS IN BLOCK (AS IF SIXBIT)
HEADER WORD 2
VERSION NO. OF THIS BLOCK
INDEX ENTRIES
AS SPECIFIED IN ISAM DIALOG
INDEX ENTRY STRUCTURE
WORD 1
POINTER TO NEXT LOWER LEVEL OF INDEX OR DATA
WORD 2
VERSION NO. OF BLOCK POINTED TO
WORDS 3 - 11
VALUE OF KEY
COMPUTATIONAL IF NUMERIC OR SIXBIT CHARACTERS
MR-S-055-79

Figure 8-27 ISAM Index File Structure

Each index block in an indexed-sequential file is written as if it were a block of a SIXBIT file. The format of the block is:

header word 1:	is the header word. The right half contains the size of the index block in characters, as if it were SIXBIT (that is, six characters per word). The left half contains a number representing the level of the index (the lowest level is 0).
header word 2:	contains the version number. This is initially set to 0 by the ISAM program, and is incremented by 1 whenever this block is divided due to the insertion of an entry when a WRITE is executed.

Following word 2 are the index entries. Each entry has the format:

word 1:	contains the pointer to a data block (if this is index level 0) or a pointer to the next lower-level index block (if this is index level 1 or higher).
word 2:	contains the version number of the index or data block to which the index entry points.

words 3-11: contain the value of a key. If the key is nonnumeric, it extends over as many words as are necessary. If the key is numeric, it is kept in COMPUTATIONAL form (even if the record key for the file is DISPLAY). It is one word if 10 or fewer digits are in the key; it is two words if 11 or more digits are in the key. If the key is COMPUTATIONAL-1 (floating point), it is one word.

NOTE

Take special care to describe your key fields in exactly the same way in both the ISAM program and your COBOL program. For example, if you describe your key field as S9(10) DISPLAY to ISAM, you should describe it the same way in your COBOL program. By using the same descriptions you ensure that appropriate key matches are made, and that the same amount of storage is generated in both the ISAM file and its record area in memory.

Within the index file, in addition to the index blocks, are two other blocks: the statistics block and the storage allocation table. The statistics block is a header containing all the necessary information about the index file and the indexed data file. Included in these statistics are: the name and extension of the data file, the number of levels in the index, the blocking factor, and a description of the record key. The storage allocation table is a bit table that shows which data blocks are in use and which are free. There are as many blocks of this table as are necessary to contain this information.

In constructing an ISAM file you must consider not only optimizing disk block usage, but also the number of levels of indexes in the index file. An indexed-sequential file should be constructed so that it does not require more than three levels of index, because the more levels of index there are, the slower the access of the data will be. Indeed, it is usually a simple matter to restrict a file of moderate size to two levels of index. For example, if the maximum file is to be 200,000 records, the blocking of the indexed data file could be 20 records per block and that of the index file 100 entries per block. Since

$$100*100*20 = 200,000$$

the file never needs more than two levels of index if it is occasionally maintained using the ISAM program. (See Section 7.1)

CHAPTER 9

SIMULTANEOUS UPDATE

The COBOL-68 simultaneous update facility allows sequential, random, or indexed-sequential data files to be updated concurrently by two or more running jobs. That is, it is possible for several truly independent jobs to modify, insert, and delete records in the same data files without loss of information or file integrity. Simultaneous update, under the control of COBOL-68, allows multiple users to share resources at the file level while having exclusive control of a portion of that resource at the record level.

You should also refer to Part 2 of this manual, COBOL-68 Language Reference Material, for the simultaneous update features of the OPEN, RETAIN, and FREE statements. To declare in your program that a file is being processed concurrently with other programs, use the appropriate syntax available with the OPEN statements. (See Section 9.1.1, The OPEN Statement.) The OPEN statement identifies the file as being open for simultaneous update and excludes most users from accessing it until you are ready to release it. However, users who attempt to open a file for READ access only, can open the file even if you already have it open under simultaneous access. The file is not released until you expressly close it by issuing a CLOSE statement.

To gain exclusive control of individual records within the file, use the RETAIN statement. (See Section 9.1.2, The RETAIN Statement.) This statement inhibits any other user from accessing the retained records until you have finished processing them. Records can be released either:

- Explicitly, by issuing a FREE statement (see Section 9.1.3, The FREE Statement)
- Implicitly, by exhausting the verb selection specified on the preceding RETAIN statement

You are advised to make careful use of the RETAIN statement in order to avoid the two most common problems that can occur using simultaneous update. The first, buried update, occurs when two users are updating the same record concurrently and one user's update is overlaid by the other's. (See Figure 9-1, The Problem of Buried Update.) The second is deadly embrace. It occurs when two users make conflicting demands upon the file resources and neither is willing or able to yield to the other. This results in both users being stalled waiting for the other to relinquish control. (See Figure 9-2, The Problem of Deadly Embrace.) Both of these problems can be avoided by carefully declaring the resources needed with a RETAIN statement prior to performing any I/O operations on a shared file.

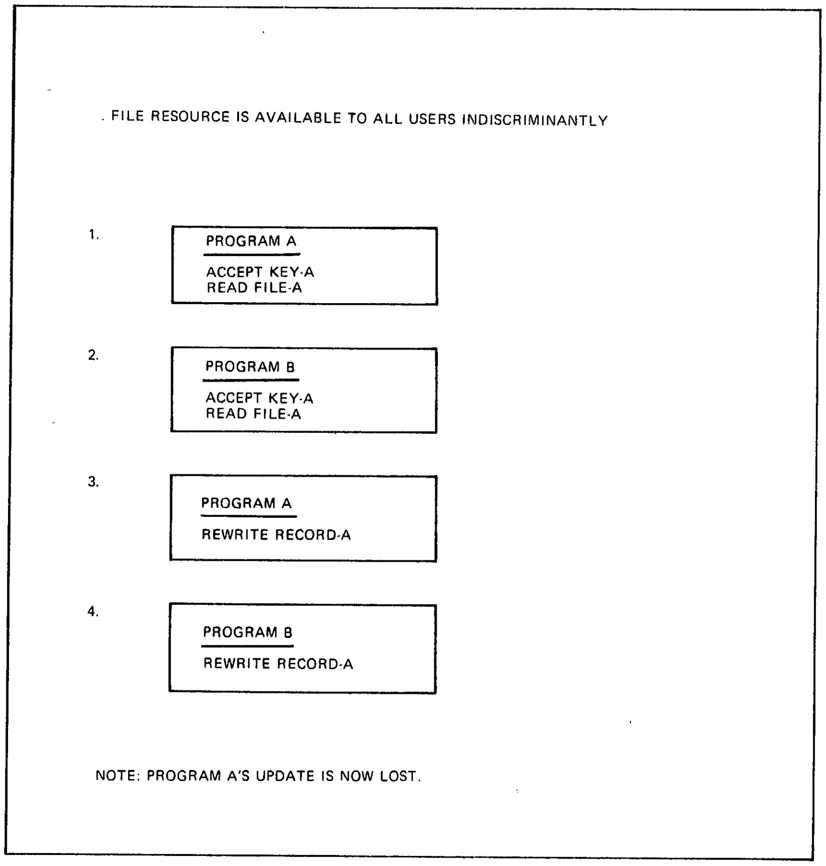


Figure 9-1 The Problem of Buried Update

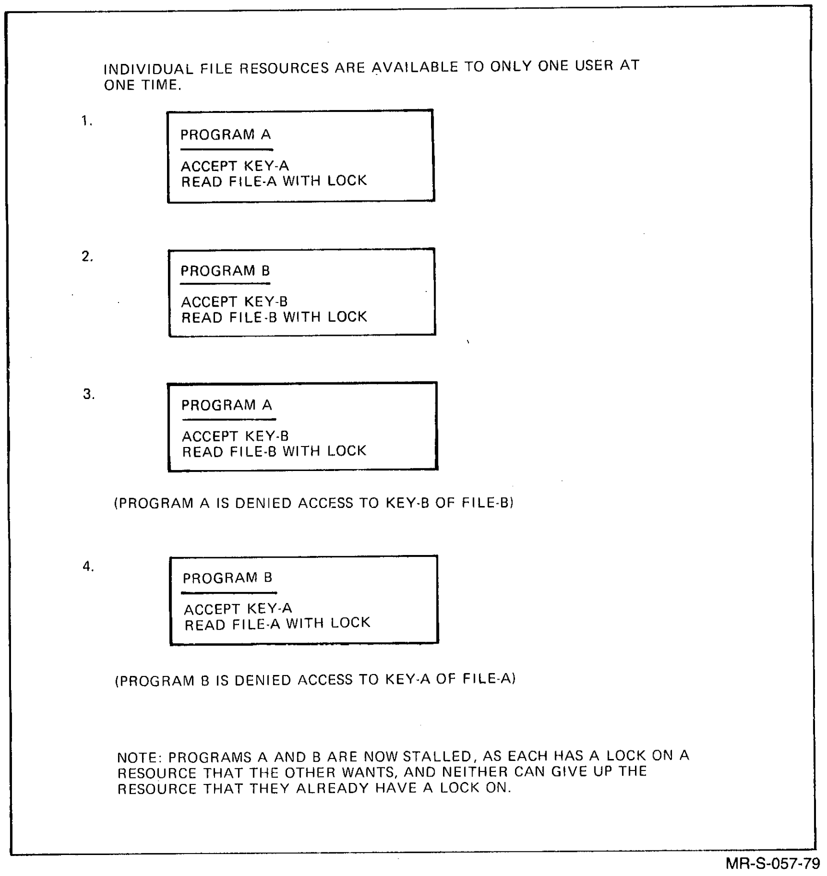

INDIVIDUAL FILE RESOURCES ARE AVAILABLE TO ONLY ONE USER AT ONE TIME.
1.
PROGRAM A
ACCEPT KEY-A
READ FILE-A WITH LOCK
2.
PROGRAM B
ACCEPT KEY-B
READ FILE-B WITH LOCK
3.
PROGRAM A
ACCEPT KEY-B
READ FILE-B WITH LOCK
(PROGRAM A IS DENIED ACCESS TO KEY-B OF FILE-B)
4.
PROGRAM B
ACCEPT KEY-A
READ FILE-A WITH LOCK
(PROGRAM B IS DENIED ACCESS TO KEY-A OF FILE-A)
NOTE: PROGRAMS A AND B ARE NOW STALLED, AS EACH HAS A LOCK ON A RESOURCE THAT THE OTHER WANTS, AND NEITHER CAN GIVE UP THE RESOURCE THAT THEY ALREADY HAVE A LOCK ON.
MR-S-057-79

Figure 9-2 The Problem of Deadly Embrace

9.1 PROGRAMMING CONSIDERATIONS

Simultaneous update allows you to declare the usage you want at both the file and record level. It also allows you to declare the usage you are allowing others to have while you have control of the file. A central clearing house in the COBOL-68 object-time system correlates these projections and takes one of three actions with respect to the intent of each user:

- Allows the process to proceed
- Suspends the process until the required resource is available
- Returns with a message to the effect that the process cannot proceed at this time

You declare file usage by specifying which of the COBOL-68 input/output verbs you execute during your tenure of the file or record and which you allow others to execute. Once allowed to proceed, you are bound by the object-time system to act within the scope of your projections and are stopped if you attempt to do otherwise. For example, if you open a file for a read operation and then issue a write you are stopped from doing so. See Figure 9-3 for an outline of how resources can be declared for simultaneous update.

```
PROCEDURE DIVISION.
BEGIN-PARAGRAPH.
     OPEN I-O FILE-NAME-1 FOR [verb selection]    (File-wide spec-
        ALLOWING OTHERS [verb selection]          ification of
        ....                                      resources)
        UNAVAILABLE [Object statements].


LOOP-PARAGRAPH.
     [Generate key values for records to be       (Specification
        retained]                                 of record re-
     RETAIN FILE-NAME-1 RECORD KEY ...            sources to be
        FOR [verb selection]                      retained and
        UNTIL FREED                               manipulated
        ...                                       within the
        UNAVAILABLE [Object statement].           context of a
                                                  user-defined
     I-O verb selection as appropriate.           transaction)
        Including READ, WRITE, DELETE,
        REWRITE.

     FREE [appropriate file records].
     GO TO LOOP-PARAGRAPH.
END-OF-JOB.
     CLOSE FILE-NAME-1 ...                        (Release of
                                                  file-wide
                                                  resource)
```

Figure 9-3 Declaring Resources For Simultaneous Update

9.1.1 The OPEN Statement

The OPEN statement is the vehicle by which you declare a file is being used for simultaneous update. It allows you to specify:

- Your projected usage of the file in terms of the I/O operations you wish to perform

- The projected usage you are willing to allow others in terms of the I/O operations they are allowed to perform

Figure 9-4 shows the general format of the OPEN statement.

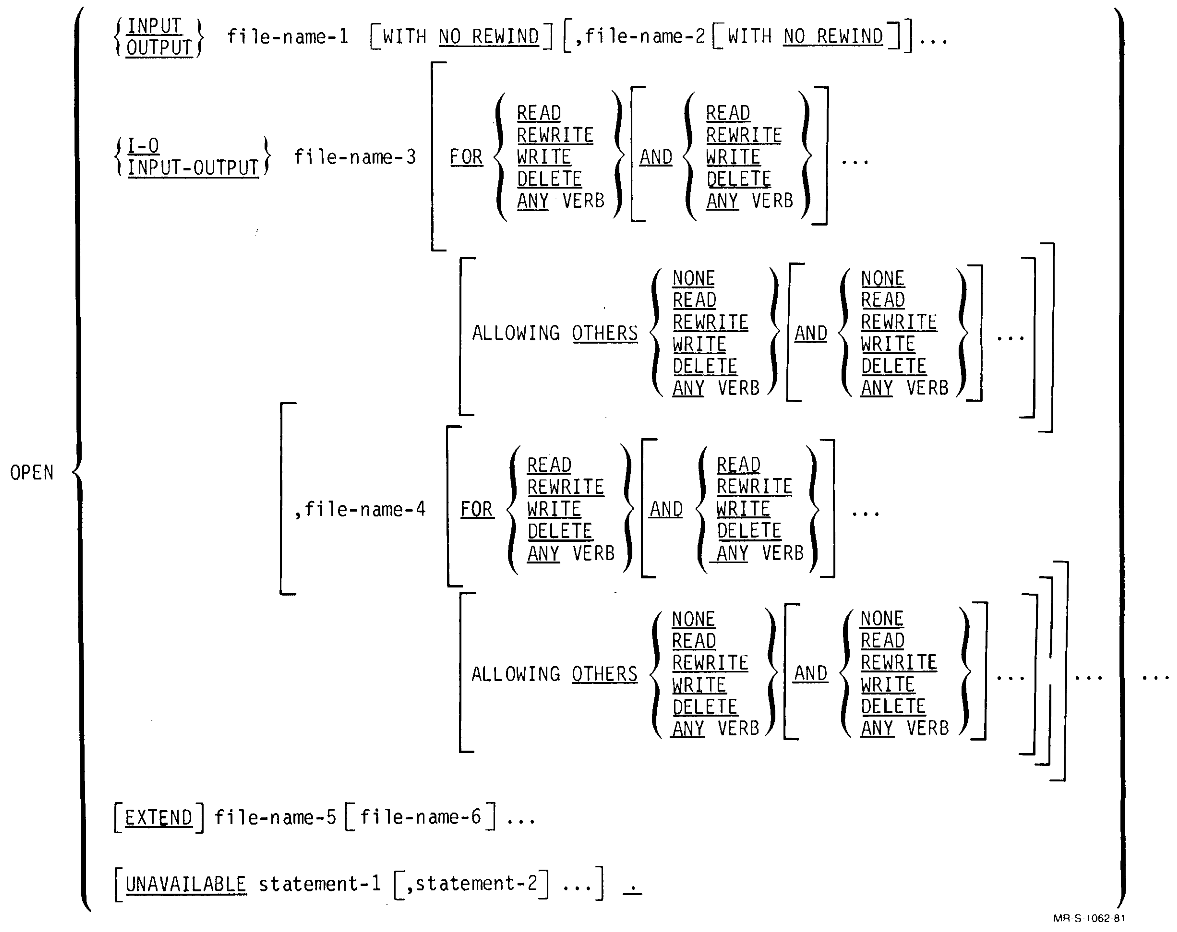

OPEN
INPUT
OUTPUT
file-name-1 WITH NO REWIND ,file-name-2 WITH NO REWIND ...
I-O
INPUT-OUTPUT
file-name-3
FOR
READ
REWRITE
WRITE
DELETE
ANY VERB
AND
ALLOWING OTHERS
NONE
,file-name-4
EXTEND file-name-5 file-name-6 ...
UNAVAILABLE statement-1 ,statement-2 ...
MR-S-1062-81

Figure 9-4 The OPEN Statement

The following rules apply to the use of an OPEN statement for files being processed under simultaneous update:

1. To open a file under simultaneous update, the ALLOWING OTHERS clause must be specified.

2. Every user, that is, every program expecting to process the file concurrently, must open the file either under simultaneous update or for input only. Other users are denied access.

NOTE

File access is determined on a first come first served basis. Therefore, if the first user opens a file for simultaneous update, all others must likewise open it under simultaneous update - unless the user who is not under simultaneous update wishes to open the file for READ access only. Conversely, if a file is open for normal processing, users attempting to open it under simultaneous update are denied access. See Figure 9-5, Competing For Program Access to Files.

3. The file must be OPEN in I/O mode.

4. The COBOL-68 I/O verbs you intend to execute must be entered following the key word FOR.

5. The COBOL-68 I/O verbs you are willing to allow others to execute must be entered following the key words ALLOWING OTHERS.

6. All files to be opened for simultaneous update must be opened in the same OPEN statement. Multiple OPEN statements for simultaneous update are not allowed. Therefore, before another file can be opened for simultaneous update, the previously opened files must be closed. This prevents deadly embrace at the file level.

7. You can use the same OPEN statement to open files for simultaneous update as well as for normal processing.

8. A maximum of sixteen (16) files can be opened by a single OPEN statement. In fact, a maximum of sixteen files can be open at once, regardless of the number of OPEN statements involved in opening them.

9. If one or more of the files being opened for simultaneous update is not available in the mode specified, the program requesting the OPEN is suspended until the requested file is available. Those files, if any, that were opened during the process remain open. Control is not returned to the program until all of the requested files are open. If the UNAVAILABLE clause is specified, no file is opened, even though available, until all of the requested files are available. In this case, the statements following the UNAVAILABLE clause are executed.

10. The I/O verbs specified in the OPEN statement are the only verbs that can be used to process the file. Likewise, the I/O verbs you allow others to use are the only ones available to them. Any attempt to use verbs other than the ones specified causes the object-time system to abort the program.

Example 9-1

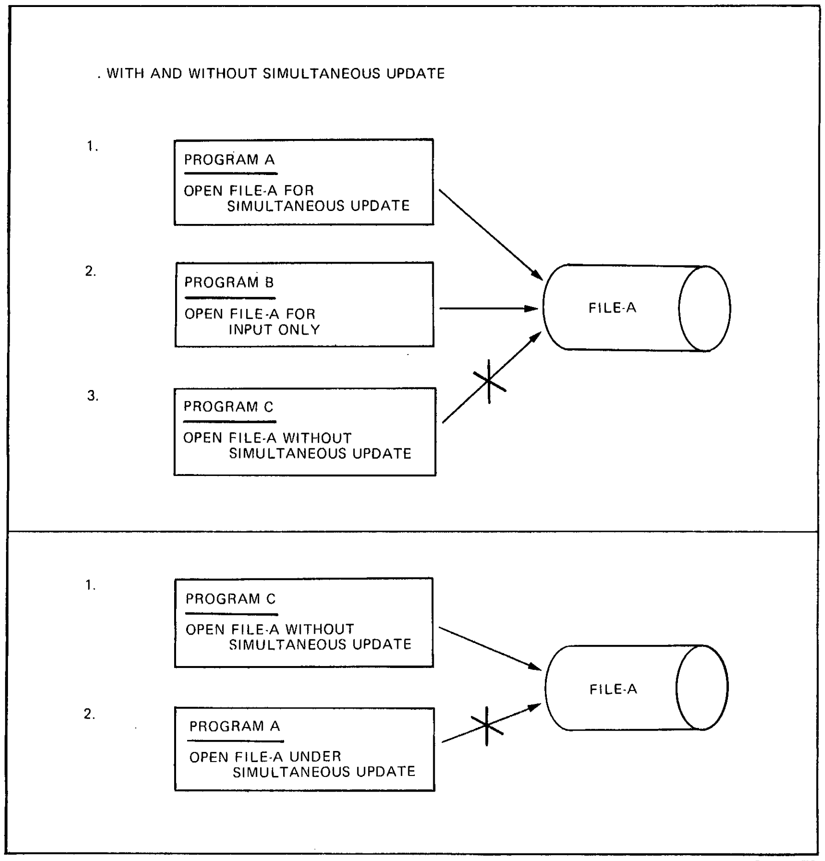
```
OPEN I/O FILE-A FOR READ AND WRITE,

   ALLOWING OTHERS READ AND WRITE.
```

Example 9-2

```
OPEN OUTPUT FILE-A, LIST,

   INPUT-OUTPUT FILE-B FOR READ AND REWRITE,
                       OTHERS ANY

                FILE-C FOR READ,
                       OTHERS READ AND REWRITE,

                FILE-D FOR ANY,
                       OTHERS NONE,

   INPUT FILE-E WITH NO REWIND,
   I-O FILE-F, FILE-G FOR WRITE.
```

Example 9-3

```
     OPEN I-O FILE-A FOR READ AND WRITE,
                     OTHERS ANY,

        UNAVAILABLE OPEN I-O FILE-A FOR READ,
                     OTHERS ANY,
                     UNAVAILABLE STOP RUN.
```



. WITH AND WITHOUT SIMULTANEOUS UPDATE
1.
PROGRAM A
OPEN FILE-A FOR SIMULTANEOUS UPDATE
2.
PROGRAM B
OPEN FILE-A FOR INPUT ONLY
3.
PROGRAM C
OPEN FILE-A WITHOUT SIMULTANEOUS UPDATE
FILE-A
1.
PROGRAM C
OPEN FILE-A WITHOUT SIMULTANEOUS UPDATE
2.
PROGRAM A
OPEN FILE-A UNDER SIMULTANEOUS UPDATE
FILE-A
MR-S-058-79

Figure 9-5 Competing For Program Access to Files

9.1.2 The RETAIN Statement

The RETAIN statement allows you to gain exclusive control of individual records within a file that was previously opened for simultaneous update. Figure 9-6 shows the general format of the RETAIN statement.

```
RETAIN file-name-1 RECORD [ KEY { identifier-1 } ]
                                { literal-1    }

     { READ                [     { READ          }     ] }
     { REWRITE             [     { REWRITE       }     ] }
     { READ-REWRITE        [     { READ-REWRITE  }     ] }
 FOR { DELETE              [ AND { DELETE        } ... ] } [UNTIL FREED]
     { WRITE               [     { WRITE         }     ] }
     { READ-WRITE          [     { READ-WRITE    }     ] }
     { ANY VERB            [     { ANY VERB      }     ] }

[
 ,file-name-2 RECORD [ KEY { identifier-2 } ]
                           { literal-2    }

     { READ                [     { READ          }     ] }
     { REWRITE             [     { REWRITE       }     ] }
     { READ-REWRITE        [     { READ-REWRITE  }     ] }
 FOR { DELETE              [ AND { DELETE        } ... ] } [UNTIL FREED]  ]
     { WRITE               [     { WRITE         }     ] }
     { READ-WRITE          [     { READ-WRITE    }     ] }
     { ANY VERB            [     { ANY VERB      }     ] }

[ UNAVAILABLE statement-1 [,statement-2] ... ] .
                                                 MR-S-1068-81
```

Figure 9-6 The RETAIN Statement

The following general rules apply to the use of the RETAIN statement. For a description of how the RETAIN statement is used for the individual file types - sequential, random, indexed-sequential - see Sections 9.1.4, 9.1.5, and 9.1.6 respectively. (See also the COBOL-68 Language Reference Material, Part 2 of this manual.)

1. The file(s) named in a RETAIN statement must have been previously opened under simultaneous update. If not, the object-time system aborts the program.

2. A RETAIN statement must be given before any record on a file opened for simultaneous update can be accessed.

3. You can use the same RETAIN statement to reserve records on sequential, random, or indexed-sequential files. The I/O verbs selected, however, must conform to those allowed for the file.

4. All records to be retained concurrently must be retained with the same RETAIN statement. Once records have been retained, no other records can be retained until the currently retained records are freed.

5. The retention of records is purely a logical operation and does not involve any actual I/O. You can, in fact, retain nonexistent records. Obviously, any attempt to read or rewrite any of these records could result in an I/O error that could cause your program to be terminated. (See note 6.)

6. A RETAIN statement, consistent with note 5, does not cause an AT END condition. This can only be caused by a READ statement. The RETAIN statement in this case merely retains a nonexistent record after the last one in the file.

7. If you retain a record for a READ operation, other users are allowed concurrent access to that record for READ. If you retain a record for any other type of I/O, all other users are denied access until you have freed it.

8. The I/O usage you specify in a RETAIN statement must agree with the usage you specified in the OPEN statement for the file. For example, if you want to retain a record for a WRITE operation, you must have specified WRITE in the OPEN statement for the file. This holds true as well for the ANY VERB option. The key words ANY VERB must appear in the OPEN statement if you want to use them in a RETAIN statement.

9. The records named in the RETAIN statement are automatically freed upon execution of the I/O verbs specified in the FOR clause. The only exceptions are:

   a. If the ANY VERB option is specified in the FOR clause, a FREE statement must be issued to release a record.

   b. If the UNTIL FREED option is specified, a FREE statement must be issued to release a record.

   NOTE

   The UNTIL FREED option allows you to retain several logically related records for processing without their being automatically freed by the I/O verbs.

   c. If an I/O verb is specified in a RETAIN statement but that verb is not executed, the record is not freed until a FREE statement is issued.

10. The KEY phrase allows you to specify a particular record or more than one record in a file.

11. The value of the key can be specified by any identifier that can be subscripted, qualified, or both. Its usage, however, must be COMPUTATIONAL. For example:

```
RETAIN FILE-A RECORD
       KEY PAY-REC OF RECORD-KEYS
       FOR READ-REWRITE.
```

It can also be a positive numeric literal containing from 1 to 10 digits. You can, for example, enter:

```
RETAIN FILE-A-RECORD
       KEY 123
       FOR READ-REWRITE.
```

12. The optional word RECORD can be used as a reminder that you are retaining records, not files. For example:

```
RETAIN FILE-A RECORD FOR READ.
```

retains the next record in FILE-A. Be aware, though, that in most cases the RETAIN statement actually locks up the block in which the requested record resides. The only time this is not true is in the case of an indexed-sequential file which is open for WRITE access; if you retain a record in this file, the entire file is locked until the record is freed. This is because all the index blocks must be locked until any new record is placed into the file.

9.1.3 The FREE Statement

The FREE statement explicitly frees records that have been retained for simultaneous update. Figure 9-7 shows the general format of the FREE statement.

```
      {              { RECORD [KEY {identifier-1}] }                               }
      { file-name-1  {              {literal-1   } }                               }
      {              { EVERY RECORD                }                               }
      {                                                                            }
FREE  {              [              { RECORD [KEY {identifier-2}] } ]              }
      {              [ ,file-name-2 {              {literal-2   } } ]              }
      {              [              { EVERY RECORD                } ]              }
      {                                                                            }
      { EVERY RECORD                                                               }

      [NOT RETAINED statement-1 [,statement-2] ...] .
                                                 MR-S-1055-81
```

Figure 9-7 The FREE Statement

The following general rules apply to the use of the FREE statement. For a description of how the FREE statement is used with the individual file types - sequential, random, and indexed-sequential - see Sections 9.1.4, 9.1.5, and 9.1.6 respectively. (See also the COBOL-68 Language Reference Material, Part 2 of this manual.)

1. The FREE statement is required to explicitly release records that have not been implicitly released by an I/O statement. This could occur when:

   a. The RETAIN statement contains the UNTIL FREED phrase

   b. An I/O statement is not issued after the RETAIN statement

   c. The FOR clause of the RETAIN statement specifies ANY VERB

2. The EVERY RECORD phrase allows you to free all of the records retained or just those of a particular file. It saves you from having to issue a separate FREE statement for every record that was retained.

3. When the EVERY RECORD phrase is used, the NOT RETAINED condition occurs only if no records are currently retained or if no records in a specific file are retained.

4. The NOT RETAINED phrase specifies the COBOL statements to be executed in the event that one or more of the record(s) you are attempting to free have not been retained. If this phrase is not specified, the program continues and you are not notified of any possible error.

5. A FREE statement issued to a file that was not opened for simultaneous update causes the statements following the NOT RETAINED phrase, if present, to be executed. If the NOT RETAINED phrase was not specified in this case, the program continues and you are not notified of a possible error condition.

6. A single FREE statement can be used to free records retained from all open files, regardless of file type.

7. All records, regardless of how they were retained, are automatically freed when the file is closed.

9.1.4 Accessing Sequential Files

The following sections describe how to use the RETAIN and FREE statements to access records in a sequential file.

9.1.4.1 Basic Reading

9.1.4.1 Basic Reading - The simplest way to read a sequential file opened for simultaneous update is to execute pairs of statements like this:

```
    RETAIN FILE-A FOR READ.

    READ FILE-A AT END GO TO EOJ.
```

The RETAIN statement declares your intent to read the next record of FILE-A. The READ statement delivers the next record to the file's record area in memory, and automatically frees it for use by other users.

9.1.4.2 **Basic Writing** - Basic writing of a sequential file opened for simultaneous update is analogous to basic reading. For example, you could use code that looks like this:

```
     RETAIN FILE-A FOR WRITE.

     (process record)

     WRITE FILE-A-RECORD.
```

In this case, FILE-A-RECORD is written out to FILE-A and automatically freed for access by other users.

9.1.4.3 **Basic Updating** - To update the next record in a file open for simultaneous upate, you can use statements that look like this:

```
     RETAIN FILE-A FOR READ-REWRITE.

     READ FILE-A AT END GO TO EOJ.
               .
               .
               .
               .
               .
     REWRITE FILE-A-RECORD.
```

FILE-A-RECORD is automatically released upon execution of the REWRITE statement because both verbs named in the RETAIN statement have been executed. If only one or none of the verbs were executed, the record would not have been freed and any attempt to RETAIN any other records would fail.

If, however, your application is such that you can or can not want to update a record once it has been read, code of this nature could be used:

```
     RETAIN FILE-A FOR READ-REWRITE.

     READ FILE-A AT END GO TO EOJ.
               .
               .
               .
               .
     IF CHANGED REWRITE FILE-A-RECORD
        ELSE FREE FILE-A.
```

9.1.4.4 **Access to Sequential File Strategies** - There are two reasons why the basic reading, writing, and updating of sequential files as outlined in Sections 9.1.4.1, 9.1.4.2, and 9.1.4.3 are not sufficient for some applications:

1. Performance
2. Logically related records

Each time you retain a record and that record happens to be already in your buffer, it is necessary to refill that buffer from mass storage to make sure that you have the very latest copy. Similarly, each time a record that you have written or rewritten is implicitly or explicitly freed, you must be certain that it is the very latest copy, and that no other user has updated that record in the interim. These considerations have little effect on the performance of random or indexed-sequential files accessed randomly, but the effect on sequentially processed files is profound. Processing a file with a blocking factor of ten, as suggested in Sections 9.1.4.1, 9.1.4.2, or 9.1.4.3, would require an order of magnitude more input/output overhead than it would if you were not using simultaneous update mode. This is the performance reason for using more sophisticated coding techniques. Sometimes, several records in a file are logically related and must be updated together. For example, a header record and subsequent trailer records might be logically related in such a way that the trailer records cannot be changed unless the header record remains static. But with the basic techniques outlined in Sections 9.1.4.1, 9.1.4.2, and 9.1.4.3, only a single record can be retained at a time. This is the logically-related-records reason for more sophisticated coding techniques.

The first step in providing for more sophisticated code is the introduction of a notation for addressing the records of a sequential file. The notation is this: record 0 is defined as the next record to be read or written. Records 1, 2, 3, through n are defined relative to record 0.

NOTE

If you have just written a record, the next record to be written is the one following it. If you have just read a record, however, the next record to be written is the one just read. Therefore, if you have just read a record and then you retain record 0 for WRITE, you have in effect retained the record just read. If, however, you have just read a record and then you retain record 0 for READ-WRITE, you have effectively retained the next record in the file.

Sequential file users should code for performance by retaining several records at a time. Performance is optimal if the number of records retained is a multiple of the blocking factor and the execution of the RETAIN statement is synchronized with logical block boundaries. A RETAIN statement for a file whose blocking factor is 5 might look like this:

```
    RETAIN FILE-A KEY 0 FOR READ,

           FILE-A KEY 1 FOR READ,

           FILE-A KEY 2 FOR READ,

           FILE-A KEY 3 FOR READ,

           FILE-A KEY 4 FOR READ.
```

This would then be followed by READ and/or FREE statements until all records have been freed. Subsequent FREE statements use the same notation for freeing records as was used for retaining them. Thus:

```
     RETAIN FILE-A KEY 0 FOR READ,

            FILE-A KEY 1 FOR READ.

     READ FILE-A AT END GO TO EOJ.
                 .
                 .
                 .
     FREE FILE-A KEY 1.
```

causes the second record of the pair to be freed, not the next one in the file.

Providing a notation for referencing several records of a sequential file is not enough for updating several logically related records together. It is also necessary to retain a record, even though you are through with it, until all of the related records have been processed. The UNTIL FREED phrase is provided for this purpose. It allows you to bypass the automatic freeing of records and retain them until you are ready to expressly free them. Also, to facilitate the freeing of multiple records, the EVERY RECORD phrase is provided. It allows you to free every record retained or every record in a particular file. Thus, to update three logically related records in a particular file, you can code:

```
     RETAIN FILE-A KEY 0 FOR READ-WRITE
                             UNTIL FREED,

            FILE-A KEY 1 FOR READ-WRITE
                             UNTIL FREED

            FILE-A KEY 2 FOR READ-WRITE.

     READ FILE-A AT END GO TO EOJ.
                 .
                 .
                 .
     WRITE FILE-A-RECORD.

     READ FILE-A AT END GO TO EOJ.
                 .
                 .
                 .
     WRITE FILE-A-RECORD.

     READ FILE-A AT END GO TO EOJ.
                 .
                 .
                 .
     WRITE FILE-A-RECORD.

     FREE FILE-A EVERY RECORD.
```

You could also use the ANY VERB phrase to accomplish the same results. For example:

```
     RETAIN FILE-A KEY 0 FOR ANY VERB
```

results in your having to expressly free the record when you have finished with it.

When retaining records, the program is normally suspended if any of the requested files or records are unavailable. You are not notified of this suspension unless you have provided the UNAVAILABLE phrase as part of the RETAIN statement. The UNAVAILABLE phrase allows you to specify a procedure to be followed in the event a record or file is unavailable at the time your program attempts to access it. For example:

```
RETAIN FILE-A KEY 0 FOR ANY VERB
       UNAVAILABLE PERFORM UNAVAIL-RTN.
```

This instructs the object-time system to execute the statement following the word UNAVAILABLE in the event that the file (FILE-A) or the next record in the file is unavailable at the time the RETAIN statement is executed.

Similarly, if you execute a FREE statement for a record or records that are not currently retained by your program, the object-time system proceeds to the next instruction in your program as though the condition did not exist. If you wish to be informed of this condition, you must provide the NOT RETAINED phrase in the FREE statement. The NOT RETAINED phrase causes the object-time system to execute the procedures immediately following the words NOT RETAINED. A FREE statement of this kind might look like this:

```
FREE FILE-A KEY 0 NOT RETAINED
                  GO TO ERROR-RTN.
```

9.1.5 Accessing Random Files

Accessing records in a random file is similar to the accessing of sequential file records. (See Section 9.1.4.) There are, however, these differences:

1. If a key is not specified, the ACTUAL KEY specified in the FD for the file is used.

2. Positive keys, whether specified directly or with ACTUAL KEY, designate fixed (absolute) records of the file (as opposed to designating records relative to the current record). Thus, record 1 is always the first record of the file, not the next record. A zero key, on the other hand, is interpreted in the same way as for sequential files: record 0 is defined as the next record to be read or written.

3. A RETAIN statement, by virtue of its not performing any actual I/O, cannot generate an INVALID KEY condition.

Example 9-4 demonstrates reading a random file sequentially.

Example 9-4

```
A.  MOVE ZERO TO FILE-A-KEY.

    RETAIN FILE-A FOR READ.

    READ FILE-A RECORD;  INVALID KEY GO TO ERROR-RTN.
               .
               .
               .
               .
    GO TO A.
```

Example 9-5 shows how a file can be processed randomly. Note that the UNTIL FREED clause is used to insure that no one can access the record until it is written.

Example 9-5

```
A.  PERFORM RANDOM-KEY-GENERATION.

    RETAIN FILE-A KEY GENERATED-KEY
           FOR READ-WRITE UNTIL FREED.
    READ FILE-A INVALID KEY GO TO ERR-RTN.
              .
              .
              .
    WRITE FILE-A-RECORD.
    FREE FILE-A RECORD.
    GO TO A.
```

Example 9-6 shows how to use a field within a record as the ACTUAL KEY for processing a chain of related records in a random file. Procedure A initializes processing with record number 64. Procedure B insures that record 64 is stable, that is, that it has not been changed by some other user after you read it and that it is not changed while you are processing it.

Example 9-6

```
A.  MOVE 64 TO FILE-A-REL-KEY.

    RETAIN FILE-A FOR READ.

    READ FILE-A INVALID KEY GO TO ERR-RTN.
              .
              .
              .
B.  RETAIN FILE-A FOR READ-REWRITE

           FILE-A KEY NUMBER OF FILE-A-RECORD
                  FOR READ-REWRITE.

    READ FILE-A INVALID KEY GO TO ERR-RTN.

    IF (record not stable) FREE FILE-A EVERY RECORD.
                           GO TO B.

C.  (process record 64 and record pointed to by NUMBER)
```

9.1.6 Accessing Indexed-Sequential Files

Accessing records in an indexed-sequential file is similar to the accessing of sequential file records. (See Section 9.1.4.) There are, however, these differences:

1. You can retain records for REWRITE, DELETE, and READ-REWRITE, in addition to READ, WRITE, and ANY VERB. You can not retain a record for READ-WRITE.

2. If no key is specified, the RECORD KEY defined in the SELECT statement for the file is used.

3. If a key is supplied, it must be specified with an identifier that agrees with the file's RECORD KEY in size, class, usage, and number of decimal places. The only exception is a key whose usage is COMP; in this case, a positive numeric literal of ten or fewer digits can be used.

4. Retaining or freeing records does not affect the "remembered" key of the file; that is, the record which would be read by a READ NEXT statement would be the same before and after a RETAIN or a FREE statement.

Example 9-7 demonstrates how an indexed-sequential file can be processed sequentially.

Example 9-7

```
A.  RETAIN FILE-A KEY FILE-A-KEY
           FOR READ.

    READ FILE-A NEXT RECORD;  INVALID KEY GO TO ERR-RTN.
                  .
                  .
                  .
    GO TO A.
```

Example 9-8 shows the random processing of an indexed file. Note how the UNTIL FREED statement is used to insure the stability of the record.

Example 9-8

```
A.  ACCEPT DATA-KEY.

    RETAIN FILE-A KEY DATA-KEY
           FOR READ-REWRITE UNTIL FREED.

    READ FILE-A INVALID KEY GO TO ERR-RTN.

    DISPLAY FILE-A-RECORD.

B.  (process and update record if you wish)
                  .
                  .

C.  FREE FILE-A-RECORD.
                  .
                  .
    GO TO A.
```

CHAPTER 10

REPORT WRITER

The COBOL compiler offers a report writing facility, REPORT WRITER. Using this facility can make it easy to format printed reports.

The example program on the following pages shows how to use the major features of REPORT WRITER. The full formats and available options for each statement are discussed in detail in the COBOL-68 Language Reference Material, Chapter 4 of this manual.

```
P R O G R A M   R E P E X M                 COBOL-68 12B(1115) BIS
                  15-JAN-81  15:19                  PAGE 1
REP1.CBL    15-JAN-81  15:20

0001    ID DIVISION.
0002    PROGRAM-ID. REPEXM.
0003
0004  * **************************************************************
0005  *                                                              *
0006  *  This program is an example of the use of REPORT WRITER.     *
0007  *                                                              *
0008  *  The program generates two reports: one is a list of         *
0009  *  customers by city and state; the other is a list of         *
0010  *  totals for each state.                                      *
0011  *                                                              *
0012  *  The two reports are generated at one time and into one      *
0013  *  file.  The line printer spooler can separate them at the    *
0014  *  time they are to be printed.                                *
0015  *                                                              *
0016  * **************************************************************
0017
0018    ENVIRONMENT DIVISION.
0019    CONFIGURATION SECTION.
0020    SPECIAL-NAMES.
0021
0022  * **************************************************************
0023  *                                                              *
0024  *  Report Codes (Line 37)                                      *
0025  *                                                              *
0026  *  The following entry in the SPECIAL-NAMES paragraph of the   *
0027  *  CONFIGURATION SECTION defines the codes 'A' and 'B' for     *
0028  *  the two reports we are going to generate.  The line printer *
0029  *  spooler can separate them when we use the /REPORT switch    *
0030  *  with the system QUEUE command.  For example, to print       *
0031  *  both reports, we would use                                  *
0032  *                                                              *
0033  *        Q LL:=CUSTMR.LPT/REPORT:A,CUSTMR.LPT/REPORT:B         *
0034  *                                                              *
0035  * **************************************************************
0036
0037          'A' IS BY-CITY-CODE;'B' IS STATE-TOTALS-CODE.
0038
0039    INPUT-OUTPUT SECTION.
0040    FILE-CONTROL.
0041          SELECT CUSTOMER-FILE; ASSIGN TO DSK;
0042                  RECORDING MODE IS ASCII.
```

```
0043      P R O G R A M   R E P E X M                    COBOL-68 12B(1115) BIS
                          15-JAN-81  15:19                     PAGE 2
REP1.CBL    15-JAN-81  15:20

0044  * *************************************************************
0045  *                                                             *
0046  *  Report file SELECTion and ASSIGNment (Line 55)             *
0047  *                                                             *
0048  *  Like any file, the file for the report must be SELECTed and *
0049  *  ASSIGNed.  Here we're using a disk file, but any device is  *
0050  *  legal.                                                     *
0051  *                                                             *
0052  * *************************************************************
0053
0054
0055            SELECT PRINTER-FILE; ASSIGN TO DSK;
0056                    RECORDING MODE IS ASCII.
0057
0058            SELECT SORT-FILE; ASSIGN TO DSK,DSK,DSK;
0059                    RECORDING MODE IS ASCII.
0060
0061    DATA DIVISION.
0062    FILE SECTION.
0063
0064    SD      SORT-FILE.
0065    01      SORT-RECORD.
0066            02 SORT-NAME          PIC X(24) USAGE DISPLAY-7.
0067            02 SORT-CITY          PIC X(20) USAGE DISPLAY-7.
0068            02 SORT-STATE         PIC XX USAGE DISPLAY-7.
0069            02 SORT-STREET        PIC X(20) USAGE DISPLAY-7.
0070            02 SORT-SALES         PIC S9(10) USAGE COMP.
0071
0072    FD      CUSTOMER-FILE
0073            VALUE OF IDENTIFICATION IS 'CUSTMRDAT'.
0074    01      CUSTMR-RECORD.
0075            02 CUSTMR-NAME        PIC X(24) USAGE DISPLAY-7.
0076            02 CUSTMR-STREET      PIC X(20) USAGE DISPLAY-7.
0077            02 CUSTMR-CITY        PIC X(20) USAGE DISPLAY-7.
0078            02 CUSTMR-STATE       PIC XX USAGE DISPLAY-7.
0079            02 CUSTMR-SALES       PIC S9(10)V99.
0080            02 FILLER             PIC X(302).
```

```
0081      P R O G R A M   R E P E X M                  COBOL-68 12B(1115) BIS
                              15-JAN-81  15:19                  PAGE 3
REP1.CBL    15-JAN-81  15:20

0082   * *************************************************************
0083   *                                                             *
0084   *  The FD for the Report File (Lines 100 - 103)               *
0085   *                                                             *
0086   *  Here we give the file for the report the name CUSTMR.LPT.  *
0087   *                                                             *
0088   *  The REPORTS ARE clause names the RD entries that we'll     *
0089   *  define in the REPORT SECTION and names the reports to be   *
0090   *  written in the file.                                       *
0091   *                                                             *
0092   *  The record named in the 01-level entry must be large enough *
0093   *  to contain the largest line written (including a 1-character*
0094   *  code.  In our example, we never refer to PRINTER-RECORD in  *
0095   *  the PROCEDURE DIVISION, so we could omit this; the default  *
0096   *  size for PRINTER-RECORD is 132 characters.                 *
0097   *                                                             *
0098   * *************************************************************
0099
0100     FD      PRINTER-FILE;
0101             REPORTS ARE STATE-TOTALS-ONLY,BY-CITY
0102             VALUE OF IDENTIFICATION IS 'CUSTMRLPT'.
0103     01      PRINTER-RECORD      PIC X(70) USAGE DISPLAY-7.
0104
0105     WORKING-STORAGE SECTION.
0106
0107     01      THIS-DATE           PIC X(8).
0108     01      TD-REDEFINED        REDEFINES THIS-DATE.
0109             02 TD-MONTH         PIC Z9.
0110             02 TD-HYF-1         PIC X.
0111             02 TD-DAY           PIC 99.
0112             02 TD-HYF-2         PIC X.
0113             02 TD-YEAR          PIC 99.
0114
0115     01      UNEDITED-DATE.
0116             02 UE-YEAR          PIC 99.
0117             02 UE-MONTH         PIC 99.
0118             02 UE-DAY           PIC 99.
0119             02 FILLER           PIC X(6).
0120
0121     77      TEMP PIC S999 USAGE COMP.
0122     77      NR-OF-CITIES PIC S999 USAGE COMP.
0123     77      NR-OF-STATES PIC S999 USAGE COMP.
0124
0125     77      ONE-COUNT           PIC S9 USAGE COMP VALUE 1.
0126     77      CURRENT-STATE       PIC XX.
0127     77      CURRENT-CITY PIC X(20) USAGE DISPLAY-7.
```

```
0128       P R O G R A M   R E P E X M               COBOL-68 12B(1115) BIS
                             15-JAN-81  15:19               PAGE 4
REP1.CBL    15-JAN-81  15:20

0129   * ***************************************************************
0130   *                                                               *
0131   *  The REPORT SECTION Statement (Line 139)                      *
0132   *                                                               *
0133   *  The REPORT SECTION is in the DATA DIVISION.  It must be the *
0134   *  last section of the division.  In the REPORT SECTION, we     *
0135   *  define the formats for the reports.                          *
0136   *                                                               *
0137   * ***************************************************************
0138
0139     REPORT SECTION.
0140
0141   * ***************************************************************
0142   *                                                               *
0143   *  The RD for a Report (Lines 160 - 453)                        *
0144   *                                                               *
0145   *  The RD is the report description for each report.  We need  *
0146   *  an RD for each report; one is here and the other is below.  *
0147   *                                                               *
0148   *  The CODE clause of the RD gives the mnemonic-name of the     *
0149   *  code assigned to the report.  This is the same code given    *
0150   *  by the literal in the SPECIAL-NAMES paragraph of the         *
0151   *  ENVIRONMENT DIVISION above.                                  *
0152   *                                                               *
0153   *  The CONTROL clause specifies the break fields in order from *
0154   *  most important to least important.  FINAL is a special case *
0155   *  in which a control break occurs at the end of the           *
0156   *  report.                                                      *
0157   *                                                               *
0158   * ***************************************************************
0159
0160     RD      STATE-TOTALS-ONLY
0161             CODE STATE-TOTALS-CODE
0162             CONTROLS ARE FINAL, SORT-STATE.
```

```
0163      P R O G R A M   R E P E X M                  COBOL-68 12B(1115) BIS
                                 15-JAN-81  15:19              PAGE 5
REP1.CBL     15-JAN-81  15:20

0164   * ***************************************************************
0165   *                                                               *
0166   *  The TYPE Statement (Line 266 and throughout the RDs)         *
0167   *                                                               *
0168   *  The TYPE statement defines the type of each record and       *
0169   *  where it appears in the report.  The record need not be      *
0170   *  named unless it is referenced in the PROCEDURE DIVISION.     *
0171   *                                                               *
0172   *  There are seven types of records:                            *
0173   *                                                               *
0174   *     REPORT HEADING (or RH) is a heading that appears at       *
0175   *         the beginning of the report.                          *
0176   *                                                               *
0177   *     REPORT FOOTING (or RF) is a footing that appears at       *
0178   *         the end of the report.                                *
0179   *                                                               *
0180   *     PAGE HEADING (or PH) is a page heading that appears       *
0181   *         at the top of each page of the report.                *
0182   *                                                               *
0183   *     PAGE FOOTING (or PF) is a page footing that appears       *
0184   *         at the bottom of each page.                           *
0185   *                                                               *
0186   *     CONTROL HEADING (or CH) is a heading that appears         *
0187   *         immediately before any detail lines whenever a        *
0188   *         control break occurs, and after the page heading of   *
0189   *         the first page.  The name of the control break is     *
0190   *         specified in the CONTROL clause, and tells REPORT     *
0191   *         WRITER which field to test for a control break.       *
0192   *                                                               *
0193   *     CONTROL FOOTING (or CF) is a footing that appears         *
0194   *         immediately after the last detail line before a       *
0195   *         control break.                                        *
0196   *                                                               *
0197   *     DETAIL (or DE) is a detail line that is printed each      *
0198   *         time a GENERATE statement is executed in the          *
0199   *         PROCEDURE DIVISION.                                   *
0200   *                                                               *
0201   * ***************************************************************
0202
0203   * ***************************************************************
0204   *                                                               *
0205   *  The NEXT GROUP Clause (Lines 266 and 424)                    *
0206   *                                                               *
0207   *  The NEXT GROUP clause given the line-number of the line for *
0208   *  the beginning of the next group written.  The argument for  *
0209   *  NEXT GROUP can be a number; for example, NEXT GROUP IS 15    *
0210   *  places the next group on line 15 of the page.  The argument *
0211   *  can also be relative; for example, NEXT GROUP IS PLUS 2      *
0212   *  places the next line two lines below the current line.      *
0213   *                                                               *
0214   * ***************************************************************
```

```
0215      P R O G R A M   R E P E X M                COBOL-68 12B(1115) BIS
                                   15-JAN-81  15:19             PAGE 6
REP1.CBL    15-JAN-81  15:20

0216  * ***************************************************************
0217  *                                                               *
0218  *  The LINE Clause (Line 267 and throughout the RDs)            *
0219  *                                                               *
0220  *  The LINE NUMBER IS clause (which can be abbreviated to LINE)*
0221  *  tells on which line of the page a report entry should be     *
0222  *  written.  The LINE clause applies to the item containing it *
0223  *  and continues to apply until the end-of-record or until      *
0224  *  another LINE clause is found.                                *
0225  *                                                               *
0226  *  The LINE clause can take three kinds of arguments:          *
0227  *                                                               *
0228  *     1.  An integer that specifies the line number.            *
0229  *         For example, LINE NUMBER IS 25 specifies line 25.     *
0230  *         If the number is smaller than the current line, a     *
0231  *         new page is begun.                                    *
0232  *                                                               *
0233  *     2.  PLUS with an integer that specifies how many lines    *
0234  *         below the current line to print the current entry.    *
0235  *         For example, LINE PLUS 3 means to skip two lines      *
0236  *         before printing the current entry.                    *
0237  *                                                               *
0238  *     3.  NEXT PAGE, which specifies the next page.  If the     *
0239  *         record is a page header, it is printed on             *
0240  *         line 1; otherwise it is printed on line 2.            *
0241  *                                                               *
0242  * ***************************************************************
0243
0244  * ***************************************************************
0245  *                                                               *
0246  *  The COLUMN Clause (Line 267 and throughout the RDs)          *
0247  *                                                               *
0248  *  The COLUMN NUMBER IS clause (we can omit NUMBER IS) tells    *
0249  *  REPORT WRITER which column is the first for a record or      *
0250  *  field.  If a record or field does not have a COLUMN entry,  *
0251  *  it is not printed.                                           *
0252  *                                                               *
0253  * ***************************************************************
```

```
0254      P R O G R A M   R E P E X M              COBOL-68 12B(1115) BIS
                             15-JAN-81  15:19                PAGE 7
REP1.CBL    15-JAN-81  15:20

0255   * ***************************************************************
0256   *                                                               *
0257   *  The SOURCE Clause (Line 269 and throughout the RDs)          *
0258   *                                                               *
0259   *  The SOURCE IS clause (we can omit IS) specifies the source   *
0260   *  for an item.  The source item must have been defined in the *
0261   *  FILE or WORKING-STORAGE SECTION.  Its value is moved into    *
0262   *  the report item before the item is written in the file.      *
0263   *                                                               *
0264   * ***************************************************************
0265
0266     01      TYPE PH NEXT GROUP PLUS 2.
0267             02 LINE 1 COLUMN 22  PIC X(25) USAGE DISPLAY-7
0268                     VALUE 'State Totals of Customers'.
0269             02 LINE 2 COLUMN 31  PIC X(8) SOURCE THIS-DATE.
0270             02 LINE 5 COLUMN 1 PIC X(5) USAGE DISPLAY-7
0271                     VALUE 'State'.
0272             02 LINE 5 COLUMN 10 PIC X(19) USAGE DISPLAY-7
0273                     VALUE 'Number of Customers'.
0274             02 LINE 5 COLUMN 44 PIC X(5) USAGE DISPLAY-7
0275                     VALUE 'Sales'.
0276
0277   * ***************************************************************
0278   *                                                               *
0279   *  The SUM Clause (Line 309 and throughout the RDs)             *
0280   *                                                               *
0281   *  The SUM clause in the second following line specifies that   *
0282   *  the data-item is summed.  The data-item summed can be        *
0283   *  either a SOURCE item from a TYPE DETAIL line (for example,   *
0284   *  SORT-SALES in this program), or a summation counter (for     *
0285   *  example, CITY-COUNT).                                        *
0286   *                                                               *
0287   *  When either the SOURCE item or the summation counter is      *
0288   *  used, the value of the item is added to a compiler-          *
0289   *  generated accumulator and this accumulator is moved to the   *
0290   *  report item before writing.  The summation counter need      *
0291   *  not be named unless it is referenced directly in the         *
0292   *  PROCEDURE DIVISION or in another REPORT SECTION statement.   *
0293   *                                                               *
0294   *  A SUM clause can appear only in a TYPE CONTROL FOOTING       *
0295   *  record.  The accumulator is zeroed after being moved to the *
0296   *  report item.                                                 *
0297   *                                                               *
0298   *  You can selectively sum portions of a data-item by using     *
0299   *  the UPON option with the SUM clause.  In that case, summing *
0300   *  occurs only when the item is referenced by a GENERATE        *
0301   *  statement.  The individual items to be summed must be        *
0302   *  SOURCE items within a data-name specified as a TYPE DETAIL   *
0303   *  report group.                                                *
0304   *                                                               *
0305   * ***************************************************************
```

```
0306      P R O G R A M   R E P E X M                   COBOL-68 12B(1115) BIS
                               15-JAN-81  15:19                PAGE 8
REP1.CBL     15-JAN-81  15:20

0307    01      TYPE CF SORT-STATE LINE PLUS 1.
0308            02 COLUMN 3          PIC XX SOURCE CURRENT-STATE.
0309            02 COLUMN 15         PIC ZZ,ZZ9 SUM ONE-COUNT.
0310            02 COLUMN 35         PIC ZZ,ZZZ,ZZZ,ZZ9 SUM SORT-SALES.
0311
0312    01      TYPE CF FINAL LINE PLUS 2.
0313            02 COLUMN 1          PIC X(5) USAGE DISPLAY-7
0314                    VALUE 'Total'.
0315            02 COLUMN 15         PIC ZZ,ZZ9 SUM ONE-COUNT.
0316            02 COLUMN 35         PIC $$,$$$,$$$,$$9 SUM SORT-SALES.
0317
0318  * *************************************************************
0319  *                                                             *
0320  *  Missing COLUMN Clause (Lines 330 - 331)                    *
0321  *                                                             *
0322  *  The following lines illustrate the fact that a report      *
0323  *  item is not written in the report (even if directly        *
0324  *  specified in a GENERATE statement) unless the item has a   *
0325  *  COLUMN NUMBER clause.                                      *
0326  *                                                             *
0327  * *************************************************************
0328
0329    01      TYPE DETAIL.
0330            02                   PIC S9(5) SOURCE ONE-COUNT.
0331            02                   PIC S9(10) SOURCE SORT-SALES.
0332
0333  * *************************************************************
0334  *                                                             *
0335  *  The PAGE LIMIT Clause (Line 351)                           *
0336  *                                                             *
0337  *  The PAGE LIMIT clause specifies the number of lines that   *
0338  *  can be written on one page of the report.  If a line is    *
0339  *  written that would exceed PAGE LIMIT, page footings are    *
0340  *  written, a new page is begun, and page headings are written.*
0341  *                                                             *
0342  *  The PAGE LIMIT clause can contain additional options to    *
0343  *  control placement of page headings and footings, and the   *
0344  *  placement of first and last TYPE DETAIL lines.             *
0345  *                                                             *
0346  * *************************************************************
0347
0348    RD      BY-CITY
0349            CODE BY-CITY-CODE
0350            CONTROLS ARE FINAL SORT-STATE,SORT-CITY;
0351            PAGE LIMIT IS 58 LINES
0352                 HEADING 1, FOOTING 58, FIRST DETAIL 6,
0353                        LAST DETAIL 55.
```

```
0354      P R O G R A M   R E P E X M                COBOL-68 12B(1115) BIS
                             15-JAN-81  15:19                 PAGE 9
REP1.CBL    15-JAN-81  15:20

0355    01      REPORT-HEADER TYPE REPORT HEADING LINE 25.
0356            02 COLUMN 27        PIC X(27) USAGE DISPLAY-7
0357                    VALUE 'Customers By City and State'.
0358            02 LINE 29 COLUMN 36    PIC X(8) SOURCE THIS-DATE.
0359
0360    01      REPORT-FOOTER TYPE REPORT FOOTING LINE PLUS 2.
0361            02 COLUMN 30        PIC X(19) USAGE DISPLAY-7
0362                    VALUE '** End of Report **'.
0363
0364  * ***************************************************************
0365  *                                                               *
0366  *  The PAGE-COUNTER (Line 384)                                  *
0367  *                                                               *
0368  *  The compiler generates a data-item called PAGE-COUNTER for   *
0369  *  each report descriptor (RD) item.  It is set to 1 by the     *
0370  *  INITIATE statement, and incremented by 1 for each new page.  *
0371  *                                                               *
0372  *  If you define more than one report in the same program, you  *
0373  *  must qualify a reference to PAGE-COUNTER by using the name   *
0374  *  of the report.                                               *
0375  *                                                               *
0376  * ***************************************************************
0377
0378    01      PAGE-HEADING TYPE PAGE HEADING.
0379            02 LINE 1 COLUMN 1    PIC X(33) USAGE DISPLAY-7
0380                    VALUE 'Customers By City and State'.
0381            02 LINE 1 COLUMN 62   PIC X(4) USAGE DISPLAY-7
0382                    VALUE 'Page'.
0383            02 LINE 1 COLUMN 66   PIC ZZZ9
0384                    SOURCE PAGE-COUNTER OF BY-CITY.
0385            02 LINE 2 COLUMN 1    PIC X(8) SOURCE THIS-DATE.
0386
0387    01      STATE-HEADING TYPE CONTROL HEADING SORT-STATE
0388                    LINE PLUS 2.
0389            02 COLUMN 1 PIC X(9) USAGE DISPLAY-7
0390                    VALUE 'Customer'.
0391            02 COLUMN 30 PIC X(5) USAGE DISPLAY-7
0392                    VALUE 'State'.
0393            02 COLUMN 36 PIC X(4) USAGE DISPLAY-7
0394                    VALUE 'City'.
0395            02 COLUMN 65 PIC X(5) USAGE DISPLAY-7
0396                    VALUE 'Sales'.
```

```
             P R O G R A M   R E P E X M                COBOL-68 12B(1115) BIS
                                  15-JAN-81  15:19             PAGE 10
REP1.CBL     15-JAN-81  15:20

        01      DETAIL-LINE-1 TYPE DETAIL LINE PLUS 2.
                02 COLUMN  1 PIC X(24) USAGE DISPLAY-7
                        SOURCE SORT-NAME.
                02 COLUMN 32 PIC X(2)  USAGE DISPLAY-7
                        SOURCE SORT-STATE.
                02 COLUMN 36 PIC X(20) USAGE DISPLAY-7
                        SOURCE SORT-CITY.
                02 COLUMN 56 PIC ZZ,ZZZ,ZZZ,ZZ9
                        SOURCE SORT-SALES.
                02           PIC ZZ,ZZ9 SOURCE ONE-COUNT.

        01      DETAIL-LINE-2 TYPE DETAIL LINE PLUS 1.
                02 COLUMN  1 PIC X(20) USAGE DISPLAY-7
                        SOURCE SORT-STREET.

        01      CITY-FOOTING TYPE CF SORT-CITY LINE PLUS 3.
                02 CITY-COUNT COLUMN 4 PIC ZZ,ZZ9 USAGE DISPLAY-7
                        SUM ONE-COUNT.
                02 COLUMN 11 PIC X(17) USAGE DISPLAY-7
                        VALUE 'customers in city'.
                02 COLUMN 36 PIC X(20) USAGE DISPLAY-7
                        SOURCE SORT-CITY.
                02 CITY-SALES COLUMN 56 PIC $$,$$$,$$$,$$9
                        SUM SORT-SALES.

        01      STATE-FOOTING TYPE CF SORT-STATE LINE PLUS 3
                        NEXT GROUP NEXT PAGE.
                02 STATE-COUNT COLUMN 4 PIC ZZ,ZZ9 USAGE DISPLAY-7
                        SUM CITY-COUNT.
                02 COLUMN 11 PIC X(18) USAGE DISPLAY-7
                        VALUE 'customers in state'.
                02 COLUMN 32 PIC X(2) SOURCE SORT-STATE.
                02 STATE-SALES COLUMN 56 PIC $$,$$$,$$$,$$9
                        SUM CITY-SALES.
```

```
0432      P R O G R A M   R E P E X M                COBOL-68 12B(1115) BIS
                            15-JAN-81  15:19                PAGE 11
REP1.CBL    15-JAN-81  15:20

      01      FINAL-FOOTING TYPE CF FINAL LINE PLUS 1.
              02 COLUMN 3 PIC X(5) USAGE DISPLAY-7
                      VALUE 'Total'.
              02 COLUMN 15 PIC X(5) USAGE DISPLAY-7
                      VALUE 'Total'.
              02 COLUMN 25 PIC X(5) USAGE DISPLAY-7
                      VALUE 'Total'.
              02 COLUMN 45 PIC X(5) USAGE DISPLAY-7
                      VALUE 'Total'.
              02 LINE PLUS 1 COLUMN 1 PIC X(9) USAGE DISPLAY-7
                      VALUE 'Customers'.
              02 COLUMN 15 PIC X(6) USAGE DISPLAY-7
                      VALUE 'States'.
              02 COLUMN 25 PIC X(6) USAGE DISPLAY-7
                      VALUE 'Cities'.
              02 COLUMN 45 PIC X(5) USAGE DISPLAY-7
                      VALUE 'Sales'.
              02 LINE PLUS 2 COLUMN 1 PIC ZZ,ZZ9 SUM STATE-COUNT.
              02 COLUMN 16 PIC ZZ9 SOURCE NR-OF-STATES.
              02 COLUMN 26 PIC ZZ9 SOURCE NR-OF-CITIES.
              02 COLUMN 36 PIC $$,$$$,$$$,$$9 SUM STATE-SALES.
```

```
0454      P R O G R A M   R E P E X M                 COBOL-68 12B(1115) BIS
                                15-JAN-81  15:19              PAGE 12
REP1.CBL    15-JAN-81  15:20

0455    PROCEDURE DIVISION.
0456
0457  * **************************************************************
0458  *                                                              *
0459  *  The USE BEFORE REPORTING Verb (Line 470)                    *
0460  *                                                              *
0461  *  You can include the USE BEFORE REPORTING verb in the        *
0462  *  DECLARATIVES SECTION of the PROCEDURE DIVISION.  A report   *
0463  *  record is specified in the USE statement to indicate when   *
0464  *  the USE procedure is to be performed.  It is performed      *
0465  *  immediately before the report record is written.            *
0466  *                                                              *
0467  * **************************************************************
0468
0469    DECLARATIVES.
0470    EOR SECTION. USE BEFORE REPORTING REPORT-FOOTER.
0471    EOR-A. DISPLAY 'END OF REPORTS'.
0472    END DECLARATIVES.
0473
0474    MAIN SECTION.
0475
0476    START-PROC1.
0477           SORT SORT-FILE ON ASCENDING KEY
0478                   SORT-STATE,SORT-CITY,SORT-NAME
0479                  INPUT PROCEDURE IS IN-PROCEDURE
0480                  OUTPUT PROCEDURE IS OUT-PROCEDURE.
0481           STOP RUN.
0482    IN-PROCEDURE SECTION.
0483
0484    START-PROC2.
0485           OPEN INPUT CUSTOMER-FILE.
0486
0487    LOOP.
0488           READ CUSTOMER-FILE AT END GO TO DONE-INPUT.
0489           COMPUTE SORT-SALES ROUNDED = CUSTMR-SALES.
0490           MOVE CUSTMR-NAME TO SORT-NAME.
0491           MOVE CUSTMR-STATE TO SORT-STATE.
0492           MOVE CUSTMR-STREET TO SORT-STREET.
0493           MOVE CUSTMR-CITY TO SORT-CITY.
0494           RELEASE SORT-RECORD.
0495           GO TO LOOP.
0496
0497    DONE-INPUT. CLOSE CUSTOMER-FILE.
```

```
0498      P R O G R A M   R E P E X M             COBOL-68 12B(1115) BIS
                                 15-JAN-81  15:19          PAGE 13
REP1.CBL     15-JAN-81  15:20

0499    OUT-PROCEDURE SECTION.
0500
0501   * **************************************************************
0502   *                                                              *
0503   *  OPEN the Report File (Line 511)                             *
0504   *                                                              *
0505   *  The report file must be OPENed before any records can be    *
0506   *  written in it.                                              *
0507   *                                                              *
0508   * **************************************************************
0509
0510    START-PROC3.
0511           OPEN OUTPUT PRINTER-FILE.
0512           MOVE TODAY TO UNEDITED-DATE.
0513           MOVE UE-DAY TO TD-DAY; MOVE UE-MONTH TO TD-MONTH;
0514                   MOVE UE-YEAR TO TD-YEAR
0515                   MOVE '-' TO TD-HYF-1,TD-HYF-2.
0516
0517   * **************************************************************
0518   *                                                              *
0519   *  INITIATE the Reports (Lines 531 - 532)                      *
0520   *                                                              *
0521   *  The INITIATE statement causes the counters and accumulators *
0522   *  to be initialized.  The summation counters are set to 0;    *
0523   *  the PAGE-COUNTER is set to 1.                               *
0524   *                                                              *
0525   *  Each report written must be named in an INITIATE statement. *
0526   *  The output file for the report must be OPENed before any    *
0527   *  INITIATE statement is executed.                             *
0528   *                                                              *
0529   * **************************************************************
0530
0531           INITIATE BY-CITY.
0532           INITIATE STATE-TOTALS-ONLY.
```

```
0533      P R O G R A M   R E P E X M              COBOL-68 12B(1115) BIS
                              15-JAN-81  15:19              PAGE 14
REP1.CBL    15-JAN-81  15:20

0534  * **************************************************************
0535  *                                                              *
0536  *  GENERATE Report Records (Lines 577 - 578)                   *
0537  *                                                              *
0538  *  The GENERATE statement causes testing of control fields and *
0539  *  writes any required control headings and footings.  If the  *
0540  *  argument to the GENERATE statement is a TYPE DETAIL record, *
0541  *  the record is written after any control breaks.  If the     *
0542  *  argument is a report descriptor (RD), the detail lines are  *
0543  *  set up but not printed, so that a summary report is written.*
0544  *                                                              *
0545  *  In this program, both types of reports are generated.  The  *
0546  *  GENERATE DETAIL-LINE statement causes a detail report to be *
0547  *  written; the GENERATE STATE-TOTALS-ONLY statement causes a  *
0548  *  summary report to be written.                               *
0549  *                                                              *
0550  *  A GENERATE statement performs the following operations:     *
0551  *                                                              *
0552  *    1.  Increments and tests the PAGE-COUNTER and produces    *
0553  *        any required page footings and headings.              *
0554  *                                                              *
0555  *    2.  Tests for any control breaks and produces any         *
0556  *        required control footings and headings.               *
0557  *                                                              *
0558  *    3.  Adds all specified identifiers to summation counters. *
0559  *                                                              *
0560  *    4.  Executes any routines defined by USE statements.      *
0561  *                                                              *
0562  *    5.  If the argument to the GENERATE statement is a TYPE-  *
0563  *        DETAIL record, writes the detail report group.        *
0564  *                                                              *
0565  *  During the first execution of a GENERATE statement, all     *
0566  *  required report headings, page headings, control headings,  *
0567  *  and detail report groups are written.                       *
0568  *                                                              *
0569  * **************************************************************
0570
0571    LOOP.
0572            RETURN SORT-FILE; AT END GO TO DONE-REPORTS.
0573            IF CURRENT-STATE NOT EQUAL SORT-STATE
0574                    ADD 1 TO NR-OF-STATES.
0575            IF CURRENT-CITY NOT EQUAL SORT-CITY
0576                    ADD 1 TO NR-OF-CITIES.
0577            GENERATE DETAIL-LINE-1.
0578            GENERATE DETAIL-LINE-2.
0579            GENERATE STATE-TOTALS-ONLY.
0580            MOVE SORT-STATE TO CURRENT-STATE.
0581            MOVE SORT-CITY TO CURRENT-CITY.
0582            GO TO LOOP.
```

```
0583      P R O G R A M   R E P E X M                COBOL-68 12B(1115) BIS
                                15-JAN-81  15:19                 PAGE 15
REP1.CBL    15-JAN-81  15:20

0584  * ***************************************************************
0585  *                                                               *
0586  *  TERMINATE the Reports (Line 609)                             *
0587  *                                                               *
0588  *  The TERMINATE statement completes the processing for a       *
0589  *  report.  When the TERMINATE statement is executed, breaks    *
0590  *  occur for all control fields and all control footings are   *
0591  *  written; all page footings and report footings are also     *
0592  *  written.  If a program writes more than one report in the   *
0593  *  same file, each report must be named in a TERMINATE          *
0594  *  statement.                                                   *
0595  *                                                               *
0596  * ***************************************************************
0597
0598  * ***************************************************************
0599  *                                                               *
0600  *  CLOSE the Report File (Line 610)                             *
0601  *                                                               *
0602  *  The CLOSE statement closes the report file.  All reports     *
0603  *  written in the file must be TERMINATEd before the CLOSE      *
0604  *  statement is executed.                                       *
0605  *                                                               *
0606  * ***************************************************************
0607
0608   DONE-REPORTS.
0609          TERMINATE BY-CITY,STATE-TOTALS-ONLY.
0610          CLOSE PRINTER-FILE.
0611

NO ERRORS DETECTED
```

The following pages show the reports produced by the program just listed.

```
                    Customers By City and State

                              2-12-81
```

```
Customers By City and State                                 Page   1
 2-12-81


Customer                    State City                         Sales

Other Side Of The Fence       CT  Darien                      61,500
1247 Main Street


        1 customers in city       Darien                     $61,500

The Lawn Man                  CT  Hartford                   874,259
174 Barrington Drive

Turf World                    CT  Hartford                    47,525
8745 Asylum Street


        2 customers in city       Hartford                  $921,784

Bushwhackers, Inc.            CT  Hew Haven                   83,247
2735 Skyline Drive


        1 customers in city       Hew Haven                  $83,247


        4 customers in state   CT                         $1,066,531
```

```
Customers By City and State                                   Page   2
 2-12-81


Customer                     State City                         Sales

Boston Garden Shop             MA  Boston                       15,389
419 Beacon Street

Sodbusters                     MA  Boston                       45,639
8941 Commonwealth Av


        2 customers in city        Boston                      $61,028

Joyce Kilmer Trees             MA  Concord                       7,649
453 Henry T Drive


        1 customers in city        Concord                      $7,649

Mass Grass, Inc.               MA  Worcester                     9,042
8789 Worcester Drive


        1 customers in city        Worcester                    $9,042


        4 customers in state   MA                              $77,719
```

```
Customers By City and State                                    Page   3
 2-12-81


Customer                     State City                            Sales

Plastic Yards, Inc.            RI  Providence                      7,747
4772 Providence Blvd


        1 customers in city        Providence                     $7,747

Astro Grass Company            RI  Woonsocket                      6,489
666 Armageddon


        1 customers in city        Woonsocket                     $6,489


        2 customers in state   RI                                $14,236
```

```
Customers By City and State                               Page    4
 2-12-81


  Total          Total     Total                Total
Customers        States    Cities               Sales

    10             3         8             $1,158,486
                                     ** End of Report **
```

```
                    State Totals of Customers
                             2-12-81

State     Number of Customers                 Sales


  CT               4                      1,066,531
  MA               4                         77,719
  RI               2                         14,236

Total             10                     $1,158,486

                                    ** End of Report **
```

CHAPTER 11

PROGRAM SEGMENTS, SUBPROGRAMS, AND OVERLAYS

Some tasks which are done by computers require them to execute extraordinarily large amounts of code in order to solve the problem at hand. In rare cases, it can be difficult to fit the program into your memory area. In other, more numerous, cases, it can be desirable to split up the task into subtasks in order to make debugging easier or so that the subtasks can be given to different people to code. There are also cases where you wish to make some type of calculation which is difficult or impossible in COBOL. For any of these reasons, you can find it convenient to organize your program into parts to make the programming task easier, to allow the program to run more efficiently, or both. A COBOL programming task can be organized into program segments, into subprograms, or into an overlay structure to allow the breaking-up of the task. These various methods for breaking the task into smaller pieces must be carefully studied in order to determine which method is most suitable for your particular use. The methods themselves differ widely in the techniques used and the results obtained.

11.1 PROGRAM SEGMENTS

You can divide the Procedure Division of a COBOL program into parts called program segments. By doing this, you cause the system to run your program with some segments in memory only when they are needed; when they are not needed, they are on disk storage. Thus, the amount of memory required for execution is reduced.

You can define program segments in a main program or in a subprogram, but only one segmented program is allowed in a single load.

11.1.1 Section-Names And Segment Numbers

A program segment is made up of one or more sections, each of which begins with a SECTION statement of the form

```
     section-name SECTION nn.
```

where nn is a two-digit segment number in the range 00 to 99. A section extends from its SECTION statement to the next SECTION statement, or to the end of the program, whichever is first. All sections having the same segment number are in the same segment.

A program segment is either resident or nonresident, and writable or nonwritable, depending on its segment number, and on the setting of the segment-limit. (The SEGMENT-LIMIT IS nn statement in the Environment Division defines the segment limit, which is the smaller of nn and 49; if nn is omitted or nn is 0, the segment-limit is 49.)

A segment with a segment number of 50 or greater is nonresident and nonwritable; it is brought into memory only when it is needed for execution. Further, such a segment loses any changes made by ALTER statements when it leaves memory. It is in its original state each time it enters memory.

A segment with a segment number in the range SEGMENT-LIMIT to 49 is nonresident, but writable; it retains changes made by ALTER statements.

A segment with a segment number less than the SEGMENT-LIMIT (or with no segment number) is a resident and writable segment; it is always in memory during execution.

Nonresident segments are suitable for routines that are executed infrequently, run for a long time once begun, and require large amounts of memory. For example, a program that has four main tasks that are executed sequentially is an ideal application for nonresident segmentation. Placing each task in a nonresident segment allows the program to run with only one of the segments in memory at a time.

On the other hand, a frequently used routine should be placed in a resident segment to avoid the overhead of bringing it into memory time after time.

11.1.2 Examples

In the following sample program, there are nine program SECTIONs forming six program segments. (Recall that sections having the same segment numbers are in the same segment.)

```
P R O G R A M   S E G M N T     COBOL-68 12B(1033) BIS
          15-JAN-81  09:22      PAGE 1

SEGMNT.CBL    15-JAN-81  09:22
0001      IDENTIFICATION DIVISION.
0002      PROGRAM-ID. SEGMNT.
0003
0004      ENVIRONMENT DIVISION.
0005      CONFIGURATION SECTION.
0006      OBJECT-COMPUTER. DECSYSTEM-20
0007      SEGMENT-LIMIT IS 25.
0008
0009      DATA DIVISION.
0010
0011      PROCEDURE DIVISION.
0012      SECT1 SECTION 20.
0013      CALL A.
0014      SECT2 SECTION 65.
0015      CALL A.
0016      SECT3 SECTION 22.
0017      CALL A.
0018      SECT4 SECTION 20.
0019      CALL A.
0020      SECT5 SECTION 60.
0021      CALL A.
0022      SECT6 SECTION 30.
0023      CALL A.
0024      SECT7 SECTION 35.
0025      CALL A.
0026      SECT8 SECTION 35.
0027      CALL A.
```

```
0028    SECT9 SECTION 60.
0029    CALL A.
0030    STOP RUN.

NO ERRORS DETECTED
```

In the example above, the segments are as follows:

1. Segment 20 contains the sections SECT1 and SECT4. The SEGMENT-LIMIT IS 25 statement causes this segment to be resident and writable.

2. Segment 22 contains section SECT3; it is resident and writable.

3. Segment 30 contains section SECT6. Since its segment number is above the SEGMENT-LIMIT but less than 50, it is nonresident and writable; changes made to the segment are preserved even if it leaves and returns to memory.

4. Segment 35 contains sections SECT7 and SECT8. It is nonresident and writable.

5. Segment 60 contains sections SECT5 and SECT9. Since its segment number is above 50, it is nonresident and nonwritable; changes made to the segment are lost when it leaves and returns to memory.

6. Segment 65 contains section SECT2. It is nonresident and nonwritable.

11.2 SUBPROGRAMS

A COBOL subprogram is written and compiled as a separate program, but is meant to be executed together with other programs. When several programs are loaded and executed together, the program in which execution begins is called the main program; the other programs are called subprograms.

A large programming task can become more manageable if the program is divided into subprograms. Each subprogram can perform a few relatively simple tasks and each can be written and tested separately by using "dummy" main programs.

Using subprograms also permits you to define an overlay structure at load time. (See Section 11.3 for a discussion of overlays.)

A subprogram can open files, perform I/O for them, and close them; but no COBOL subprogram can perform I/O for files in another program. Any COBOL subprogram that performs I/O must be linked to the main program. That is, there must be a link, consisting of CALL statements, or a series of CALL statements through a series of subprograms, from the main COBOL program to any COBOL subprogram that wishes to do I/O. The CALL statement does not have to be executed to provide a link - in fact, it can be in such a position that it is never executed. This requirement is met by any group of subprograms all of which are written in COBOL. If, however, you wish to call a non-COBOL subprogram, you must make sure that any COBOL routines which are called by the non-COBOL subprogram have a link to the main COBOL program if the COBOL routines wish to do any I/O.

The COBOL compiler recognizes a subprogram by its use of LINKAGE SECTION, ENTRY, GOBACK, or the presence of the USING clause in the Procedure Division header. If a program has none of these, the compiler treats it as a main program.

The compiler generates a start address for a main program, but not for a subprogram. This start address is the address of the beginning of the Procedure Division, that is, the address where the first executable instruction is generated. This start address tells LINK and, in turn, the system where to begin execution of the program.

You can force the compiler to generate a start address for a subprogram by using the /J switch. You can prevent the compiler from generating a start address for a main program by using the /I switch.

NOTE

A subprogram can be treated as a main program (that is, can contain a start address) only if no statements in the Procedure Division refer to data in the Linkage Section. This is because in a main program only Data Division statements can allocate memory locations. There is no space in memory for data in the Linkage Section.

11.2.1 Inter-Program Communication

Main programs and subprograms communicate by transfering execution control and by sharing data. The shared data can be in files, but it is often more useful for them to share data that is already in memory.

11.2.1.1 **The Calling Program** - In the calling program, a CALL statement transfers execution control to a subprogram and optionally makes a list of data-items available to the called subprogram. The CALL statement has the form:

```
CALL {program- or entry-name} [USING identifier-1 [,identifier-2]...].
```

The program- or entry-name specifies the point to which execution control is to be passed in a subprogram. If a program-name is given, it is the PROGRAM-ID name in the subprogram, and control is transferred to the beginning of the subprogram's Procedure Division. If an entry-name is given, it is the name given by an ENTRY statement in the subprogram, and control is transferred to that statement.

Each program-name and entry-name must be unique among all those loaded together.

The identifiers specified in the CALL statement give a list of data-items in the calling program. The memory locations associated with them are then available for use in the called subprogram. If you omit the USING clause, no memory locations in the calling program are available to the called subprogram.

Each identifier must be defined in the File Section, Working-Storage Section, or Linkage Section of the calling program. Each data-item must be word-aligned. (Items at the 01 and 77 levels and COMP items are already word-aligned; others can be aligned by using the SYNCHRONIZED LEFT clause.)

11.2.1.2 **The Called Subprogram** - A subprogram can begin execution at any of its entry points. The beginning of the Procedure Division is always an entry point. Its entry-name is the name given in the subprogram's PROGRAM-ID statement.

You can name data-items to be available to the called program with a USING clause in the PROCEDURE DIVISION statement. This statement has the form:

```
     PROCEDURE DIVISION [USING identifier-1 [,identifier-2]...].
```

You can define additional entry points using the ENTRY statement, which has the form:

```
     ENTRY entry-name [USING identifier-1 [,identifier-2]...].
```

The specified entry-name is defined for use by CALL statements in calling programs and must be unique among all entry-names and program-names loaded together.

The USING clause of the calling program's CALL statement can have defined data-items to be made available to the called subprogram. If so, the USING clause of the entry-point statement (PROCEDURE DIVISION or ENTRY) can give identifiers to be used as local names for the shared memory.

The identifiers in the called subprogram's USING clause are assigned data-items in the shared memory from left to right. The lengths of the data-items in the called subprogram need not match those in the calling program; but the total length of the data-items in the called program must not exceed that in the calling program.

The identifiers in the USING clause must be defined in the subprogram's Linkage Section and they must be level-01 or level-77 identifiers.

When a subprogram is called, execution proceeds as in any program. Control leaves the subprogram at the first executed GOBACK, EXIT PROGRAM, or STOP statement.

If the subprogram does any I/O there must be a link to the main program consisting of COBOL subprograms. You can not have a COBOL subprogram doing I/O that is called by a non-COBOL subprogram.

Execution of a GOBACK or EXIT PROGRAM statement in a subprogram returns control to the calling program. Execution of the calling program resumes at the statement immediately following the CALL statement that called the subprogram. Any changes to the data-items specified in USING clauses at the entry point are preserved on return to the calling program.

The forms of the GOBACK and EXIT PROGRAM statements are:

```
     GOBACK.

     EXIT PROGRAM.
```

Execution of a STOP statement halts execution of the entire loaded program. The STOP statement has the form:

```
     STOP {RUN or literal}.
```

The STOP RUN statement ends program execution; there is no return to the calling program. The STOP literal statement causes a pause in program execution and the literal is typed on your terminal. If you then type CONTINUE, execution continues at the statement following the STOP literal.

11.2.2 Loading A Subprogram Structure

There are two ways to load a subprogram structure:

1. For simple loads, you can use the COMPILE-class commands.

2. For more complex loads, you must use LINK directly.

In either case, the following special considerations for loading subprogram structures apply: every entry point (program-name or entry-name) referenced in a CALL statement anywhere in the loaded program must be satisfied by loading a program containing the program-name or entry-name. If some referenced entry points are missing, a fatal LINK error occurs at load time.

11.2.3 Object Libraries And Searches

An object library is a file having one or more object modules; when LINK searches an object library, a module is loaded from the file only if it satisfies an unresolved global reference. (COBOL global references are created by the CALL or ENTER statement in a program; additional global references to routines in the object-time system are created by the COBOL compiler.)

NOTE

Object libraries are very different from source libraries. The source library is built using the COBOL utility program LIBARY and is accessed by the COPY statement in a COBOL program. The object library is built using the system program MAKLIB and is accessed by LINK command strings or by COMPIL-class system commands.

The /SEARCH and /NOSEARCH switches turn on and off LINK's library search mode. When the library search mode is off (the initial default), LINK loads each input file you specify. When the library search mode is on, LINK searches each specified input file as a library.

If the /SEARCH switch is appended to a file specification, then the switch is automatically turned off after that file is searched. For example:

```
     MYCOBL/SEARCH, COB4
```

searches MYCOBL.REL, but loads all of COB4.REL.

If the /SEARCH switch is not appended to a file specification, then the switch remains on until end-of-line or until a /NOSEARCH switch is found, whichever is earlier. For example:

```
     COB0,/SEARCH MYLIB1,MYLIB2,/NOSEARCH COB1
```

loads COB0, searches MYLIB1 and MYLIB2, and loads COB1.

The system library LIBOL.REL is searched automatically when LINK loads programs compiled with COBOL. If the program has LINK overlays, this search occurs at the end of each overlay (sometimes referred to as nodes).

You can change this normal search procedure by using LINK switches. The /SYSLIB switch requires LINK to search specified system libraries no matter what kind of modules are loaded. The /NOSYSLIB switch forbids search of specified system libraries. Using these two switches, you can select the time for searching system libraries.

The /USERLIB switch specifies that for modules from a specified translator, a given user library must be searched before the corresponding system library. For example, using the switch MYCOBL/USERLIB:COBOL requires LINK to search MYCOBL.REL before searching LIBOL.REL. The /NOUSERLIB switch can suspend the effect of a /USERLIB switch.

Using combinations of these search-related switches gives you precise control of library searches. All LINK switches are described in detail in the LINK Reference Manual.

11.2.4 Examples

Section 11.3.6 contains program listings of seven programs. The first of these is called CBL0; it is a main program. The remaining six programs are subprograms. Each has a Linkage Section that defines data items named in USING clauses of PROCEDURE DIVISION or ENTRY statements. The program CBL2 has two entry points defined by ENTRY statements.

The following example shows how to load, save, and run these programs. The LOAD system command loads the programs; the SAVE command creates a file (CBL0.EXE) for the loaded program; the RUN CBL0 command executes the program. All text between the RUN and EXIT lines were written by the executed program. The example is shown with a TOPS-10 system prompt character (.), but the TOPS-20 system prompt (@) could be there instead. TOPS-20 responds the same way to the LOAD command.

```
.LOAD CBL0,CBL1,CBL2,CBL3,CBL4,CBL5,CBL6 (RET)
COBOL:  CBL0     [CBL0.CBL]
COBOL:  CBL1     [CBL1.CBL]
COBOL:  CBL2     [CBL2.CBL]
COBOL:  CBL3     [CBL3.CBL]
COBOL:  CBL4     [CBL4.CBL]
COBOL:  CBL5     [CBL5.CBL]
COBOL:  CBL6     [CBL6.CBL]
LINK:   Loading

EXIT

.SAVE (RET)
CBL0.EXE.1 saved

.RUN CBL0 (RET)
We're at level 0 in program CBL0
CBL0   calling CBL2A
        We're at level 1 in program CBL2   at CBL2A
        CBL2   calling CBL5
                We're at level 2 in program CBL5
                CBL5 doesn't call anything
        Returned to CBL2
        CBL2   calling CBL6
                We're at level 2 in program CBL6
                CBL6   calling CBL3
                        We're at level 3 in program CBL3
                        CBL3 doesn't call anything
                Returned to CBL6
        Returned to CBL2
Returned to CBL0
CBL0   calling CBL4
        We're at level 1 in program CBL4
        CBL4   calling CBL1
                We're at level 2 in program CBL1
                CBL1   calling CBL2B
                        We're at level 3 in program CBL2   at CBL2B
                        CBL2B doesn't call anything
                Returned to CBL1
        Returned to CBL4
Returned to CBL0
Execution ends in CBL0

EXIT
```

11.3 OVERLAYS

If your loaded program would be too large to execute in one piece, you can define an overlay structure for it. This permits the system to execute the program with only some parts in your memory space at one time. (See the chapter on overlays in the LINK Reference Manual.)

11.3.1 When To Use Overlays

You do not need an overlay structure unless your program is too large for your memory area. If the program can fit in your memory space, you should not define an overlay structure for it; the monitor's page-swapping facility is faster than overlay execution.

11.3.2 Overlayable COBOL Programs

A COBOL subprogram structure is overlayable if it observes the following rules:

1. If a subprogram contains I/O verbs other than ACCEPT and DISPLAY, it must be placed in the root link. (The other I/O verbs are CLOSE, DELETE, OPEN, READ, REWRITE, START, and WRITE.) Further, the subprogram that does I/O must have a chain of calls from the main program entirely within the root link; the chain of calls cannot contain calls to subprograms in other links. Suppose, finally, that control could pass from the main program to an overlaid subprogram and thence to a root-node subprogram. If the root-node subprogram wishes to do any I/O (again, other than ACCEPT and DISPLAY), there must be a call from the main program to the root-node subprogram to establish the direct link.

2. The subprogram structure must not contain RERUN statements.

3. The subprogram structure must not contain reentrant code (compiled with /R under TOPS-10 or compiled without switches under TOPS-20.) Thus, users of TOPS-20 must use the /U switch and users of TOPS-10 need not use the /U switch, as it is the default, to avoid reentrant code.

To insure proper execution of a COBOL overlay, observe the following rules:

1. After bringing the overlay into memory (by a LOAD command), run it using the RUN command (not the START command).

2. Be sure that enough free memory is in the root link for the program to execute. (See Section 11.3.4.)

A subprogram loaded into a nonroot link is not writable. Each time the link comes into memory, it is in its original state.

11.3.3 Defining Overlays

A program overlay has a tree structure. The tree is made up of links, each containing one or more program modules. These links are connected by paths. Using LINK switches, you define each link and each path.

At the top of the tree is the root link, which must contain the main program. First-level links are below the root link; each first-level link is connected to the root link by one path.

Second-level links are below the first-level links, and each is connected by a path to exactly one first-level link. A link at level n is connected by a path to exactly one link at level n-1.

Notice that a link can have more than one downward path (to successor links), but only one upward path (to ancestor links).

Figure 11-1 shows a diagram of an overlay structure with 5 links. The root link is TEST; the first-level links are LEFT and RIGHT; the second-level links are LEFT1 and LEFT2.

Example of an Overlay Structure

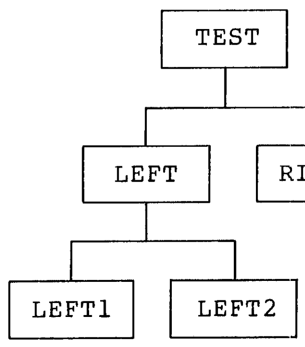


Figure 11-1 Example of an Overlay Structure

Defining an overlay structure allows your program to execute in a smaller space. This is because the code in a given link is allowed to make reference to memory only in links along a direct upward or downward path.

In the structure in Figure 11-1, the link LEFT can reference memory in itself, in the root link TEST, or in its successor links LEFT1 and LEFT2. More generally, a link can reference memory in any link that is vertically connected to it.

Referencing memory in any other link is illegal; for example, a path from LEFT1 to LEFT2 is not a direct upward or downward path.

Because of this restriction on memory references, only one complete vertical path (at most) is required in the memory area at any one time. The remaining links can be stored on disk while they are not needed.

LINK has a family of overlay-related switches for defining overlays. These switches are described in detail in the LINK Reference Manual. The following example shows command strings for defining the overlay diagrammed in Figure 11-1.

```
TEST/LOG/LOGLEVEL:2                                  ;Define TEST.LOG
/ERRORLEVEL:5                                        ;Important messages
TEST/OVERLAY                                         ;Define TEST.OVL
TEST/MAP                                             ;Define TEST.MAP
LPT:TEST/PLOT                                        ;Request diagram
CBL0,CBL1/LINK:TEST                                  ;Root link
     /NODE:TEST  CBL2,CBL3/LINK:LEFT                 ;Left branch
          /NODE:LEFT  CBL5/LINK:LEFT1                ;Left-left branch
          /NODE:LEFT  CBL6/LINK:LEFT2                ;Left-right branch
     /NODE:TEST  CBL4/LINK:RIGHT                     ;Right branch
TEST/SAVE                                            ;Define TEST.EXE
/E/GO                                                ;Execute now
```

The first command string above defines the .LOG file for the overlay. TEST/LOG specifies that the file is named TEST.LOG. The /LOGLEVEL:2 switch directs that only LINK messages at level 2 or greater be written in the .LOG file.

In the second command string, the /ERRORLEVEL:5 switch directs that messages below the level of 5 be suppressed for terminal typeout. The third command string, TEST/OVERLAY, tells LINK that an overlay structure is to be defined and that the file for the overlay is to be TEST.OVL.

The fourth command string, TEST/MAP, defines the file TEST.MAP for overlay symbol maps.

The next command string, LPT:TEST/PLOT directs that a diagram of the overlay links be printed on the line printer.

The next command string, CBL0,CBL1/LINK:TEST, loads the files CBL0.REL and CBL1.REL into the root link. The /LINK:TEST switch tells LINK that no more modules are to be in the root link and that the link name is TEST.

Each of the next four lines defines one link with a string of the form:

```
/NODE:linkname filenames/LINK:linkname
```

where:

/NODE:/linkname	specifies the previously defined link to which the present link is an immediate successor.
filenames/LINK:linkname	names the files in the current link and specifies the name of the link.

The first of these four lines begins with /NODE:TEST, which tells LINK that the link being defined is to be an immediate successor to TEST, the root link. Then (on the same line), the string CBL2,CBL3/LINK:LEFT loads the files CBL2.REL and CBL3.REL, ends the link, and names the link LEFT.

The next line, /NODE:LEFT CBL5/LINK:LEFT1, defines a link named LEFT1 containing the file CBL5.REL, and this link is an immediate successor to the link LEFT.

The next line, /NODE:LEFT CBL6/LINK:LEFT2, defines another immediate successor to LEFT, this time containing the file CBL6.REL and called LEFT2.

The last link is defined in the next line, /NODE:TEST CBL4/LINK:RIGHT. This string defines the link RIGHT, which is an immediate successor to TEST and contains the file CBL4.REL.

The next-to-last line in the example, TEST/SAVE, directs LINK to create the saved file TEST.EXE. The last line, /E/GO, specifies that the loaded program is to be executed and that all commands to LINK are completed.

11.3.4 The /SPACE Switch To LINK

For a COBOL overlay structure to execute properly, it must have free memory in its root link for the following uses:

1. General-purpose I/O buffers

2. I/O buffers and file tables for sorting

3. Label record area for multireel files

4. File index blocks for split index blocks of ISAM files

The /SPACE switch to LINK reserves free memory. It has the form:

```
/SPACE:n
```

where n is the decimal number of words to be reserved.

The /SPACE switch is used in the root link. For example, to allocate 5000 words of free memory in the overlay example above, you would type:

```
CBL0,CBL1/SPACE:5000/LINK:TEST
```

There are two types of space needed in the root link of a COBOL overlay: space for buffers and space for dynamic allocation.

Use the following guidelines to compute the free memory needed for buffers:

1. Two buffers are needed for each sequential file and one additional buffer is needed for each extra area used in the program.

   For an unblocked sequential file (on disk or magnetic tape), each buffer is 128 words. For example, the buffer space needed for one sequential file on disk with one alternate area is 3*128 = 384 words.

   For a blocked sequential file on magnetic tape, the buffer size is the blocksize (record-size*records/block). For example, the buffer space needed for one blocked sequential file with 100 records per block and records of 100 words each is 2*100*100 = 20000 words.

2. One buffer is needed for each random-access file and one for each file that is open for I/O. The buffer size is the number of 128-word blocks needed to hold the logical block, plus seven words.

   For example, a random-access file with logical blocks of 25 10-word records has a block size of 250 words. The smallest number of 128-word blocks containing 250 words is 2 (= 256 words). Therefore the buffer size is 256 + 7 = 263 words.

3. Indexed-sequential files require one buffer for each file. The buffer size is the sum of the following:

   a. Enough 128-word blocks to contain a logical block for each level of the index file.

   b. Enough 128-word blocks to contain a logical block of data.

   c. A number of 128-word blocks equal to the number used in an index block. These are used for storage allocation tables.

   d. One 128-word block for the statistics block.

e. One 128-word block for the index table.

f. A number of words equal to the largest index key-size, plus two words.

g. A number of words equal to the largest blocking factor of all the indexed-sequential files in the program. For example, if the largest blocking factor is 10, then 10 words are required in the buffer.

h. Enough 128-word blocks to contain the largest of the data or index blocks in all indexed-sequential files in the program.

For example, to compute the buffer size for an indexed sequential file with four levels, with 128-word index blocks and 256-word data blocks, compute as follows:

	512	Four 128-word index blocks
	256	One 256-word data block
	128	One 128-word storage allocation table block
	128	One 128-word statistics block
	128	One 128-word index table block
	256	Two 128-word blocks for the largest of all data or index blocks
	2	Two words for the largest blocking factor
	4	2-word index key plus two words
Total	1414	Buffer size (in words)

Use the following guidelines to compute the amount of free memory needed for dynamic allocation during program execution:

1. The size of the label-record area for a multireel file. This size is 16 words for standard labels. For nonstandard labels, the size is the number of characters in the label divided by 5.

2. The size of the index block of an indexed-sequential file if the top index block is split.

3. The size of the sort I/O buffers if sorting is used in the program. This size is calculated as the number of devices assigned to the sort file in the SELECT clause times two (for two buffers for each file) plus 26 words for each file table for each device.

For example, for a sort file with four assigned devices, calculate buffers as follows:

4 * 128 words *2 + (4 * 26 words) = 1128 words

NOTE

This calculation reflects only the requirements needed by COBOL. See also the SORT User's Guide for sort requirements.

If you do not allocate sufficient free memory with the /SPACE switch, either your program does not begin execution or it fails during execution.

11.3.5 The CANCEL Statement

You can use the CANCEL statement in a COBOL subprogram overlay structure to reduce memory size during program execution. This statement has the form:

```
     CANCEL subprogram-1 [,subprogram-2]....
```

where each named subprogram is in one of the overlay links.

The CANCEL statement creates a call to the REMOV. Overlay Handler subroutine. This directs removal from core of the links containing the named subroutines, along with all their successor links. The Overlay Handler attempts to return the recovered memory.

A CANCEL statement cannot direct removal of its own link or of any of its ancestor links, including the root link.

In the overlay structure diagrammed in Figure 11-1, for example, a subprogram loaded into the link LEFT can CANCEL subprograms in link LEFT1, LEFT2, or both. But it cannot CANCEL subprograms in its own link, LEFT, or in the root link, TEST.

11.3.6 Examples

The following pages show terminal listings of files associated with the example above. These pages show:

1. COBOL listing files for the programs used in the overlay (seven pages)
2. Terminal copy of the interactive use of LINK to define and execute the overlay (two pages)
3. The file TEST.MAP, generated by LINK, which shows symbol maps for the overlay (eight pages)

```
P R O G R A M   C B L 0                 COBOL-68 12B(1033) BIS       10-SEP-80  08:45  PAGE 1
CBL0.CBL    15-JAN-81  08:31

ID DIVISION.
PROGRAM-ID. CBL0.CBL
DATA DIVISION.
WORKING-STORAGE SECTION.
01      INFO.
        02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7 VALUE "We're at level ".
        02      LEVEL   PIC 9V    VALUE 0.
        02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7 VALUE " in program ".
        02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7 VALUE " calling ".
        02      RETMSG  PIC X(12) USAGE IS DISPLAY-7 VALUE "Returned to ".
        02      B       PIC X(8)  VALUE "       ".
01      PGMNAM  PIC X(6) VALUE "CBL0".
01      ENDMSG  PIC X(18) USAGE IS DISPLAY-7 VALUE "Execution ends in ".
PROCEDURE DIVISION.
        DISPLAY LEVMSG,LEVEL,PGMMSG,PGMNAM.
        DISPLAY PGMNAM,CALMSG,"CBL2A".
        CALL CBL2A USING INFO.
        DISPLAY RETMSG,PGMNAM.
        DISPLAY PGMNAM,CALMSG,"CBL4".
        CALL CBL4 USING INFO.
        DISPLAY RETMSG,PGMNAM.
        DISPLAY PGMNAM,CALMSG,"CBL2B".
        CALL CBL2B USING INFO.
        DISPLAY RETMSG,PGMNAM.
        DISPLAY ENDMSG,PGMNAM.
        STOP RUN.

NO ERRORS DETECTED
```

```
S U B   C B L 1              COBOL-68 12B(1033) BIS             10-SEP-80  08:46  PAGE 1
CBL1.CBL    15-JAN-81  08:30

ID DIVISION.
PROGRAM-ID. CBL1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01      PGMNAM  PIC X(6) VALUE "CBL1".
LINKAGE SECTION.
01      INFO.
        02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7.
        02      LEVEL   PIC 9V.
        02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7.
        02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7.
        02      RETMSG  PIC X(12) USAGE IS DISPLAY-7.
        02      B       PIC X(8).
PROCEDURE DIVISION USING INFO.
        ADD 1 TO LEVEL.
        DISPLAY B,B,LEVMSG,LEVEL,PGMMSG,PGMNAM.
        DISPLAY B,B,"CBL1 doesn't call anything".
        SUBTRACT 1 FROM LEVEL.
        GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 2                COBOL-68 12B(1033) BIS                 10-SEP-80  08:46  PAGE 1
CBL2.CBL    15-JAN-81  08:40

         ID DIVISION.
         PROGRAM-ID. CBL2.
         DATA DIVISION.
         WORKING-STORAGE SECTION.
         01      PGMNAM  PIC X(6) VALUE "CBL2".
         01      ENTNAM  PIC X(6).
         01      ENTMSG  PIC X(4) USAGE IS DISPLAY-7 VALUE " at ".
         LINKAGE SECTION.
         01      INFO.
                 02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7.
                 02      LEVEL   PIC 9V.
                 02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7.
                 02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7.
                 02      RETMSG  PIC X(12) USAGE IS DISPLAY-7.
                 02      B       PIC X(8).
         PROCEDURE DIVISION.
         ENTRY CBL2A USING INFO.
                 ADD 1 TO LEVEL.
                 MOVE "CBL2A" TO ENTNAM.
                 DISPLAY B,LEVMSG,LEVEL,PGMMSG,PGMNAM,ENTMSG,ENTNAM.
                 DISPLAY B,PGMNAM,CALMSG,"CBL5".
                 CALL CBL5 USING INFO.
                 DISPLAY B,RETMSG,PGMNAM.
                 DISPLAY B,PGMNAM,CALMSG,"CBL6".
                 CALL CBL6 USING INFO.
                 DISPLAY B,RETMSG,PGMNAM.
                 SUBTRACT 1 FROM LEVEL.
                 GOBACK.
         ENTRY CBL2B USING INFO.
                 ADD 1 TO LEVEL.
                 MOVE "CBL2B" TO ENTNAM.
                 DISPLAY B,LEVMSG,LEVEL,PGMMSG,PGMNAM,ENTMSG,ENTNAM.
                 DISPLAY B,"CBL2B doesn't call anything".
                 SUBTRACT 1 FROM LEVEL.
                 GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 3              COBOL-68 12B(1033) BIS              10-SEP-80  08:46  PAGE 1
CBL3.CBL    15-JAN-81  08:51

0001      ID DIVISION.
0002      PROGRAM-ID. CBL3.
0003      DATA DIVISION.
0004      WORKING-STORAGE SECTION.
0005      01     PGMNAM  PIC X(6) VALUE "CBL3".
0006      LINKAGE SECTION.
0007      01     INFO.
0008             02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7.
0009             02      LEVEL   PIC 9V.
0010             02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7.
0011             02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7.
0012             02      RETMSG  PIC X(12) USAGE IS DISPLAY-7.
0013             02      B       PIC X(8).
0014      PROCEDURE DIVISION USING INFO.
0015             ADD 1 TO LEVEL.
0016             DISPLAY B,B,B,LEVMSG,LEVEL,PGMMSG,PGMNAM.
0017             DISPLAY B,B,B,"CBL3 doesn't call anything".
0018             SUBTRACT 1 FROM LEVEL.
0019             GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 4                COBOL-68 12B(1033) BIS                  10-SEP-80  08:46  PAGE 1
CBL4.CBL    15-JAN-81  09:05

ID DIVISION.
PROGRAM-ID. CBL4.
DATA DIVISION.
WORKING-STORAGE SECTION.
01      PGMNAM  PIC X(6) VALUE "CBL4".
LINKAGE SECTION.
01      INFO.
        02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7.
        02      LEVEL   PIC 9V.
        02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7.
        02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7.
        02      RETMSG  PIC X(12) USAGE IS DISPLAY-7.
        02      B       PIC X(8).
PROCEDURE DIVISION USING INFO.
        ADD 1 TO LEVEL.
        DISPLAY B,LEVMSG,LEVEL,PGMMSG,PGMNAM.
        DISPLAY B,PGMNAM,CALMSG,"CBL1".
        CALL CBL1 USING INFO.
        DISPLAY B,RETMSG,PGMNAM.
        SUBTRACT 1 FROM LEVEL.
        GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 5              COBOL-68 12B(1033) BIS              10-SEP-80  08:48  PAGE 1
CBL5.CBL    15-JAN-81  09:05

         ID DIVISION.
         PROGRAM-ID. CBL5.
         DATA DIVISION.
         WORKING-STORAGE SECTION.
         01      PGMNAM  PIC X(6) VALUE "CBL5".
         LINKAGE SECTION.
         01      INFO.
                 02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7.
                 02      LEVEL   PIC 9V.
                 02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7.
                 02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7.
                 02      RETMSG  PIC X(12) USAGE IS DISPLAY-7.
                 02      B       PIC X(8).
         PROCEDURE DIVISION USING INFO.
                 ADD 1 TO LEVEL.
                 DISPLAY B,B,LEVMSG,LEVEL,PGMMSG,PGMNAM.
                 DISPLAY B,B,"CBL5 doesn't call anything".
                 SUBTRACT 1 FROM LEVEL.
                 GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 6                COBOL-68 12B(1033) BIS                10-SEP-80  08:48  PAGE 1
CBL6.CBL    15-JAN-81  09:04

         ID DIVISION.
         PROGRAM-ID. CBL6.
         DATA DIVISION.
         WORKING-STORAGE SECTION.
         01     PGMNAM  PIC X(6) VALUE "CBL6".
         LINKAGE SECTION.
         01     INFO.
                02      LEVMSG  PIC X(15) USAGE IS DISPLAY-7.
                02      LEVEL   PIC 9V.
                02      PGMMSG  PIC X(12) USAGE IS DISPLAY-7.
                02      CALMSG  PIC X(9)  USAGE IS DISPLAY-7.
                02      RETMSG  PIC X(12) USAGE IS DISPLAY-7.
                02      B       PIC X(8).
         PROCEDURE DIVISION USING INFO.
                ADD 1 TO LEVEL.
                DISPLAY B,B,LEVMSG,LEVEL,PGMMSG,PGMNAM.
                DISPLAY B,B,PGMNAM,CALMSG,"CBL3".
                CALL CBL3 USING INFO.
                DISPLAY B,B,RETMSG,PGMNAM.
                SUBTRACT 1 FROM LEVEL.
                GOBACK.

NO ERRORS DETECTED
```

```
.R LINK (RET)
*TEST/LOG/LOGLEVEL:5 (RET)              ;Define TEST.LOG
*/ERRORLEVEL:5 (RET)                    ;Important messages
*TEST/OVERLAY (RET)                     ;Define TEST.OVL
*TEST/MAP (RET)                         ;Define TEST.MAP
*CBL0,CBL1/LINK:TEST (RET)              ;Root link
[LNKLMN Loading module CBL0]
[LNKLMN Loading module CBL1]
[LNKLMN Loading module OVRLAY]
[LNKLMN Loading module CON012]
[LNKLMN Loading module LILOWS]
[LNKLMN Loading module TRACED]
[LNKLMN Loading module USRDSL]
[LNKLMN Loading module DBSTP$]
[LNKELN End of link number 0 name TEST]
*       /NODE:TEST   CBL2,CBL3/LINK:LEFT (RET)                        ;Left branch
[LNKLMN Loading module CBL2]
[LNKLMN Loading module CBL3]
[LNKELN End of link number 1 name LEFT]
*               /NODE:LEFT   CBL5/LINK:LEFT1 (RET)                    ;Left-left branch
[LNKLMN Loading module CBL5]
[LNKELN End of link number 2 name LEFT1]
*               /NODE:LEFT   CBL6/LINK:LEFT2 (RET)                    ;Left-right branch
[LNKLMN Loading module CBL6]
[LNKELN End of link number 3 name LEFT2]
*       /NODE:TEST   CBL4/LINK:RIGHT (RET)                             ;Right branch
[LNKLMN Loading module CBL4]
[LNKELN End of link number 4 name RIGHT]
*TEST/SAVE (RET)
*/E/GO (RET)
[LNKXCT CBL0 Execution]
```

```
We're at level 0 in program CBL0
CBL0   calling CBL2A
        We're at level 1 in program CBL2   at CBL2A
        CBL2   calling CBL5
                We're at level 2 in program CBL5
                CBL5 doesn't call anything
        Returned to CBL2
        CBL2   calling CBL6
                We're at level 2 in program CBL6
                CBL6   calling CBL3
                        We're at level 3 in program CBL3
                        CBL3 doesn't call anything
                Returned to CBL6
        Returned to CBL2
Returned to CBL0
CBL0   calling CBL4
        We're at level 1 in program CBL4
        CBL4   calling CBL1
                We're at level 2 in program CBL1
                CBL1 doesn't call anything
        Returned to CBL4
Returned to CBL0
CBL0   calling CBL2B
        We're at level 1 in program CBL2   at CBL2B
        CBL2B doesn't call anything
Returned to CBL0
Execution ends in CBL0

EXIT

.
```

```
          LINK symbol map of      TEST    version 12B(1102)              page 1
        Produced by LINK version 4B(1272) on 18-Mar-81 at  9:05:27

        Overlay no.     0       name    TEST
        Low  segment starts at       0 ends at    3144 length     3145 =   4P
        High segment starts at       0 ends at    3506 length     3507 =   4P
        Control Block address is   3105, length     30 (octal), 24. (decimal)
        411 words free in Low segment, 185 words free in high segment
        366 Global symbols loaded, therefore min. hash size is 407
        Start address is 400010, located in program CBLO

                *************

JOBDAT-INITIAL-SYMBOLS

        Zero length module

                *************

LIBOL-STATIC-AREA
        Low  segment starts at     140 ends at    1477 length     1340 (octal),    736. (decimal)

        .COMM.          140    Common  length     736.        .COMM.          140    Common  length     736.

                *************

CBLO    from DSK:CBLO.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:21:00
        Low  segment starts at    1500 ends at    1747 length      250 (octal),    168. (decimal)
        High segment starts at  400010 ends at  400214 length      205 (octal),    133. (decimal)

        CBLO           400022    Entry   Relocatable

                *************

CBL1    from DSK:CBL1.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:21:00
        Low  segment starts at    1750 ends at    2167 length      220 (octal),    144. (decimal)
        High segment starts at  400215 ends at  400440 length      224 (octal),    148. (decimal)

        CBL1           400217    Entry   Relocatable

                *************

OVRLAY  from SYS:OVRLAY.REL[4,20]       created by MACRO on 15-May-80 at 16:54:00
        Low  segment starts at    2170 ends at    2673 length      504 (octal),    324. (decimal)
        High segment starts at  400441 ends at  403506 length     3046 (octal),   1574. (decimal)

        BOUT%   104000000051    Global  Absolute              CLOSF%  104000000022    Global  Absolute
        ERJMP   320700000000    Global  Absolute              ERSTR%  104000000011    Global  Absolute
        GCVEC%  104000000300    Global  Absolute              GETOV.        402056    Entry   Relocatable
        GJ%OLD  100000000000    Global  Absolute   Suppressed GTJFN%  104000000020    Global  Absolute
        HALTF%  104000000170    Global  Absolute              INIOV.        402045    Entry   Relocatable
        JFNS%   104000000030    Global  Absolute              JS%DIR   70000000000    Global  Absolute    Suppressed
        JS%NAM    7000000000    Global  Absolute   Suppressed JS%PAF             1    Global  Absolute    Suppressed
        LOGOV.        402644    Entry   Relocatable           OF%BSZ  770000000000    Global  Absolute    Suppressed
        OF%RD         200000    Global  Absolute   Suppressed OF%WR         100000    Global  Absolute    Suppressed
        OPENF%  104000000021    Global  Absolute              PA%EX    20000000000    Global  Absolute    Suppressed
```

```
                 LINK symbol map of      TEST    version 12B(1102)              page 2
OVRLAY
         PA%PRV     200000000    Global  Absolute     Suppressed  PBOUT%  104000000074    Global  Absolute
         PSOUT%  104000000076    Global  Absolute                 REMOV.        402077    Entry   Relocatable
         RMAP%   104000000061    Global  Absolute                 RPACS%  104000000057    Global  Absolute
         RUNOV.        402115    Entry   Relocatable              RUNTM%  104000000015    Global  Absolute
         SFPTR%  104000000027    Global  Absolute                 SIN%    104000000052    Global  Absolute
         SOUT%   104000000053    Global  Absolute                 %OVRLA     402000052    Global  Absolute    Suppressed
         .FHJOB        777773    Global  Absolute     Suppressed  .FHSLF        400000    Global  Absolute    Suppressed
         .JSAOF             1    Global  Absolute     Suppressed  .NULIO        377777    Global  Absolute    Suppressed
         .OVRLA          2171    Entry   Relocatable              .OVRLO          2176    Global  Relocatable
         .OVRLU        402401    Entry   Relocatable              .OVRWA          2175    Global  Relocatable

                 *************

CON012   from SYS:LIBOL.REL[4,20]        created by MACRO on 30-Jan-81 at 17:52:00
         Low  segment starts at     2674 ends at    3073 length       200 (octal),    128. (decimal)

         CN.12           2674    Entry   Relocatable              COBST.          2674    Global  Relocatable
         GET%    104000000200    Global  Absolute                 GJ%PHY      10000000    Global  Absolute    Suppressed
         GJ%SHT       1000000    Global  Absolute     Suppressed  GT%ADR        200000    Global  Absolute    Suppressed
         JS%GEN      70000000    Global  Absolute     Suppressed  JS%TYP     700000000    Global  Absolute    Suppressed
         RF%LNG  400000000000    Global  Absolute     Suppressed  RFSTS%  104000000156    Global  Absolute
         .RFSFL             4    Global  Absolute     Suppressed

                 *************

LILOWS   from SYS:LIBOL.REL[4,20]        created by MACRO on 30-Jan-81 at 17:52:00

         Zero length module

                 *************

LIDISP   from SYS:LIBOL.REL[4,20]        created by MACRO on 30-Jan-81 at 17:52:00

         Zero length module

                 *************

TRACED   from SYS:LIBOL.REL[4,20]        created by MACRO on 30-Jan-81 at 17:52:00
         Low  segment starts at     3074 ends at    3103 length        10 (octal),      8. (decimal)

         BTRAC.          3077    Entry   Relocatable              C.TRCE          3074    Entry   Relocatable
         CBDDT.          3101    Entry   Relocatable              CNTRC.          3077    Entry   Relocatable
         HSRPT.          3077    Entry   Relocatable              PTFLG.          3102    Global  Relocatable
         SBPSG.          3077    Entry   Relocatable              SFOV.           3077    Entry   Relocatable
         TRPD.           3100    Entry   Relocatable              TRPOP.          3077    Entry   Relocatable

                 *************

USRDSL   from SYS:LIBOL.REL[4,20]        created by MACRO on 30-Jan-81 at 17:52:00

         Zero length module

                 *************
```

```
                LINK symbol map of      TEST    version 12B(1102)               page 3

        DBSTP$  from SYS:LIBOL.REL[4,20]        created by MACRO on 30-Jan-81 at 17:52:00
                Low  segment starts at    3104 ends at    3104 length        1 (octal),     1. (decimal)

                DBSTP$          3104    Entry   Relocatable

                        *************
```

```
Index to LINK symbol map of      TEST    version 12B(1102)              page 4

Name     Page     Name     Page     Name     Page     Name     Page

CBL0      1       DBSTP$    3       LILOWS    2       TRACED    2
CBL1      1       LIDISP    2       OVRLAY    1       USRDSL    2
CON012    2
```

```
        LINK symbol map of      TEST    version 12B(1102)       #1      page 5

Overlay no.     1       name    LEFT
Low  segment starts at    7145 ends at   10704 length     1540 =   2P
Control Block address is  10641, length     30 (octal), 24. (decimal)
Path is 0
59 words free in Low segment, 185 words free in high segment
24 Global symbols loaded, therefore min. hash size is 27

        *************

CBL2    from DSK:CBL2.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:23:00
        Low  segment starts at    7711 ends at   10160 length      250 (octal),   168. (decimal)
        High segment starts at    7145 ends at    7710 length      544 (octal),   356. (decimal)

        CBL2           7147    Entry   Relocatable             CBL2A           7165    Entry   Relocatable
        CBL2B          7450    Entry   Relocatable

        *************

CBL3    from DSK:CBL3.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:23:00
        Low  segment starts at   10421 ends at   10640 length      220 (octal),   144. (decimal)
        High segment starts at   10161 ends at   10420 length      240 (octal),   160. (decimal)

        CBL3          10163    Entry   Relocatable

        *************
        LINK symbol map of      TEST    version 12B(1102)       #2      page 6

Overlay no.     2       name    LEFT1
Low  segment starts at   10705 ends at   11376 length      472 =   1P
Control Block address is  11351, length     16 (octal), 14. (decimal)
Path is 0, 1
257 words free in Low segment, 185 words free in high segment
19 Global symbols loaded, therefore min. hash size is 22

        *************

CBL5    from DSK:CBL5.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:24:00
        Low  segment starts at   11131 ends at   11350 length      220 (octal),   144. (decimal)
        High segment starts at   10705 ends at   11130 length      224 (octal),   148. (decimal)

        CBL5            10707    Entry   Relocatable

        *************
```

```
                LINK symbol map of      TEST    version 12B(1102)       #3      page 7

        Overlay no.     3       name    LEFT2
        Low  segment starts at   10705 ends at   11446 length      542 =   1P
        Control Block address is  11421, length     16 (octal), 14. (decimal)
        Path is 0, 1
        217 words free in Low segment, 185 words free in high segment
        20 Global symbols loaded, therefore min. hash size is 23

                *************

CBL6    from DSK:CBL6.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:24:00
        Low  segment starts at   11177 ends at   11420 length      222 (octal),   146. (decimal)
        High segment starts at   10705 ends at   11176 length      272 (octal),   186. (decimal)

        CBL6            10707   Entry   Relocatable

                *************
                LINK symbol map of      TEST    version 12B(1102)       #4      page 8

        Overlay no.     4       name    RIGHT
        Low  segment starts at    7145 ends at    7664 length      520 =   1P
        Control Block address is   7637, length     16 (octal), 14. (decimal)
        Path is 0
        75 words free in Low segment, 185 words free in high segment
        20 Global symbols loaded, therefore min. hash size is 23

                *************

CBL4    from DSK:CBL4.REL[4,206]        created by COBOL-68 on 21-Jan-81 at 10:23:00
        Low  segment starts at    7415 ends at    7636 length      222 (octal),   146. (decimal)
        High segment starts at    7145 ends at    7414 length      250 (octal),   168. (decimal)

        CBL4            7147    Entry   Relocatable

                *************
```

```
Index to overlay numbers of TEST        version 12B(1102)              page 9

Overlay Page    Overlay Page    Overlay Page    Overlay Page

#0       4      #2       6      #3       7      #4       8
#1       5

Index to overlay names of TEST  version 12B(1102)

Name     Page   Name     Page   Name     Page   Name     Page

LEFT     5      LEFT2    7      RIGHT    8      TEST     4
LEFT1    6

[End of LINK map of      TEST]
```

CHAPTER 12

CALLING NON-COBOL SUBPROGRAMS

Some programming tasks are more conveniently done in a language other than COBOL. You can write non-COBOL subprograms for these tasks, and then call the subprograms from COBOL programs.

To call a non-COBOL subprogram, use the ENTER verb in the Procedure Division. The call has the form:

```
     ENTER language entry-name [USING string-1 [,string-2]...].
```

where:

language	is the name of the compiler that generated the subprogram.
entry-name	is the name of the entry point you want to call.
string	is an identifier, literal, or procedure-name. The list of strings is usually called an argument list or a parameter list.

The compilers that can generate COBOL-callable subprograms are COBOL, FORTRAN, and MACRO. The phrase ENTER COBOL is equivalent to CALL and is not discussed further here.

The entry point used in the ENTER statement must be an entry-name symbol generated by the compiler for the called program. COBOL generates an entry-name for each ENTRY statement and program-name. FORTRAN generates an entry-name for each SUBROUTINE, FUNCTION, and ENTRY statement. MACRO generates an entry-name for each ENTRY statement.

NOTE

You can use the "weaker" MACRO statement INTERN instead of ENTRY if you explicitly load the MACRO module. ENTRY is required only if the module must be loaded in a library search.

In the USING clause, using an identifier passes the value of the identifier to the called subprogram; using a literal passes the literal to the subprogram; using a procedure-name passes the address of the beginning of the named procedure, which can be used for alternate returns. FORTRAN cannot accept DISPLAY-6 (SIXBIT), DISPLAY-9 (EBCDIC), or COMP-3 (packed-decimal) data.

12.1 CALLING FORTRAN SUBPROGRAMS

When the COBOL compiler finds an ENTER FORTRAN statement, it generates a call for the named subprogram. If the ENTER statement contains a USING clause, the values indicated by the given identifiers, literals, and procedure-names are passed to the subprogram.

FORTRAN programs called by COBOL programs should not use blank COMMON, even among themselves. Doing so can overwrite storage in the COBOL program.

In the following example, the COBOL program CFSQRT calls the FORTRAN subprogram FSQRT to perform a square-root operation. The following list shows how values are passed from the main program to the subprogram:

Use of Value	COBOL Identifier	FORTRAN Variable
Input number	INPUT-NUMBER	INPUT
Answer	ANSWER	ANSWER
Error message location	ERROR-MESSAGE	ERRMSG
Exit message location	EXIT-MESSAGE	EXMSG

The following is the source file for the COBOL program CFSQRT:

```
ID DIVISION.
PROGRAM-ID. CFSQRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 INPUT-NUMBER USAGE COMP-1.
01 ANSWER USAGE COMP-1.
PROCEDURE DIVISION.
LOOP.
        DISPLAY 'Type a positive integer.'.
        ACCEPT INPUT-NUMBER.
        ENTER FORTRAN FSQRT USING INPUT-NUMBER,ANSWER,
                ERROR-MESSAGE,EXIT-MESSAGE.
        DISPLAY ANSWER.
        GO TO LOOP.
ERROR-MESSAGE.
        DISPLAY 'No negative numbers, please.'.
        GO TO LOOP.
EXIT-MESSAGE.
        DISPLAY 'Thank you.'.
        STOP RUN.
```

The following is the source file for the FORTRAN program FSQRT:

```
        SUBROUTINE FSQRT(INPUT,ANSWER,*,*)
        REAL INPUT
        INTEGER ERRMSG,EXMSG
        ERRMSG=1
        EXMSG=2
        IF(INPUT.LT.0) RETURN ERRMSG
        IF(INPUT.EQ.0) RETURN EXMSG
        ANSWER=SQRT(INPUT)
        RETURN
        END
```

In the following lines, these two source programs are executed. Each positive integer input yields its square root; a negative number yields an error message at an alternate return in the COBOL program; 0 yields the exit message at another alternate return. Note that the TOPS-10 system prompt could be replaced by the TOPS-20 prompt (@) without altering the example - the programs run exactly the same way under TOPS-20.

```
.EX CFSQRT.CBL,FSQRT.FOR
FORTRAN: FSQRT
FSQRT
COBOL:  CFSQRT  [CFSQRT.CBL]
LINK:   Loading
[LNKXCT CFSQRT Execution]
Type a positive integer.
4
2.0E0
Type a positive integer.
3
1.7320508E0
Type a positive integer.
2
1.4142136E0
Type a positive integer.
1
1.0E0
Type a positive integer.
-1
No negative numbers, please.
Type a positive integer.
0
Thank you.

EXIT
```

12.2 CALLING MACRO SUBPROGRAMS

When the COBOL compiler finds an ENTER MACRO statement, it generates the standard calling sequence:

```
MOVEI 16,arglist
PUSHJ 17,entry point
```

where arglist is the address of the first word of the argument list, and entry point is an entry-name symbol.

If the ENTER statement contains a USING clause, the compiler creates an argument list containing an entry for each identifier or literal in the clause. The word immediately preceding the argument list is of the form:

```
-length,,0
```

where length is the number of arguments in the list. If no USING clause appears in the ENTER statement, the length of the list is 0 (but the length word still appears).

Each entry in the argument list is a 36-bit storage word of the form:

```
|=========================================================|
|       0       | Code |       Effective Address (E)       |
|=========================================================|
0            8 9   12 13                               35
```

where code is a 4-bit code (described below), and bits 13-35 contain the effective address (E) of the first word of the argument.

If the passed argument is a 1-word COMP item, the code is 2 and E is the location of the argument.

If the passed argument is a 2-word COMP item, the code is 11 (octal) and E is the location of the first word of the argument; the second word of the argument is at E+1.

If the passed argument is a COMP-1 item, the code is 4 and E is the location of the argument.

If the passed argument is a DISPLAY-6, DISPLAY-7, DISPLAY-9, or COMP-3 item, or a figurative constant, the code is 15 (octal) and E is the location of a 2-word descriptor for the argument. The first word of the descriptor is a byte pointer word pointing to the argument. Its byte size is 6 for DISPLAY-6 or TODAY, 7 for DISPLAY-7 or literals or other figurative constants, and 9 for DISPLAY-9 and COMP-3 items.

The format of the second word is:

Bit 0	Reserved
Bit 1-4	Type code: 1= DISPLAY-6 2= DISPLAY-7 3= DISPLAY-9 4= COMP-3
Bit 5	Item is a literal
Bit 6	Item is a figurative constant (e.g. SPACES)
Bit 7	Item is numeric

If bit 7 = 0:

Bit 8-11	Reserved
Bit 12-35	Size of item in bytes

If bit 7 = 1:

Bit 8	Item is signed
Bit 9	Item is scaled (i.e. PICTURE contains "P's" to left of implied decimal point, e.g. 999PP)
Bit 10	Item is numeric edited

If bit 9 = 0:

Bit 11-25	Reserved
Bit 26-30	Number of decimal places

Bit 31-35	Size of item in bytes
If bit 9 = 1:	
Bit 11-25	Reserved
Bit 26-30	Scale factor (e.g. 2 if picture was 999PP)
Bit 31-35	Size of item in bytes (e.g. 3 if picture was 999PP)

The following are additional usage notes:

1. A figurative constant is passed exactly as a one-character, non-numeric DISPLAY-7 data item, except that bit 6 in the second descriptor word is set.

2. The figurative constant TODAY is passed as a 12-character DISPLAY-6 data item, except that bit 6 in the second descriptor word is set.

3. The figurative constant TALLY is passed as a 1-word COMP data item.

4. For an item with a picture that contains "P's" on the right side of the assumed decimal point (e.g. PIC VPP999), bit 9 is not set. This case can be recognized because the number of decimal places is greater than the total size of the item.

5. If the passed argument is a procedure-name (not allowed in a call to a COBOL subprogram), the code is 7 and E is the location of the first word of the procedure.

In the following example, the COBOL program CMSQRT calls the MACRO subprogram MSQRT to perform a square-root operation. (The subprogram uses the FORLIB routine SQRT to take the square root.)

The argument list generated by the ENTER MACRO statement is as follows:

```
                -4,,0           ;-Arglength,,0
        ARGLST: Z 4,address     ;<4B12>+<Address of 1st COMP-1 item>
                Z 4,address     ;<4B12>+<Address of 2nd COMP-1 item>
                Z 7,address     ;<7B12>+<Address of 1st procedure>
                Z 7,address     ;<7B12>+<Address of 2nd procedure>
```

The following is the source file for the COBOL program CMSQRT:

```
ID DIVISION.
PROGRAM-ID. CMSQRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 INPUT-NUMBER USAGE COMP-1.
01 ANSWER USAGE COMP-1.
PROCEDURE DIVISION.
LOOP.
        DISPLAY 'Type a positive integer.'.
        ACCEPT INPUT-NUMBER.
        ENTER MACRO MSQRT USING INPUT-NUMBER,ANSWER,
                ERROR-MESSAGE,EXIT-MESSAGE.
        DISPLAY ANSWER.
        GO TO LOOP.
ERROR-MESSAGE.
        DISPLAY 'No negative numbers, please.'.
        GO TO LOOP.
EXIT-MESSAGE.
        DISPLAY 'Thank you.'.
        STOP RUN.
```

The following is the source file for the MACRO program MSQRT. Notice that the entry-name MSQRT must be declared ENTRY and that the FORLIB routine SQRT, which is to be called, must be declared EXTERNAL.

Notice also that at NEG and ZERO, the return address in the stack is replaced by a procedure-name (address) to set up the alternate returns. At POS, the pointer to the argument list must be saved before calling SQRT.

```
        TITLE       MSQRT
        ENTRY       MSQRT
        EXTERN      SQRT
MSQRT:  SKIPN       1,@0(16)         ;Skip if not zero
        JRST        ZERO             ;To zero routine
        JUMPL       1,NEG            ;To negative routine
                                     ;Fall into positive routine
POS:    MOVEM       1,ARG            ;Save arg in reg 1
        MOVEM       16,SAVPTR        ;Save return address
        MOVEI       16,1+[-1,,0      ;Set up arg for SQRT
                    Z 4,ARG]
        PUSHJ       17,SQRT          ;FORLIB square root routine
        MOVE        16,SAVPTR        ;Restore return address
        MOVEM       0,@1(16)         ;Set up return arg
        POPJ        17,              ;Return
ZERO:   MOVEI       1,@3(16)         ;Set up alternate return
        MOVEM       1,0(17)          ;  for zero arg
        POPJ        17,              ;Return
NEG:    MOVEI       1,@2(16)         ;Set up alternate return
        MOVEM       1,0(17)          ;  for negative arg
        POPJ        17,              ;Return
ARG:    BLOCK 1
SAVPTR: BLOCK 1
        END
```

In the following lines, these two source programs are executed. Since neither program is a FORTRAN program, FORLIB must be explicitly searched.

Each positive integer input yields its square root; a negative number yields an error message at an alternate return in the COBOL program; 0 yields the exit message at another alternate return. Note that the execution of these programs yields the same output if run under TOPS-10.

```
@EXE CMSQRT.CBL,MSQRT.MAC,SYS:FORLIB.REL/SEARCH
COBOL:  CMSQRT  [CMSQRT.CBL]
MACRO:  MSQRT
LINK:   Loading
[LNKXCT CMSQRT Execution]
Type a positive integer.
4
2.0E0
Type a positive integer.
3
1.7320508E0
Type a positive integer.
2
1.4142136E0
Type a positive integer.
1
1.0E0
Type a positive integer.
-1
No negative numbers, please.
Type a positive integer.
0
Thank you.

EXIT

@
```

CHAPTER 13

IMPROVING PERFORMANCE OF COBOL-68 PROGRAMS

Normally, the code generated by the COBOL-68 compiler is adequately efficient. However, since there are certain COBOL-68 constructions for which efficient code is not generated, it is possible to write programs that perform poorly. If your programmed application performs inefficiently, you are left with the following alternatives:

1. Assume that a higher-performance version of the COBOL-68 compiler solves the problem

2. Purchase new or faster hardware

3. Redesign the entire program

4. Rewrite only the bad portions of the program

Assuming that you are unwilling to wait for an improved compiler or purchase new or faster hardware, let us consider the remaining alternatives.

Although redesigning the entire program or application is possible, it is expensive and is generally not done. Like any system rewrite, however, it does offer the opportunity to add new features and eliminate old, out-of-date ones. It is a good alternative, in the long run.

The much cheaper solution is to determine why a program is performing poorly and rewrite only the inefficient portions. This normally does not require a large effort since most COBOL programs spend 90% of the time executing only 10% of their code. The biggest task involves determining why a program is inefficient.

Most programs lend themselves to some improvement. There have been many instances where a program used less than half the CPU time after improvement than it did before. Most often, the gain is in the range of 30%. Most significant is the fact that the reprogramming generally involved only 20 lines or less.

Because some optimization techniques can be contrary to programming standards, it is necessary to use discretion when choosing which programs to improve and how much to improve them. It is, therefore, not recommended that all programs be optimized. For example, little is gained if a weekly application has its CPU time cut from 10 to 5 minutes. A program that runs for 2 hours a day, on the other hand, probably should be investigated.

Program optimization is usually done on an as-needed basis: the greater the resource consumption by a program, the greater the priority for optimization. Therefore, your installation's programming standards should guide programmers towards efficient, partially optimized programs.

Each computer system is different. Therefore, it is likely that installation programming standards reflects, to some extent, practices that promote efficient use of the presently installed system. On some systems, for example, the size of a program, the number of files open, and the type of devices used affects a program's performance. On other systems, emphasis is placed on data types, coding practices, and data patterns. It is normal for a programming standard to reflect those practices that normally produce efficient results without impairing reliability or maintainability.

The standard, therefore, could stipulate that all counters, indexes, and subscripts be described as COMPUTATIONAL. It could also, as is the case with most TOPS-10 and TOPS-20 installations, standardize around DISPLAY-6 files because of file-space economy. Another standard practice is to request that an analysis of the data be made and that the program be written to efficiently process it. For example, the following program statements make some decisions based on the value of a particular item:

```
     IF ABLE > BAKER GO TO CHARLIE.
     IF ABLE < BAKER GO TO DOG.
     IF ABLE = BAKER GO TO ECHO.
```

If the value of ABLE is normally equal to BAKER, the program should be reordered with the following statement first:

```
     IF ABLE = BAKER GO TO ECHO.
```

Programming techniques of this type promotes efficiency on virtually every system and should be encouraged.

Any programmer who can write COBOL programs can optimize them. Most of the programming tools currently available require minimal knowledge of anything other than COBOL. The optimization tools and techniques described in this chapter plus the techniques described in your installation standard provide most of the information needed to improve most COBOL programs.

It is easy to apply already known optimizations to a program. It becomes more difficult to make programs more efficient, however, when the known optimization techniques are not applicable. The person who can be most successful is one who understands a little about the code generated by the compiler and can read assembler code. By using the /A switch option to obtain a listing of the assembly language code generated for the program, you can determine, from the code generated, which alternatives produce the best results.

There are many ways to make a program more efficient. The best results come from good program design. Minimizing disk access, segmenting programs into small well-defined pieces, and keeping irrelevant information out of records are some ways to gain more efficiency. Discussion of these techniques, because they are applications-specific, are beyond the scope of this chapter. They are mentioned here in order that you take them into consideration when designing your individual applications. The remainder of this chapter deals with program improvements. It is a collection of techniques that have been used to good advantage by various installations.

13.1 HOW TO PROCEED WITH PROGRAM OPTIMIZATION

The actual coding required to optimize a program is usually minimal and not time-consuming. The largest component of time is spent learning the nature of the problem, that is, determining where and how much time is being spent by the program. Therefore, once a program has been selected for investigation, it is advisable to form a plan or procedure to be followed. This plan should consist of a series of small steps each designed to improve a small portion of the program. As one portion of the program is improved, begin on the next, and so on until the entire program has been improved to your satisfaction.

NOTE

Do not attempt program optimization until the program has been debugged and runs correctly!

13.1.1 Where To Begin

Begin by gathering together the following material and information:

1. An understanding of the goal (lower elapsed time or CPU time)
2. Copies of the source program and supporting software
3. Enough data to make this program run long enough to measure, and short enough to allow quick turn-around - 10 to 15 minutes is usually sufficient
4. Files for output verification
5. Access to the measurement tools (see Section 13.1.2)
6. A notebook to record all observations, measurements, and results (see Section 13.1.5)

13.1.2 What Tools Are Available

There are some tools that are part of the system software; you can have others at your installation; and some are available through DECUS and other agencies. This chapter discusses those that are part of the system software and are commonly used and understood. These tools are:

- COBDDT - For users of TOPS-10 and TOPS-20 - see Sections 7.3 and 13.2
- SET WATCH - For users of TOPS-10 only - see the TOPS-10 Operating System Commands Manual

13.1.3 What Method Or Procedure To Use

Once you have gathered all of the information and materials required, and are familiar with the various tools at your disposal, it is time to decide upon a course of action. The following procedure is provided as a guide. You can expand or shorten it as benefits your application or installation.

1. Generate a version of the program and its data that uses 10 to 15 minutes elapsed time. Remove anything from the program (terminal interaction, logical names, etc.) that make it difficult to run.

2. Schedule your machine time to coincide with periods when the system is lightly loaded. This enables you to make better use of the elapsed time statistics.

3. Run the unaltered (original) program and determine the following statistics:

   a. Amount of CPU time used

   b. Elapsed time

   c. Amount of idle time on the system

   d. Amount of disk I/O, swapping, etc.

   e. Use SET WATCH to observe the program during its execution; SET WATCH aids you in determining CPU time, peripheral usage, etc.

   Some of these statistics are not too meaningful on a system with even a moderate work load. Only the person conducting the test can determine to what extent the system work load can bias the measurement. However, even if the system is loaded, CPU time is normally a good indication of how the program performs. If the program runs with idle time, determine the reason for it (disk wait, tape wait, etc.). Often, additional buffering can lower the elapsed time. (See Section 13.1.4, Evaluating Performance.)

4. Run a COBDDT histogram to determine its runtime statistics. The histogram aids you in spotting potential problem areas in the program.

5. If other tools are available, use them.

6. Save the output from this first run for verification.

7. Analyze the results and make any changes you believe improves the program.

8. Recompile, link, and execute the program using the tools and techniques mentioned above.

9. Compare the statistics from this run with those of the previous or original run.

10. Write down all observations, facts, and hunches. (See Section 13.1.5, Documentation.)

11. Repeat steps 7 through 10 until you are satisfied with the results.

The last step, repeat until satisfied, is very important. It is very easy to get carried away with program optimization. Start with a premise, for example, "I will be satisfied with a 30% improvement". When you reach this level of performance, stop.

13.1.4 Evaluating Performance

Generally the best criterion for evaluating performance is the one that led you to be suspicious of the program in the first place. Most generally, CPU time is used. It is easy to measure and easy to reproduce. You simply observe the CPU time in the original program, make changes as appropriate, rerun the program and observe it again. If the CPU time decreases, the changes were effective.

NOTE

Because CPU time can vary with the load on the system, only changes in excess of 5% can be considered significant.

Another, more effective, way to determine performance is to measure the amount of work done per second of CPU time. By counting the number of records processed per second or minute, you have a good way to document a program's performance. Thus, if a program can normally process 100 records per CPU minute, and the volume increases by 1000 records per run, the effect is clearly an increase of about 10 CPU minutes per run.

13.1.5 Documentation

It is a good practice to document everything you have done during program optimization. You may want to improve other programs, and the notes you take for the first attempt can aid you in saving time and effort on each succeeding attempt. The documentation kept should be simple and should include the following information:

1. The name of the program, the time and date of the run, and the name of the programmer
2. The amount of data used by the test program, for example, 1000 records for a 10-minute run
3. The time (CPU and elapsed) used by the original program
4. The level of performance desired
5. The optimization techniques utilized
6. The results obtained, both positive and negative
7. COBDDT histogram
8. Any observations about system performance

9. Any other statistics collected, feelings, hunches, and other perceptions

The documentation need not be elegant. It should, however, be permanent. You might even tape portions of the console log into your notebook as a quick way of recording timings.

13.2 LISTING THE TOOLS

This section discusses the tools most commonly used by COBOL programmers for program optimization: COBDDT and SET WATCH. You are advised to read Section 7.3, COBDDT, before reading this section. The write-up on SET WATCH in the TOPS-10 Operating System Commands Manual is also recommended for users of TOPS-10. This section does not attempt to redo anything that has already been done. It attempts only to present information relevant to program optimization.

13.2.1 COBDDT

This section discusses COBDDT as used for evaluating program performance. Therefore, only the histogram feature is described. The COBDDT histogram provides you with the following information for each procedure that was executed in your program (see Figure 13-1, Sample COBDDT Histogram):

- Procedure name
- The number of times the procedure was entered (ENTRIES)
- The CPU time the procedure used (CPU)
- The elapsed time the procedure used (ELAPSED)

```
COBDDT HISTOGRAM FOR XDDT04                           REPORT:  1
XDDT4B.HIS

PROCEDURE                               ENTRIES        CPU        ELAPSED

1ST                                           1      0.336          1.649
 P12                                          5      0.251          1.239
 PP3                                          1      0.028          0.333
 PP4                                          1      0.005          0.005
 PP5                                          2      0.045          0.065
2ND                                           3      0.123          0.398
 2P0                                          3      0.013          0.029
 2P1                                          7      0.032          0.065
 2P3                                          7      0.030          0.152
 2P10                                         7      0.030          0.047
3RD                                          10      0.115          0.380
 3P0                                         10      0.050          0.108

XDDT4B.HIS


OVERHEAD:                               ELAPSED:    0.002   CPU:     0.002
```

Figure 13-1 Sample COBDDT Histogram

13.2.1.1 **The ENTRIES Column** - The information listed in the ENTRIES column of the histogram helps you to set your priorities for program improvement. Very high counts relative to others establishes the paragraph as one which needs further investigation. For example:

1. Why is it entered so often?

2. Is anything done there that could be done more effectively elsewhere?

3. Can it be rewritten to do less? (See Section 13.5, Efficient Coding Conventions.)

Often, the numbers guide you into understanding how to order your decision lists. For example:

> Suppose P-1 was typically entered 1000 times, P-2 was entered 500 times, and these paragraphs are chosen by a decision list that looks like this:

```
S-1.  IF A = "  " GO TO P-2.

S-2.  IF A = "00" GO TO P-1.
```

> It is apparent, then, that the order of S-1 and S-2 should be reversed because A is usually 00.

Also, based on the number of records processed, unexpected counts in certain paragraphs should be accounted for.

Do not be afraid to add new paragraph names to the program. Not only does this technique allow you to break large paragraphs up into smaller ones, it also enables you to better understand exactly where the program spends its time.

13.2.1.2 **The CPU Column** - The histogram's CPU column lists the amount of CPU time each paragraph used. Generally, if you can cut the CPU time, the elapsed time also drops and the application performs more efficiently. By analyzing this column, you can easily identify the big spenders - those procedures that eat up most of the CPU time. One approach is to rank the paragraphs in terms of CPU time and to look for paragraphs that spend more time per entry than others. Then, proceeding in rank order, determine what each paragraph is doing, if it has to do it, and if a better coding technique is in order. Usually only a few paragraphs need be examined.

NOTES

1. CPU time for a paragraph also includes time spent in paragraphs performed or routines called. Therefore, the sum of the CPU time is greater than the total time actually spent within this paragraph. (See Section 13.2.1.4.)

2. CPU time also includes time spent in the object-time system, the monitor, and for users of TOPS-20, the compatability package.

If after examining the list of the most time-consuming paragraphs, you determine that all can be explained, it is unlikely that changing any particular thing improves performance. Either the program cannot be improved any further, or other techniques are needed.

13.2.1.3 **ELAPSED Column** - In a lightly loaded system, the elapsed time can be a guide to the effective blocking of records. Some experiences with programs that seemed I/O-bound indicated that they were spending a great deal of time in the paragraphs that dealt with random or ISAM reads and updates. Inspection of the blocking revealed that while the files were blocked to conserve disk space, large amounts of data were being transferred (1 block) when the desired object was to update 1 record. Therefore, if a disproportionate amount of time is spent in some paragraphs, there could be a problem in processing. These paragraphs should definitely be investigated.

13.2.1.4 **OVERHEAD** - This entry in the histogram, (see Figure 13-1) represents the time spent for PERFORM or CALL overhead. Look at this entry to evaluate the cost of PERFORM-loop control mechanisms. If this figure is high, then some very short paragraph is being performed a large number of times. If this is the case, a more efficient method of loop control is probably in order.

13.3 USING THE CORRECT DATA TYPE

Understanding the various data types available is extremely important because there are so many of them. COBOL-68 offers you three different DISPLAY types and several COMPUTATIONALS. Each data type offers some advantages and some disadvantages. It is necessary to understand these in order to maximize the efficiency of a particular application.

13.3.1 DISPLAY Data Types

There are 3 display data types used within COBOL.

EBCDIC
ASCII
SIXBIT

EBCDIC and ASCII are character codes that occupy 8 and 7 bits per character respectively. The representations for each character are defined by industry standards. SIXBIT is a 6-bit BCD code which is defined by DIGITAL.

13.3.2 EBCDIC

The 8-bit EBCDIC code allows 256 different characters. It is compatible with IBM and thus is a natural where data interchange with 360s and 370s is necessary. EBCDIC files can contain a mixture of EBCDIC and COMPUTATIONAL-3 data. EBCDIC is packed into the computer's memory, 4 characters per word.

EBCDIC processing is going to be somewhat slower than either ASCII or SIXBIT because of the amount of space that each character takes up. As an example, a 120-character record would occupy:

1. 30 words in EBCDIC
2. 24 words in ASCII
3. 20 words in SIXBIT

Since movement of data is roughly linear with volume (it takes twice as long to move twice as much), it can be seen that SIXBIT and ASCII are 33% and 20% more efficient than EBCDIC, respectively.

The amount of file storage is also proportional to the byte size. For example, five ASCII records or 6 SIXBIT records can be stored in the same space taken by only 4 EBCDIC records.

Thus the use of EBCDIC should be restricted to those cases where:

1. The ASCII and SIXBIT character set is too small (128 and 64 characters respectively compared with 256 for EBCDIC).
2. The transmittal of data to and from EBCDIC systems is a major part of the application.
3. The application depends on the collating sequence (numerics after alphabetics).
4. The existence of many redefined records with mixtures of EBCDIC and COMP-3 makė reprogramming unthinkable.

In summary, it suffices to say that EBCDIC is a useful data type available to the COBOL user. For whatever its benefit, you must realize that it is slower and that a 33% increase on throughput could be realized by going to SIXBIT.

13.3.3 ASCII

Seven-bit ASCII is the coding sequence utilized by the unit record peripherals and terminals. Any other data type (EBCDIC or SIXBIT) has to be converted to ASCII if it is to be sent to one of these devices.

In memory, the use of ASCII makes the movement of data proceed faster than EBCDIC but slower than SIXBIT because of the number of characters per word. On the disk, all ASCII records are variable length as defined by industry standards, the end of an ASCII record is defined by the existence of a "vertical form" (normally a line feed) character (or several such characters). Thus, it is necessary to read ASCII files a character at a time in order to find the end-of-record character. This implies that ASCII records can be variable length and efficiently stored on the disk. It also implies that moving such records to or from memory is more costly than the other data types that can be moved by the block transfer instruction.

ASCII is the standard data type for "text files". Files created by editors that contain arbitrary length records can be stored economically and processed easily using the ASCII data type. Cards from a reader can be "trailing blank suppressed" so that they can be stored economically and are easily manipulated using ASCII. However, unless the full character set capabilities of ASCII (128 with lowercase plus line control) are necessary or the data is coming from or going to an ASCII peripheral, conversion to SIXBIT files is probably preferable.

13.3.4 SIXBIT

By far the most efficient DISPLAY code is SIXBIT. Six characters can be packed per word. Each record on disk or tape is preceded by a word with a character count allowing for block transfers of data. And the transmission time for moving the data around memory is less than any other data type.

The only problem with SIXBIT is the number of characters possible within the 6-bit code. The 64 characters allow for uppercase, numerics, and punctuation. It does not allow for lowercase, device control characters, or special graphics.

Most installations put the bulk of their files into SIXBIT due to the storage economy and the processing efficiency. Use if SIXBIT files is highly recommended wherever possible.

13.3.5 COMPUTATIONAL

There are several flavors of computational data types available to the COBOL programmer including:

1. COMPUTATIONAL-3, the four-bit complement of EBCDIC
2. COMPUTATIONAL, internal binary (35 bits plus sign)
3. Double-word COMPUTATIONAL, automatically invoked when the number of digits desired is greater than 10 (70 bits plus sign)
4. COMPUTATIONAL-1, floating point (the hardware supports double-precision floating point, but COBOL does not)

Aside from the use of COMP-3 as an adjunct to EBCDIC, the most useful data type is COMPUTATIONAL. This is normally used for indexes, counters, and subscripts. If other data types are used for these purposes, there is continual conversion taking place since all arithmetic is done in binary.

You can read arbitrary files by defining them as BINARY mode and then use the data as desired.

13.4 DATA EFFICIENCIES

Programming standards should insist on using the correct data types for certain operations. Using COMPUTATIONAL for counters works better on almost any machine.

13.4.1 Counter, Indexes, Subscripts

In DIGITAL COBOL indexes and subscripts are not different (this is not the case with some systems). They are, in fact, the same as COMPUTATIONAL. A data item that is used as a counter or subscript should be declared:

```
77 THE-NAME     PICTURE S9(10) COMPUTATIONAL.
```

COMPUTATIONAL items are always word-aligned no matter at what level they are defined and thus are equally efficient. However, there are some things which must be observed.

1. If the number of digits is greater than 10, the item becomes double-word computational. If you have the BIS compiler on TOPS-10 or you are using TOPS-20, all arithmetic is done in-line using double-precision instructions. Otherwise, all arithmetic is done with calls to the object-time system. However, it is still faster and more efficient than DISPLAY.

2. It is important that the variable be signed. If it is not, much less efficient code is generated in order to insure that it is never negative.

13.4.2 File Storage

SIXBIT files are the best for file storage and data manipulation efficiencies. Not only do they require less space than ASCII or EBCDIC, but they are efficient to move about. Each SIXBIT record is preceded by a "length descriptor" that provides the information necessary to do block transfers of data in memory rather than character by character. Also, since SIXBIT records are always word aligned, they can be transferred with block transfer instructions.

ASCII is good for text which is of variable length (for example those created by EDIT) and for line control. It suffers from the necessity to process each character to determine the end of the record.

EBCDIC is necessary if more than 128 characters are needed and if data transfer to systems using EBCDIC is necessary. It is also necessary to read files character by character since EBCDIC records (fixed length) need not necessarily be word aligned. It can be somewhat more efficiently processed than ASCII, however, since a specified number of characters is always transferred rather than an arbitrary number.

13.4.3 Blocking Data

Processing data from disk is more efficient if it is not blocked. This allows the system to pack information as tightly as possible on the disk with no slack bytes between blocks. Blocks always start on one of the disk's 128-word sector boundaries. Thus blocking inefficiently could waste considerable space.

If you block disk records, remember to count the length descriptor words on SIXBIT and variable-length EBCDIC records. Records for these two data types are also word aligned.

13.5 EFFICIENT CODING CONVENTIONS

This section contains a listing of some practical coding practices that have proven efficient. You can show these to be beneficial by writing short programs that execute these sequences a large number of times. It is also possible to look at the MACRO expansion of the program to see what is different about the compiled code.

13.5.1 Alignment

When the addresses of the data items are known at compile time, the compiler can generate efficient in-line code. This code can include the use of the block transfer instruction where the two data items are aligned and of the same type. When they are not aligned, or when conversion is necessary, an object-time system routine can be called.

The simplest way to insure that data is aligned is to define it at either the "77" level or at the "01" level. It is possible by counting characters or by using the COBOL data map to also determine alignment.

Alignment simply means that the first byte of each item begins in the same position in the beginning word, that the items are the same length, and that they are of the same type.

13.5.2 Use Of Subscripts

Avoid the use of subscripts whenever possible. Subscripts are recomputed every time they are used, they are never remembered. If you use a subscripted item more than once, it is more efficient to move it into a simple variable and then use that. For example:

```
 01 THE-TABLE OCCURS 200 TIMES
  02 THE-COUNT PIC S9(10) COMP.
  02 THE-DATA PIC XXXXXXXX.

77 THE-TABLE-COUNT PIC S9(10) COMP.
```

The sequence:

```
MOVE THE-COUNT(IDX) TO THE-TABLE-COUNT.
IF THE-TABLE-COUNT = 3 GO TO P-1.
IF THE-TABLE-COUNT = 4 GO TO P-2.
```

is more efficient if it is likely that the count is not 3, than the following:

```
IF THE-COUNT(IDX) = 3 GO TO P-1.
IF THE-COUNT(IDX) = 4 GO TO P-1.
```

It is usually advantageous to move the whole entry from a table into some 01-level structure which contains similar items rather than to process the data from the table with subscripts. For example:

```
01 THE-TABLE OCCURS 20 TIMES.
  02 THE-CNT PIC S9(10) COMP.
  02 THE-DATA PIC XXXXXXX.

01 THE-TABLE-ENTRY.
  02 THE-TABLE-CNT PIC S9(10) COMP.
  02 THE-TABLE-DATA XXXXXXX.

MOVE THE-TABLE(IDX) TO THE-TABLE-ENTRY.
IF THE-TABLE-CNT = 5 DISPLAY THE-TABLE-DATA.
```

In this example, only one subscript had to be calculated, and one unsubscripted move performed. Savings in often-referenced paragraphs (in a loop) can be quite large. Simply remember that there is additional overhead involved in subscripting and it pays to eliminate it.

13.5.3 Incrementing Counters

COBOL-68 provides three ways of incrementing counters. Each performs the same function in different ways. For example:

```
77 COUNTER PIC S9(10) COMP.
```

This counter can be modified in the following ways:

```
SET COUNTER UP BY 1.

ADD 1 TO COUNTER.

COMPUTE COUNTER = COUNTER +1.
```

The first two examples are equivalent, the third is much slower and, therefore, not recommended.

Keep in mind that computational counters should always be signed even when they logically never become negative. If they are not signed, the compiler must generate additional instructions to make sure they do not become negative.

13.5.4 The PERFORM Statement

The PERFORM statement provides an essential element of structured programming. It provides implicit loop control and it makes listings easy to follow. The power of the statement is not provided without some cost, however. Every entrance to a routine requires that some information be posted, and every exit from a routine requires that some information be cleared. Because of the complexity of these operations, COBOL-68 uses the concept of level. Each time a PERFORM statement is encountered, the level counter is incremented by 1. Each time a performed routine exits, it is decremented by 1. The level counter must have the same value at exit time as it has at entry or else there is an error in the program.

Here are a few known ways to improve the efficiency of programs that use PERFORMs.

Example 13-1

```
              SET IDX TO 0    PERFORM PAR1 100 TIMES.
              .
              .
     PAR1.      SET IDX UP BY 1.
              IF TABLE(IDX) = ABLE MOVE 6 TO FOO.
              .
```

Example 13-1 is more efficient than:

```
     PERFORM PAR2 VARYING IDX FROM 2 BY 1
          UNTIL IDX > 100.
```

When a loop or PERFORM is done repeatedly, the loop should do everything possible on each iteration. This minimizes the expense of the loop control mechanism. Thus:

```
     PERFORM F-1 1000 TIMES.
     PERFORM F-2 1000 TIMES.
```

should be rewritten so that both functions of F-1 and F-2 can be accomplished by a single PERFORM. This is most meaningful when the amount of work being accomplished by each paragraph is small.

13.5.5 Use Of The EXAMINE Statement

Use of the EXAMINE statement is preferable to doing the same process in other ways. You should understand all the options (including REPLACING) so that the power of the statement can be applied. For information on the EXAMINE statement see Part 2, COBOL Language Reference Material.

13.5.6 Data Movement

If you are moving data from one record to another and the records differ in their usages, the object-time system has to convert the data from one character-set to another. In this case, it is more efficient to change all fields in a record with one MOVE statement than to move the data field by field. If, on the other hand, the usages are the same for the two records, the compiler recognizes that data conversion is not necessary. If the records are also aligned, then efficient in-line code can be generated. Due to the method of moving data used by TOPS-10 and TOPS-20, however, the fixed cost to move any number of characters is higher than on some systems. You should therefore try to avoid loops where small numbers of characters are continually being transferred. If it is impossible to avoid such situations, then make sure that the data is aligned.

13.5.7 Ordering Statements

All programs should be written so that they avoid executing large numbers of useless instructions. Thus classic decision lists like:

```
     IF AB = " " GO TO FOO.
     IF CD = "1" GO TO FOO-1
     IF EF = "2" GO TO FOO-2.
                 .
                 .
     IF GH = "3" GO TO FOO-3
```

should be ordered by expected frequency. The following type of coding should be avoided if it is in a highly used spot:

```
     IF AB = " " MOVE Z TO DDD.
     IF AB = "1" MOVE Z TO FFF.
     IF AB = "2" MOVE Z TO GGG.
```

In this type of code the conditional clauses of all statements get executed each time, even though only one actually does anything useful.

13.5.8 Asking The Correct Question

Some small efficiencies can be gained by asking the correct questions. Thus the following example is inefficient.

```
             SET X TO 1.
     LOOP.   MOVE B(X) TO C(X).
             SET X UP BY 1.
             IF X > 1000 GO TO ZIP.
             GO TO LOOP.
```

While this is not bad coding, the program only goes to ZIP one time in a 1000. Some better code is developed if the statement were rewritten:

```
     IF X < 1001 GO TO LOOP ELSE GO TO ZIP.
```

The first option is the one that happens the most often.

APPENDIX A

COBOL RESERVED WORDS

In the listing below, words not preceded by symbols are reserved in both ANSI-68 Standard COBOL and in COBOL-68. Words preceded by '*' are reserved in ANSI-68 Standard COBOL but not reserved in COBOL-68. Words preceded by '**' are reserved in COBOL-68 but not reserved in ANSI-68 Standard COBOL. Reserved words can not be used as user-created names.

A

ACCEPT	ACCESS	ACTUAL
ADD	*ADDRESS	ADVANCING
AFTER	ALL	**ALLOWING
ALPHABETIC	ALTER	ALTERNATE
AND	**ANY	ARE
AREA	AREAS	ASCENDING
**ASCII	ASSIGN	AT
AUTHOR		

B

BEFORE	BEGINNING	**BINARY
BLANK	BLOCK	BY
**BYTE		

C

**CALL	**CANCEL	**CD
CF	CH	CHARACTERS
**CHECK	**CHECKPOINT	**CLASS
*CLOCK-UNITS	CLOSE	COBOL
CODE	COLUMN	COMMA
**COMMUNICATION	COMP	**COMP-1

COBOL RESERVED WORDS

**COMP-3	**COMPILE	COMPUTATIONAL
**COMPUTATIONAL-1	**COMPUTATIONAL-3	COMPUTE
CONFIGURATION	**CONSOLE	CONTAINS
CONTROL	CONTROLS	COPY
CORR	CORRSPONDING	**COUNT
CURRENCY	**CURRENT	

D

DATA	**DATABASE-KEY	**DATE
**DATE-COMPILED	DATE-WRITTEN	**DBKEY
DE	**DEC	DECIMAL-POINT
DECLARATIVES	**DECSYSTEM-10	**DECSYSTEM-20
**DECSYSTEM10	**DEFERRED	**DELETE
**DELIMITED	**DELIMITER	**DENSITY
DEPENDING	**DEPTH	DESCENDING
**DESTINATION	DETAIL	**DISABLE
DISPLAY	**DISPLAY-6	**DISPLAY-7
**DISPLAY-9	DIVIDE	DIVISION
DOWN	**DUP	**DUPLICATE

E

**EGI	ELSE	**EMI
**EMPTY	**ENABLE	END
ENDING	ENTER	**ENTRY
ENVIRONMENT	**EPI	EQUAL
**EQUALS	ERROR	**ESI
**EVEN	EVERY	EXAMINE
**EXCL	**EXCLUSIVE	EXIT
**EXTEND		

F

FD	FILE	FILE-CONTROL
FILE-LIMIT	FILE-LIMITS	**FILE-STATUS
FILLER	FINAL	**FIND

COBOL RESERVED WORDS

FIRST	FOOTING	FOR
**FORTRAN-IV	**FORTRAN	**FREE
FREED	FROM	
G		
GENERATE	**GET	GIVING
GO	**GOBACK	GREATER
GROUP		
H		
HEADING	HIGH-VALUE	HIGH-VALUES
I		
I-O	I-O CONTROL	**ID
IDENTIFICATION	IF	IN
INDEX	INDEXED	INDICATE
**INITIAL	INITIATE	INPUT
INPUT-OUTPUT	**INSERT	INSTALLATION
INTO	INVALID	**INVOKE
IS		
J		
**JOURNAL	JUST	JUSTIFIED
K		
KEY	KEYS	
L		
LABEL	LAST	LEADING
LEFT	*LENGTH	LESS
LIMIT	LIMITS	LINE
*LINE-COUNTER	LINES	**LINKAGE
LOCK	LOW-VALUE	LOW-VALUES
M		
**MACRO	**MEMBER	**MEMBERS

MEMORY	**MERGE	**MESSAGE
MODE	**MODIFY	MODULES
MOVE	MULTIPLE	MULTIPLY

N

NEGATIVE	NEXT	NO
**NOMINAL	**NONE	NOT
NOTE	NUMBER	NUMERIC

O

OBJECT-COMPUTER	OCCURS	**ODD
OF	OFF	OMITTED
ON	**ONLY	OPEN
**OPT	OPTIONAL	OR
OTHERS	OUTPUT	**OVERFLOW
**OWNER		

P

*PAGE-COUNTER	**PARITY	**PDP-10
PERFORM	PF	PH
PIC	PICTURE	PLUS
**POINTER	POSITION	**POSITIONING
POSITIVE	**PRIOR	**PRIVACY
PROCEDURE	PROCEED	PROCESSING
**PROGRAM	PROGRAM-ID	**PROT
**PROTECTED		

Q

**QUEUE	QUOTE	QUOTES

R

RANDOM	RD	READ
**READ-REWRITE	**READ-WRITE	**RECEIVE
RECORD	**RECORDING	RECORDS
REDEFINES	REEL	**RELATIVE

COBOL RESERVED WORDS

RELEASE	**REMAINDER	REMARKS
**REMOVE	RENAMES	REPLACING
REPORT	REPORTING	REPORTS
RERUN	RESERVE	RESET
**RETAIN	**RETAINED	**RETR
**RETRIEVAL	RETRUN	REVERSED
REWIND	**REWRITE	RF
RH	RIGHT	ROUNDED
RUN	**RUN-UNIT	
S		
SAME	**SCHEMA	SD
SEARCH	SECTION	SECURITY
SEEK	**SEGMENT	SEGMENT-LIMIT
SELECT	**SELECTIVE	**SEND
SENTENCE	**SEQUENCE	SEQUENTIAL
SET	**SETS	SIGN
**SIXBIT	SIZE	SORT
SOURCE	SOURCE-COMPUTER	SPACE
SPACES	SPECIAL-NAMES	STANDARD
**STANDARD-ASCII	STATUS	STOP
**STORE	**STRING	**SUB-QUEUE-1
**SUB-QUEUE-2	**SUB-QUEUE-3	**SUB-SCHEMA
SUBTRACT	SUM	**SUPPRESS
**SWITCH	**SYMBOLIC	SYNC
SYNCHRONIZED		
T		
**TABLE	TALLY	TALLYING
TAPEL	**TERMINAL	TERMINATE
**TEXT	THAN	THROUGH
THRU	**TIME	TIMES
TO	TODAY	**TRACE

COBOL RESERVED WORDS

**TRANSACTION	TYPE	

U

**UNAVAILABLE	UNIT	**UNSTRING
UNTIL	UP	**UPDATE
**UPDATES	UPON	USAGE
**USAGE-MODE	USE	**USER-NUMBER
USING		

V

VALUE	VALUES	VARYING
**VERB	**VIA	

W

WHEN	WITH	**WITHIN
WORDS	WORKING-STORAGE	WRITE

Z

ZERO	ZEROES	ZEROS

APPENDIX B

COLLATING SEQUENCES AND CONVERSION TABLES

Table B-1 shows the ASCII and SIXBIT collating sequence and the conversions from ASCII to EBCDIC, SIXBIT to ASCII, and SIXBIT to EBCDIC. If the ASCII character does not convert to the same character in EBCDIC, the EBCDIC character is shown in parentheses next to the EBCDIC code. Note that the first and last 32 characters do not exist in SIXBIT. Also, the characters in the first column (NUL, SOH, STX, and so forth) are control characters, which are nonprinting.

Table B-1
ASCII and SIXBIT Collating Sequence and Conversion to EBCDIC

Character	ASCII 7-bit	EBCDIC 9-bit
NUL	000	000
SOH	001	001*
STX	002	002*
ETX	003	003*
EOT	004	067
ENQ	005	055*
ACK	006	056*
BEL	007	057*
BS	010	026
HT	011	005
LF	012	045
VT	013	013*
FF	014	014*
CR	015	025*(NL)
SO	016	006*(LC)
SI	017	066*(UC)
DLE	020	044*(BYP)
DC1	021	024*(RES)
DC2	022	064*(PN)
DC3	023	065*(RS)
DC4	024	004*(PF)
NAK	025	075*
SYN	026	027*(IL)
ETB	027	046*(EOB)
CAN	030	052*(CM)
EM	031	031*
SUB	032	032*(CC)
ESC	033	047*(PRE)
FS	034	023*(TM)
GS	035	041*(SOS)
RS	036	040*(DS)
US	037	042*(FS)

Character	SIXBIT	ASCII 7-bit	EBCDIC 9-bit
Space	00	040	100
!	01	041	132
"	02	042	177
#	03	043	173
$	04	044	133
%	05	045	154
&	06	046	120
'	07	047	175
(	10	050	115
)	11	051	135
*	12	052	134
+	13	053	116
,	14	054	153
-	15	055	140
.	16	056	113
/	17	057	141
0	20	060	360
1	21	061	361
2	22	062	362
3	23	063	363
4	24	064	364
5	25	065	365
6	26	066	366
7	27	067	367
8	30	070	370
9	31	071	371
:	32	072	172
;	33	073	136
<	34	074	114
=	35	075	176
>	36	076	156
?	37	077	157

Table B-1 (Cont.)
ASCII and SIXBIT Collating Sequence and Conversion to EBCDIC

Character	SIXBIT	ASCII 7-bit	EBCDIC 9-bit	Character	ASCII 7-bit	EBCDIC 9-bit
@	40	100	174	\	140	171
A	41	101	301	a	141	201
B	42	102	302	b	142	202
C	43	103	303	c	143	203
D	44	104	304	d	144	204
E	45	105	305	e	145	205
F	46	106	306	f	146	206
G	47	107	307	g	147	207
H	50	110	310	h	150	210
I	51	111	311	i	151	211
J	52	112	321	j	152	221
K	53	113	322	k	153	222
L	54	114	323	l	154	223
M	55	115	324	m	155	224
N	56	116	325	n	156	225
O	57	117	326	o	157	226
P	60	120	327	p	160	227
Q	61	121	330	q	161	230
R	62	122	331	r	162	231
S	63	123	342	s	163	242
T	64	124	343	t	164	243
U	65	125	344	u	165	244
V	66	126	345	v	166	245
W	67	127	346	w	167	246
X	70	130	347	x	170	247
Y	71	131	350	y	171	250
Z	72	132	351	z	172	251
[	73	133	255[1]	{	173	300[1]
\	74	134	340	|	174	117
]	75	135	275	}	175	320
^	76	136	137	~	176	241
_	77	137	155	Delete	177	007

1. These EBCDIC codes either have no equivalent in the ASCII or SIXBIT character sets, or are referred to by different names. They are converted to the indicated ASCII characters to preserve their uniqueness if the ASCII character is converted back to EBCDIC.

Table B-2 shows the conversion of ASCII code to SIXBIT code. The table does not show ASCII codes 000 through 037 because they all convert to SIXBIT 74 (\), except 11 (TAB) which converts to SIXBIT 00 (space).

Table B-2
ASCII to SIXBIT Conversion

Character	ASCII 7-bit	SIXBIT	Character	ASCII 7-bit	SIXBIT
Space	040	00	@	100	40
!	041	01	A	101	41
"	042	02	B	102	42
#	043	03	C	103	43
$	044	04	D	104	44
%	045	05	E	105	45
&	046	06	F	106	46
'	047	07	G	107	47
(	050	10	H	110	50
)	051	11	I	111	51
*	052	12	J	112	52
+	053	13	K	113	53
,	054	14	L	114	54
-	055	15	M	115	55
.	056	16	N	116	56
/	057	17	O	117	57
0	060	20	P	120	60
1	061	21	Q	121	61
2	062	22	R	122	62
3	063	23	S	123	63
4	064	24	T	124	64
5	065	25	U	125	65
6	066	26	V	126	66
7	067	27	W	127	67
8	070	30	X	130	70
9	071	31	Y	131	71
:	072	32	Z	132	72
;	073	33	[	133	73
<	074	34	\	134	74
=	075	35	]	135	75
>	076	36	^	136	76
?	077	37	_	137	77

Table B-2 (Cont.)
ASCII to SIXBIT Conversion

ASCII code	ASCII character	SIXBIT code	SIXBIT character
140	\	74	\
141	a	41	A
142	b	42	B
143	c	43	C
144	d	44	D
145	e	45	E
146	f	46	F
147	g	47	G
150	h	50	H
151	i	51	I
152	j	52	J
153	k	53	K
154	l	54	L
155	m	55	M
156	n	56	N
157	o	57	O
160	p	60	P
161	q	61	Q
162	r	62	R
163	s	63	S
164	t	64	T
165	u	65	U
166	v	66	V
167	w	67	W
170	x	70	X
171	y	71	Y
172	z	72	Z
173	{	73	[
174	|	74	\
175	}	75	]
176	~	74	\
177	delete	74	\

Table B-3 shows the EBCDIC collating sequence and the conversion from EBCDIC to ASCII. When conversion is from EBCDIC to SIXBIT, it is as if the code was converted to ASCII and then from ASCII to SIXBIT.

Table B-3
EBCDIC Collating Sequence and Conversion to ASCII

EBCDIC code	EBCDIC character	ASCII code	ASCII character
000	NUL	000	NUL
001	SOH	001	SOH
002	STX	002	STX
003	ETX	003	ETX
004	PF	024	DC4
005	HT	011	HT
006	LC	016	SO
007	Delete	177	Delete
010		134	\
011		134	\
012	SMM	134	\
013	VT	013	VT
014	FF	014	FF
015	CR	134	\
016	SO	134	\
017	SI	134	\
020	DLE	134	\
021	DC1	134	\
022	DC2	134	\
023	TM	034	FS
024	RES	021	DC1
025	NL	015	CR
026	BS	010	BS
027	IL	026	SYN
030	CAN	134	\
031	EM	031	EM
032	CC	032	SUB
033	CU1	134	\
034	IFS	134	\
035	IGS	134	\
036	IRS	134	\
037	IUS	134	\
040	DS	036	RS
041	SOS	035	GS
042	FS	037	US
043		134	\
044	BYP	020	DLE
045	LF	012	LF
046	ETB	027	ETB
047	ESC	033	ESC
050		134	\
051		134	\
052	SM	030	CAN
053	CUZ	134	\
054		134	\
055	ENQ	005	ENQ
056	ACK	006	ACK
057	BEL	007	BEL
060		134	\
061		134	\
-62		134	\
063		134	\
064	PN	022	DC2
065	RS	023	DC3
066	UC	017	SI
067	EOT	004	EOT
070		134	\
071		134	\
072		134	\
073		134	\
074	CU3	134	\
075	DC4	025	NAK
076	NAK	134	\
077	SUB	134	\
100	Space	040	Space
101		134	\
102		134	\
103		134	\
104		134	\
105		134	\
106		134	\
107		134	\
110		134	\
111		134	\
112	CENT	134	\
113	.	056	.
114	<	074	<
115	(	050	(
116	+	053	+
117		174	

Table B-3 (Cont.)
EBCDIC Collating Sequence and Conversion to ASCII

EBCDIC code	EBCDIC character	ASCII code	ASCII character	EBCDIC code	EBCDIC character	ASCII code	ASCII character
120	&	046	&	170		134	\
121		134	\	171		140	\
122		134	\	172	:	072	:
123		134	\	173	#	043	#
124		134	\	174	@	100	@
125		134	\	175	'	47	'
126		134	\	176	=	075	=
127		134	\	177	"	042	"
130		134	\	200		134	\
131		134	\	201	a	141	a
132	!	041	!	202	b	142	b
133	$	044	$	203	c	143	c
134	*	052	*	204	d	144	d
135	)	051	)	205	e	145	e
136	^	073	^	206	f	146	f
137		137	\	207	g	147	g
140	-	055	-	210	h	150	h
141	/	057	/	211	i	151	i
142		134	\	212		134	\
143		134	\	213		134	\
144		134	\	214		134	\
145		134	\	215		134	\
146		134	\	216		134	\
147		134	\	217		134	\
150		134	\	220		134	\
151		134	\	221	j	152	j
152		134	\	222	k	153	k
153	,	054	,	223	l	154	l
154	%	045	%	224	m	155	m
155	_	137	_	225	n	156	n
156	>	076	>	226	o	157	o
157	?	077	?	227	p	160	p
160		134	\	230	q	161	q
161		134	\	231	r	162	r
162		134	\	232		134	\
163		134	\	233		134	\
164		134	\	234		134	\
165		134	\	235		134	\
166		134	\	236		134	\
167		134	\	237		134	\

Table B-3 (Cont.)
EBCDIC Collating Sequence and Conversion to ASCII

EBCDIC code	EBCDIC character	ASCII code	ASCII character	EBCDIC code	EBCDIC character	ASCII code	ASCII character
240		134	\	310	H	110	H
241		176		311	I	110	I
242	s	163	s	312		134	\
243	t	164	t	313		134	\
244	u	165	u	314		134	\
245	v	166	v	315		134	\
246	w	167	w	316		134	\
247	x	170	x	317		134	\
250	y	171	y	320		175	
251	z	172	z	321	J	112	J
252		134	\	322	K	113	K
253		134	\	323	L	114	L
254		134	\	324	M	115	M
255	[	133	[	325	N	116	N
256		134	\	326	O	117	O
257		134	\	327	P	120	P
260		175		330	Q	121	Q
261		134	\	331	R	122	R
262		134	\	332		134	\
263		134	\	333		134	\
264		134	\	334		134	\
265		134	\	335		134	\
266		134	\	336		134	\
267		134	\	337		134	\
270		134	\	340		134	\
271		134	\	341		134	\
272		134	\	342	S	123	S
273		134	\	343	T	124	T
274		134	\	344	U	125	U
275	]	135	]	345	V	126	V
276		134	\	346	W	127	W
277		134	\	347	X	130	X
300		173		350	Y	131	Y
301	A	101	A	351	Z	132	Z
302	B	102	B	352		134	\
303	C	103	C	353		134	\
304	D	104	D	354		134	\
305	E	105	E	355		134	\
306	F	106	F	356		134	\
307	G	107	G	357		134	\
360	0	060	1	370	8	070	8
361	1	061	1	371	9	071	9
362	2	062	2	372		134	\
363	3	063	3	373		134	\
364	4	064	4	374		134	\
365	5	065	5	375		134	\
366	6	066	6	376		134	\
367	7	067	7	377		134	\

APPENDIX C

DEFINING LOGICAL NAMES UNDER TOPS-20

Most of the file specifications for the COBOL compiler and the utilities associated with COBOL-68 use project-programmer numbers to identify areas on the disk. Users of TOPS-20 do not normally deal with project-programmer numbers; named directories are used instead. However, the compiler and the utilities often do not accept named directories in the command strings. There are two ways for TOPS-20 users to specify a directory to be searched. One is to use the TRANSL command to translate a named directory to a project-programmer number. This way is perfectly functional, but usually inconvenient. The other way is to define a logical name and use it in the command string in place of the device name and the project-programmer number. The TRANSL and DEFINE commands are described in the TOPS-20 User's Guide. Refer to that manual for more information on these two commands. A short description of the DEFINE command has been included here for convenience.

The DEFINE command has the following format:

```
@DEFINE (LOGICAL NAME) logname: (AS) filespecs
```

where:

logname:	is the logical name being defined. It consists of up to 6 alphanumeric characters (A-Z and 0-9 only) followed by a colon.
filespecs	is a list of file specifications (separated by commas) that define the logical name. A file specification can contain any combination of a structure name, device name, directory, file name, file type, generation number, and wildcards. If you wish to remove a logical name, you should leave the filespecs entry blank.

The following characteristics of the DEFINE command should be noted:

1. The DEFINE command is used at TOPS-20 monitor level (or in a batch or command file). The command does not alter any program and leaves you at monitor level.

2. Some programs can expect certain logical names to be defined certain ways. You should exercise caution in deciding on a character string to use as a logical name. See the INFORMATION command in the TOPS-20 User's Guide for a description of how to determine what logical names are already defined.

Example:

```
DEFINE PR:  <PAYROLL>
```

allows you to type the following command to the COBOL-68 compiler:

```
PR:FEDTAX=TESTFT.CBL
```

This command string takes a file in your connected directory named TESTFT.CBL and compile it, writing the .REL file in the directory <PAYROLL>. As written, the command string would also write the .LST file to your connected directory. If you wish to have it in the <PAYROLL> directory you must use the following command:

```
PR:FEDTAX,PR:FEDTAX=TESTFT.CBL
```

APPENDIX D

ALTERNATE NUMERIC TEST

LIBOL as normally assembled includes the ANSI standard NUMERIC test. However, an assembly switch has been provided to allow the installation manager to replace this with the ALTERNATE NUMERIC test at installation time.

The ALTERNATE NUMERIC test result is TRUE under the following conditions:

1. For alphanumeric and unsigned numeric items, each character must be a digit (0 through 9). Leading and trailing spaces and leading and trailing tabs are ignored. No signs are permitted.

2. For signed numeric items, the sign can have only one of the three following representations:

    1. A leading graphic sign ("+" or "-"),

    2. A trailing graphic sign, or

    3. A trailing embedded sign.

    Leading and trailing spaces and leading and trailing tabs are ignored. All other characters must be digits.

In this case, you do not want to use a STANDARD-ASCII tape, so you do not use the SET TAPE FORMAT command. If your COBOL program is running on a TOPS-10 system, you can use the simplest TOPS-10 MOUNT command:

```
     MOUNT tapnam:.....
```

If your program is running on a TOPS-20 system, you must use a logical name in the MOUNT command to allow you to set the attribute ;FORMAT (with the DEFINE command). You must set this attribute to avoid confusion with the case where you wish to write an ASCII U-format tape. On TOPS-20, you can write both F-format and D-format tapes, so you can use either ;FORMAT:F or ;FORMAT:D with the DEFINE command. It is also a good idea to declare the record and block size in the DEFINE command. The format is shown below (here x stands for one of the letters F and D, and nn and mm for a decimal number of bytes).

```
     MOUNT TAPE tapnam:.....

     DEFINE taplnm: tapnam: ;FORMAT:x ;RECORD:nn ;BLOCK:mm
```

E.4.1.2 **Undefined-Format Tapes - U-Format** - Undefined-format tapes include all tapes written in EBCDIC, SIXBIT, and binary recording modes, as well as tapes written in other recording modes if the label type is set to U. U-format tapes cannot be written with STANDARD-ASCII or ASCII recording modes on TOPS-10.

EBCDIC for TOPS-10

If you want to use EBCDIC recording mode on an ANSI-labeled tape, you simply declare in your COBOL program that:

```
     RECORDING MODE IS EBCDIC
```

Then you mount the tape using the simple form of the MOUNT command:

```
     MOUNT TAPE tapnam
```

The tape-labeling software recognizes the recording mode and realize that it must write a U-format tape.

EBCDIC and ASCII for TOPS-20

If you wish to use EBCDIC or ASCII recording modes on a U-format tape, you should include in your COBOL program an explicit declaration of the recording mode you want to use. You should also include a statement of the form:

```
     SELECT filnam ASSIGN TO taplnm
```

When you mount the tape, use some name that is different from taplnm, thus:

```
     MOUNT TAPE tapnam
```

Finally, use the DEFINE command to set the label type:

```
     DEFINE taplnm: tapnam: ;FORMAT:U
```

Another way for users of TOPS-20 to handle ASCII data on an undefined-format tape is to use the SET TAPE FORMAT monitor command.

APPENDIX E

TAPE HANDLING

E.1 DIRECTIONS AND DEFINITIONS

This appendix describes in detail the methods you use to handle tapes with COBOL programs. The appendix does not describe the use of the MOUNT command; for this information, see the TOPS-10 Operating System Commands Manual or the TOPS-20 Commands Reference Manual. This appendix also does not describe the procedures used by the operating system in handling tapes and tape drives. For more information on this subject, consult the TOPS-10 or TOPS-20 Tape Processing Manual.

E.1.1 Definitions

Several terms used in this appendix require explicit definition.

COBOL Labels

Labels written by a COBOL program that includes the LABEL RECORDS clause. COBOL labels and system labels (see definition below) are mutually exclusive. You cannot write COBOL labels onto a system-labeled tape; nor can you write system labels onto a tape that has COBOL labels, since you must initialize a tape to write a system label on it.

Label Type

A field in the system label that defines the structure of records on the tape. There are four label types: F, meaning fixed-length records, D, indicating variable-length records, U, signifying that an undefined format has been used to write the records, and S, meaning spanned records.

Object-Time System

The object-time system consists of LIBOL.REL and LIBO12.EXE.

System Labels

These are labels written by the TOPS-10 or TOPS-20 operating system. Labels are written on a tape by the OPR program when it initializes the tape. You must therefore ask the operator to initialize your tape if you do not have the privileges required to run OPR. COBOL labels and system labels are mutually exclusive.

Volume

A single magnetic tape; a reel.

Volume Number

A number signifying the position of the reel in the volume set.

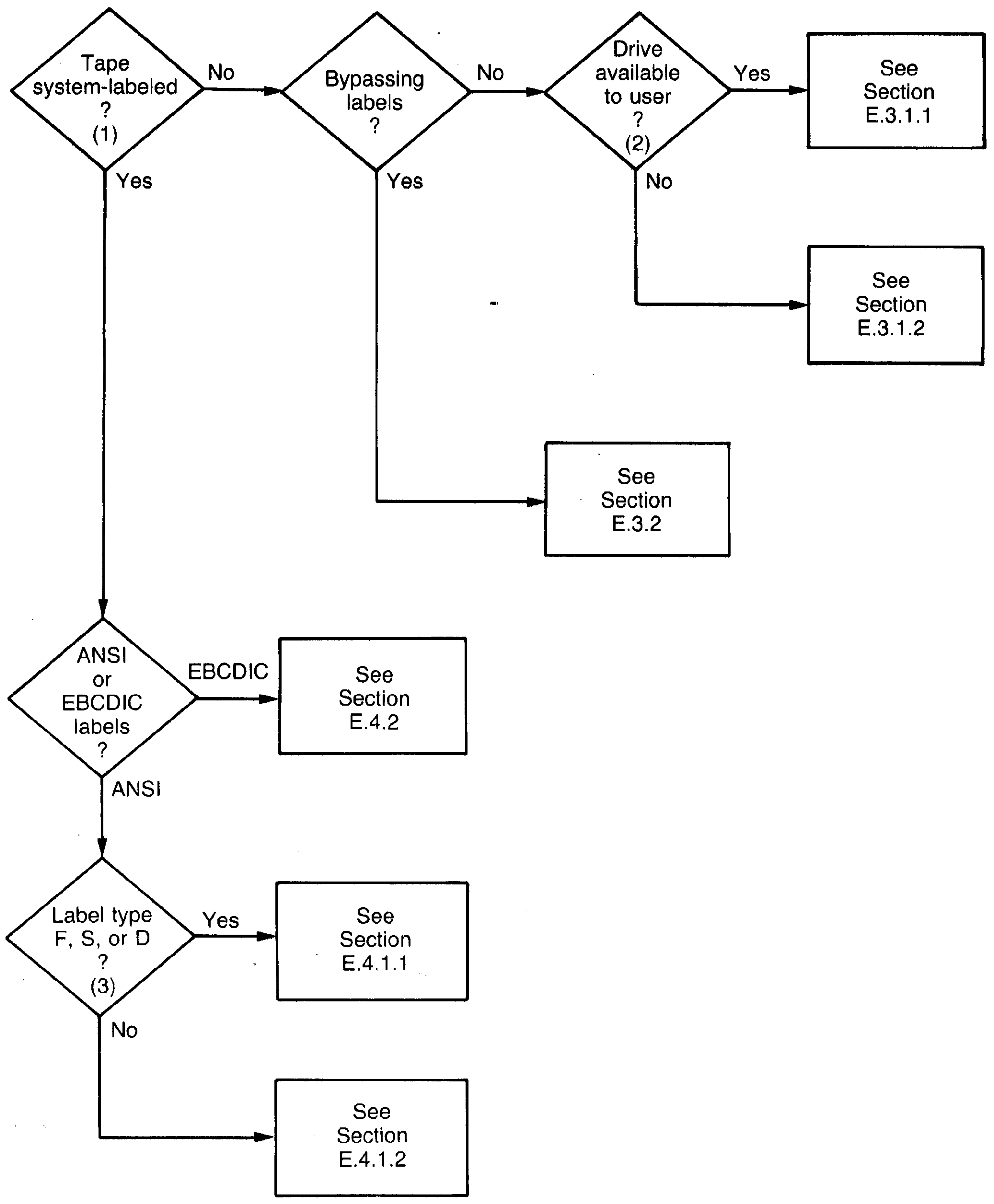
Volume Set

A group of magnetic tapes that are accessed as a single unit.

E.1.2 Finding The Right Instructions

This appendix presents a number of specific methods of labeled-tape handling that are recommended ways to use the tape-handling software. Many situations can be devised that do not fit exactly into one of the cases listed here. If you attempt to use the tape-labeling software in one of these ways, the results are not predictable. Therefore, you are advised to set up a system of handling tapes at your site that is consistent with the methods discussed in this appendix.

The flowchart below helps to guide you to the section that describes the type of tape handling you are doing.



(1) See definition of system labels in Section E.1.2.

(2) A tape drive is either assigned by the system tape–handling software or assigned by you.

(3) Tapes with F–format, S–format, and D–format ANSI labels are transportable to other systems, and must be written in either ASCII or STANDARD–ASCII recording mode.

MR-S-1377-81

E.1.3 Symbols Used In The Text

The sample ASSIGN clauses and MOUNT commands that follow contain text strings that represent user-supplied information. The strings, and the information they represent, are shown below.

String	Represents	Where used
tapnam	Name of tape	ASSIGN clause, MOUNT command
taplnm	Logical name	ASSIGN clause, MOUNT command
filnam	Name of file	ASSIGN clause
xxx	Any legal switch	MOUNT command
idN (N=0,1,2,...)	Volume ID	MOUNT command

E.2 FACTORS TO CONSIDER WHEN USING TAPES

Several new situations have been introduced into the COBOL environment by the addition of software to TOPS-10 and TOPS-20 for handling labeled tapes. This section presents some factors you should consider when deciding what type of tape labeling to use.

E.2.1 General Defaults And Restrictions

The following defaults and restrictions apply to tape handling on both TOPS-10 and TOPS-20.

1. If you are using system labels, no COBOL labels are read or written on your tape. The object-time system can tell whether the system tape-handling software found labels on your tape; if system labels were found, the object-time system does not write or read COBOL labels.

2. All STANDARD-ASCII tapes (explicit or default) are written without carriage return/line feed pairs unless you introduce the carriage return/line feeds by using the ADVANCING clause.

3. COBOL-68 USE procedures for END-OF-REEL no longer work if you use system labeled tapes. This is because the EOT (End Of Tape) condition is never seen by the COBOL object-time system; the system tape-handling software intervenes when EOT is reached and requests the next tape from the operator. The only time a USE procedure for END-OF-REEL is executed is when the program contains a CLOSE REEL statement. When this statement is executed, the USE procedure is performed.

4. If you have multiple devices in the ASSIGN statement in your COBOL program, your system labeled tapes use only the first device in the list. The only exception to this is when you use the CLOSE REEL syntax to end one tape and start another. This syntax causes the system tape-handling software to see one volume set being closed, rather than one reel. This allows COBOL to switch to the next device in the ASSIGN statement while the system tape-handling software switches to the next volume set, as specified in the next MOUNT command you gave.

5. You cannot write an unblocked tape with an F-format or D-format label. If your COBOL program declares the blocking factor to be zero, or if you omit the BLOCK CONTAINS clause, the object-time system acts as if you had declared BLOCK CONTAINS 1 RECORD.

6. The COBOL compiler and object-time system do not always enforce the description of your file given in the FD when you are writing tapes. In particular, you could write variable-length records to an F-format ANSI-labeled tape without receiving any errors. The data on the tape would be padded with nulls to the specified maximum record size.

E.2.2 Defaults And Restrictions Specific To TOPS-20 Systems

The following defaults and restrictions apply to tape handling on TOPS-20 only.

1. The default recording mode for TOPS-20 systems is determined by a fairly simple algorithm. If you use the RECORDING MODE IS STANDARD-ASCII clause in your program, and you set no tape attributes (using the DEFINE command), your tape becomes D-format. If your RECORDING MODE IS ASCII, and you give the SET TAPE FORMAT ANSI-ASCII command at monitor level, your tape again becomes D-format. You can also get a D-format tape by omitting the RECORDING MODE clause from your program and giving the SET TAPE FORMAT ANSI-ASCII command. In all other default cases, TOPS-20 writes a U-format tape. TOPS-20 never defaults to F-format. (This does not mean that you cannot write an F-format tape if you specify that you wish to do so.)

2. Users of TOPS-20 who wish to write compatible tapes should set record and block length attributes (using the DEFINE command - see examples below) to make explicit the exact sizes of the record and block. Setting attributes is advisable because the tape-handling software allows several attributes to default to values that can conflict with the declarations in your program. When you are determining the size of any record, you must count all the characters in the record, including explicit and implicit advancing characters and header characters (see the TOPS-20 Tape Processing Manual for more information on the format of record headers).

3. The TOPS-20 user-set attributes (set with the DEFINE command) are the overriding factor in determining the label information when you are writing a labeled tape. When you are reading a labeled tape, the overriding factor is the information written in the tape's label.

4. You can write STANDARD-ASCII tapes (both labeled and unlabeled) by declaring the recording mode to be STANDARD-ASCII, as you would expect. You can also write STANDARD-ASCII tapes by using the SET TAPE FORMAT ANSI-ASCII command at monitor level and declaring the recording mode to be ASCII or allowing the recording mode to default to SIXBIT.

5. TOPS-20 cannot write EBCDIC labels, but it can read them.

E.2.3 Defaults And Restrictions Specific To TOPS-10 Systems

The following defaults and restrictions apply to tape handling on TOPS-10 only.

1. Both the TOPS-10 monitor and the TOPS-10 COBOL object-time system use F-format by default.

2. TOPS-10 can both read and write EBCDIC labels.

3. The COBOL object-time system running under TOPS-10 cannot read or write S-format or D-format labeled tapes.

4. TOPS-10 cannot write a U-format tape in STANDARD-ASCII or ASCII recording modes. The object-time system causes all such tapes to be written with F-format labels.

E.2.4 Converting Tapes Between Labeled And Unlabeled

System-labeled tapes can only be created with the OPR program (both TOPS-10 and TOPS-20). Since OPR writes a system label on a tape by initializing the tape, you cannot convert an unlabeled tape to a system-labeled one simply by writing a system label on it. The simplest way to create a system-labeled tape with the same data as a given unlabeled tape is to write a short COBOL program to copy the data. This COBOL program should read every record on the unlabeled tape and copy it to a tape that has been initialized and mounted as a system-labeled tape. If there are several files on a tape, the COBOL program must deal with end-of-file conditions. Once all the data on the unlabeled tape has been copied, the system-labeled version of the tape has been created; no further steps are necessary. You can create an unlabeled version of a system-labeled tape by reversing this process.

E.3 USING SYSTEM-UNLABELED TAPES

This section describes the use of tapes that the system considers to be unlabeled. This includes tapes that actually have system labels if you have told the system tape-handling software that labels are to be bypassed. The section is divided into procedures for tapes that have no system labels and procedures for tapes whose labels are bypassed.

E.3.1 **Tape Has No Labels**

If you wish to use tapes that actually have no system labels, as opposed to tapes whose system labels are bypassed, the method you should use depends on whether the tape drive can be assigned by you.

E.3.1.1 **Tape Drive Is Available To The User** - If you wish to use a tape that actually has no system labels (as opposed to bypassing labels that are on the tape), and the tape drive is available for you to assign, you can assign it and proceed with your processing exactly as you have always done. The introduction of system label processing does not interfere with this style of tape usage.

E.3.1.2 **Tape Drive Is Owned By The System** - This section contains instructions for using unlabeled tapes on tape drives assigned by the system tape-handling software. The instructions are in two parts: one part for single-reel volume sets, and one part for multiple-reel volume sets.

SINGLE-REEL VOLUME SET

To use a single tape, put the following statement into your COBOL program:

```
     SELECT filnam ASSIGN TO tapnam
```

At run time, use the following command to mount the tape:

```
     MOUNT TAPE tapnam: /LABEL-TYPE:UNLABELED /xxx
```

MULTIPLE-REEL VOLUME SET

There are three methods you can use to deal with multiple-reel volume sets. The most convenient way to use the multiple tapes is to put the following statement into your COBOL program:

```
     SELECT filnam ASSIGN TO tapnam
```

This allows you to mount the tapes at run time with the following command:

```
     MOUNT TAPE tapnam /LABEL-TYPE:UNLABELED /VOLIDS:id1,...,idN /xxx
```

where N is the number of tapes in the volume set, and idN is the volume ID of the tape that corresponds to tapnamN (the name of tape N).

It is also possible to deal with a multiple-volume set by mounting the first volume and notifying the operator of the names and volume numbers of the remaining tapes. The tape-labeling software recognizes the end of a volume and requests the operator to mount the next tape in the volume set. The MOUNT command is of the form:

```
     MOUNT TAPE tapnam /LABEL-TYPE:UNLABELED /xxx
```

Finally, you can use in your COBOL program an ASSIGN statement of the form:

```
SELECT filnam ASSIGN TO tapnam1,...,tapnamN
```

where N is the number of tapes in the volume set. If you use this form of the ASSIGN clause, you must use N MOUNT commands (which requires you to have N tape drives). Each MOUNT command is of the simplest form:

```
MOUNT TAPE tapnam1 /LABEL-TYPE:UNLABELED /xxx
        .
        .
        .
MOUNT TAPE tapnamN /LABEL-TYPE:UNLABELED /xxx
```

E.3.2 Tape Has Labels

If you wish to bypass the system labels on a tape and use the tape as if it had no labels, you can use the instructions above for unlabeled tapes. However, you should note the following exceptions:

1. You must have privileges to bypass the labels on a tape.

2. You must tell the system to bypass the system labels on the tape. The MOUNT command for doing this is:

   ```
   MOUNT TAPE tapnam /LABEL-TYPE:BYPASS /xxx
   ```

3. If you want to look at the system label in your COBOL program, simply read it as the first file on the tape.

4. If you want to skip the system label, give the REWIND and SKIP 1 commands at monitor level.

E.4 USING SYSTEM-LABELED TAPES

This section describes the use of tapes that the system considers to be labeled. This does not include tapes whose system labels are being bypassed. (For information on using this kind of tape, see Section E.3.2 above.) There are two types of system labels: ANSI labels and EBCDIC labels.

E.4.1 Tape Has ANSI Labels

Methods for using ANSI-labeled tapes can be separated into those used with transportable tapes and those used with undefined-format tapes. The first section that follows deals with transportable tapes, which have F-format, D-format, or S-format labels and must be written in STANDARD-ASCII or ASCII recording modes. The next section presents information on using U-format tapes. U-format tapes include all those recorded in EBCDIC, SIXBIT, and binary, and it is possible to write U-format tapes in ASCII and STANDARD-ASCII as well.

E.4.1.1 **Transportable Tapes - F, D, And S Formats** - You can write transportable tapes in two styles of ASCII. One kind, STANDARD-ASCII, does not contain any carriage return/line feed pairs unless your COBOL program explicitly introduces them (using the ADVANCING clause). The other kind, ASCII, does include carriage return/line feed pairs in each record unless they are specifically excluded (using, once again, the ADVANCING clause). This section presents two methods for writing STANDARD-ASCII transportable tapes that do not have implicit carriage return/line feed pairs, and another for writing ASCII transportable tapes that do have implicit carriage return/line feed pairs.

STANDARD-ASCII

There are two ways to write a tape in STANDARD-ASCII. The first method requires your COBOL program to include the following statement:

```
      RECORDING MODE IS STANDARD-ASCII
```

This allows the MOUNT command to be in its simplest form:

```
      MOUNT TAPE tapnam
```

The second method, usable only on TOPS-20, requires your COBOL program to include the following statement:

```
      RECORDING MODE IS ASCII
```

However, the data on the tape is (or can be, depending on whether you are reading or writing the tape) recorded as if you had said STANDARD-ASCII. Therefore, you must tell the object-time system to read (or write) a STANDARD-ASCII tape. To do this, type the following command at monitor level before you run your COBOL program:

```
      SET TAPE FORMAT ANSI-ASCII
```

This command must be given at monitor level, but it does not matter whether you type it before or after you mount the tape with the MOUNT command. The MOUNT command you should use is the simplest form:

```
      MOUNT TAPE tapnam
```

You can produce the same kind of tape by omitting the RECORDING MODE clause from your COBOL program altogether, assuming that you have not changed the default recording mode (with the /X compiler switch or the DISPLAY IS clause). In this case, your recording mode defaults to STANDARD-ASCII, despite the fact that the usual default recording mode is SIXBIT. If you omit the clause, you must set the default tape format as shown in the previous case, and you can use the simple MOUNT command format shown for the previous case as well.

ASCII

If you wish to write a transportable tape and you also want carriage return/line feed pairs in each record, you can use regular ASCII recording mode. To do this, you must include in your COBOL program a statement of the following type:

```
      SELECT filnam ASSIGN TO taplnm
```

This must be followed by the statement:

```
      RECORDING MODE IS ASCII
```

You can avoid the use of the logical name by giving the command:

```
SET TAPE FORMAT CORE-DUMP
```

This command, along with the declaration of ASCII recording mode in your COBOL program, allows you to read or write an ASCII U-format tape with carriage return/line feed pairs included. You could also read/write an ASCII U-format tape, omitting carriage return/line feeds, by including in your COBOL program the statement:

```
RECORDING MODE IS STANDARD-ASCII
```

Omitting the RECORDING MODE clause from the program would have the same effect.

SIXBIT and Binary

If you want to deal with SIXBIT or binary data on a labeled tape, you must declare the recording mode in your COBOL program. You cannot allow the recording mode to default to SIXBIT; if you did, you would get STANDARD-ASCII data on your tape (assuming the default hardware data mode to be ANSI-ASCII; see the section on EBCDIC and ASCII tapes above). You must also make sure that the default tape format is not INDUSTRY-COMPATIBLE or ANSI-ASCII, so that you are sure to get a U-format tape (INDUSTRY-COMPATIBLE and ANSI-ASCII formats indicate an F-format or D-format tape). Users of TOPS-20 can make sure of this by giving the monitor command shown below:

```
SET TAPE FORMAT CORE-DUMP
```

Users of TOPS-10 cannot use this command; the default tape format is set for the site and cannot be changed by individual users. If you have trouble handling SIXBIT or binary tapes, check with your system administrator to make sure the default tape format is not set to INDUSTRY-COMPATIBLE or ANSI-ASCII.

E.4.2 Tape Has EBCDIC Labels

If your tape has EBCDIC labels, you can read it on both TOPS-10 and TOPS-20 by using the simple forms of the ASSIGN clause and the MOUNT command:

```
SELECT filnam ASSIGN TO tapnam

MOUNT TAPE tapnam /LABEL-TYPE:EBCDIC /xxx
```

You can write EBCDIC-labeled tapes on TOPS-10 using the same ASSIGN clause and MOUNT command. However, TOPS-20 systems cannot write to EBCDIC labeled tapes.

The COBOL object-time system running under TOPS-10 writes F-format tapes if you declare the recording mode to be F in your COBOL program; the same software writes D-format tapes if you declare the recording mode to be V. If the recording mode is something other than EBCDIC F or V, the TOPS-10 COBOL object-time system writes a U-format tape.

TOPS-10 does not read S-format tapes (S-format tapes contain spanned records). F-format, D-format, and U-format tapes can be read on a TOPS-10 system. To read F-format tapes, you must declare the recording mode to be F in your COBOL program. To read D-format tapes, you must declare the recording mode to be V in your COBOL program. If you wish to read a U-format tape, you can read the data in any fashion you wish; no checking is done on the recording mode.

TOPS-20, as mentioned above, does not write EBCDIC-labeled tapes. However, you can read EBCDIC-labeled tapes on TOPS-20. The TOPS-20 monitor processes the data coming from the tape; it knows how big a logical record is, and it hands the COBOL program one logical record. If you are expecting to receive the data in the format in which it was actually recorded, you get the right data when you read a record. If, however, your idea of the data format of the tape is not accurate, your results are unpredictable.

GLOSSARY

The terms in this glossary are defined in accordance with COBOL as used in this document. Therefore, these terms may not have the same meanings in other languages.

These definitions are also intended to serve either as reference material or as introductory material to be reviewed before reading the detailed language specifications. For this reason, these definitions are, in most instances, brief and do not include detailed syntactical rules.

Abbreviated Combined Relation Condition

The combined condition that results from the explicit omission of a common subject, or a common subject and common relational operator, in a consecutive sequence of relation conditions.

Access Mode

The manner in which records are to be operated upon within a file.

Actual Decimal Point

The physical representation (decimal point characters period (.) or comma (,)) of the decimal point position in a data item.

Actual Key

A key whose contents identify a logical record in a random file.

Alphabetic Character

A character that belongs to the following set of letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, and Space.

Alphanumeric Character

Any character in the computer's character set.

Arithmetic Expression

An arithmetic expression can be an identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

Arithmetic Operator

A single character, or a fixed 2-character combination that belongs to the following set:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Ascending Key

A key upon the values of which data is ordered starting with the lowest value of key up to the highest value of key in accordance with the rules for comparing data·items.

Assumed Decimal Point

A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

At End Condition

A condition caused:

1. During the execution of a READ statement for a sequentially accessed file.
2. During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.
3. During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

Block

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained, or to the size of the logical record(s) that are either continued within the block or that overlap the block. The term is synonymous with physical record.

Body Group

Generic name for a report group of TYPE DETAIL, CONTROL HEADING, or CONTROL FOOTING.

Called Program

A program that is the object of a CALL statement combined at object time with the calling program to produce a run unit.

Calling Program

A program that executes a CALL to another program.

Cd-Name

A user-defined word that names an MCS interface area described in a communication description entry within the Communication Section of the Data Division.

Character

The basic indivisible unit of the language.

Character Position

A character position is the amount of physical storage required to store a single standard data format character described as usage is DISPLAY. Further characteristics of the physical storage are defined by the implementor.

Character-String

A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character-string, or a comment-entry.

Class Condition

The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric.

Clause

A clause is an ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

COBOL Character Set

The complete COBOL character set consists of the 51 characters listed below:

Character	Meaning
0,1,...,9	digit
A,B,...,Z	letter
	space (blank)
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	stroke (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

COBOL Word

(See Word.)

Collating Sequence

The sequence in which the characters that are acceptable in a computer are ordered for purposes of sorting, merging, and comparing.

Column

A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined Condition

A condition that is the result of connecting two or more conditions with the 'AND' or the 'OR' logical operator.

Comment-Entry

An entry in the Identification Division that can be any combination of characters from the computer character set.

Comment Line

A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only as documentation in a program. A special form of comment line represented by a stroke (/) in the indicator area of the line, and any characters from the computer's character set in area A and area B of that line, causes page ejection prior to printing the comment.

Communication Description Entry

An entry in the Communication Section of the Data Division that is composed of the level indicator CD, followed by a cd-name, and then followed by a set of clauses as required. It describes the interface between the Message Control System (MCS) and the COBOL program.

Communication Device

A mechanism (hardware or hardware/software) capable of sending data to a queue and/or receiving data from a queue. This mechanism can be a computer or a peripheral device. One or more programs containing communication description entries and residing within the same computer define one or more of these mechanisms.

Communication Section

The section of the Data Division that describes the interface areas between the MCS and the program. This section is composed of one or more CD description entries.

Compile Time

The time at which a COBOL source program is translated by a COBOL compiler to a COBOL object program.

Compiler Directing Statement

A statement beginning with a compiler-directing verb that causes the compiler to take a specific action during compilation.

Complex Condition

A condition in which one or more logical operators act upon one or more conditions. (See Negated Simple Condition, Combined Condition, Negated Combined Condition.)

Computer-Name

A system-name that identifies the computer upon which the program is to be compiled or run.

Condition

A status of a program at execution time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2, ...) appears in these language specifications in or in reference to 'condition' (condition-1, condition-2, ...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

Condition-Name

A user-defined word assigned to a specific value, set of values, or range of values, within the complete set of values that a conditional variable can possess; or the user-defined word assigned to a status of an implementor-defined switch or device.

Condition-Name Condition

The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

Conditional Expression

A simple condition or a complex condition specified in an IF, PERFORM, or SEARCH statement. (See Simple Condition and Complex Condition.)

Conditional Statement

A conditional statement specifies that the truth value of a condition is to be determined, and that the subsequent action of the object program is dependent on this truth value.

Conditional Variable

A data item of which one or more values has a condition-name assigned to it.

Configuration Section

A section of the Environment Division that describes overall specifications of source and object computers.

Connective

A reserved word that is used to:

1. Associate a data-name, paragraph-name, condition-name, or text-name with its qualifier.

2. Link two or more operands written in a series.

3. Form conditions (logical connectives). (See Logical Operator.)

Contiguous Items

Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchical relationship to each other.

Control Break

A change in the value of a data item that is referenced in the CONTROL clause. More generally, a change in the value of a data item that is used to control the hierarchical structure of a report.

Control Break Level

The relative position within a control hierarchy at which the most major control break occurred.

Control Data Item

A data item, in whose contents a change can produce a control break.

Control Data-Name

A data-name that appears in a CONTROL clause and refers to a control data item.

Control Footing

A report group that is presented at the end of the control group of which it is a member.

Control Group

A set of body groups that is presented for a given value of a control data item or of FINAL. Each control group can begin with a CONTROL HEADING, end with a CONTROL FOOTING, and contain DETAIL report groups.

Control Heading

A report group that is presented at the beginning of the control group of which it is a member.

Control Hierarchy

A designated sequence of report subdivisions defined by the positional order of FINAL and the data-names within a CONTROL clause.

Counter

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

Currency Sign

The character '$' of the COBOL character set.

Currency Symbol

The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL source program, the currency symbol is identical to the currency sign.

Current Record

The record that is available in the record area associated with the file.

Current Record Pointer

A conceptual entity that is used in the selection of the next record.

Data Clause

A clause that appears in a data description entry in the Data Division and that provides information describing a particular attribute of a data item.

Data Description Entry

An entry in the Data Division that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

Data Item

A character or a set of contiguous characters (excluding, in either case, literals) defined as a unit of data by the COBOL program.

Data-Name

A user-defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, 'data-name' represents a word that cannot be subscripted, indexed, or qualified unless specifically permitted by the rules for that format.

Declaratives

A set of one or more special-purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES, and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

Declarative-Sentence

A compiler-directing sentence consisting of a single USE statement terminated by the separator period.

Delimiter

A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending Key

A key upon the values of which data is ordered starting with the highest value of key. down to the lowest value of key, in accordance with the rules for comparing data items.

Destination

The symbolic identification of the receiver of a transmission from a queue.

Digit Position

A digit position is the amount of physical storage required to store a single digit. This amount can vary depending on the usage of the data item describing the digit position. Further characteristics of the physical storage are defined by the implementor.

Division

A set of zero, one, or more sections of paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

Division Header

A combination of words followed by a period and a space that indicates that beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION [USING data-name-1 [data-name-2] ...  ] .
```

Editing Character

A single character or a fixed 2-character combination belonging to the following set:

Character	Meaning
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
$	currency sign
,	comma (decimal point)
.	period (decimal point)

Elementary Item

A data item that is described as not being further logically subdivided.

End of Procedure Division

The physical position in a COBOL source program after which no further procedures appear.

Entry

Any descriptive set of consecutive clauses terminated by a period and written in the Identification Division, Environment Division, or Data Division of a COBOL source program.

Environment Clause

A clause that appears as part of an Environment Division entry.

Execution Time

(See Object Time.)

Extend Mode

The state of a file after execution of an OPEN statement with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement for that file.

Figurative Constant

A compiler-generated value referenced through the use of certain reserved words.

File

A collection of records.

File Clause

A clause that appears as part of any of the following Data Division entries:

File description (FD)
Sort-merge file description (SD)
Communication description (CD)

FILE-CONTROL

The name of an Environment Division paragraph in which the data files for a given source program are declared.

File Description Entry

An entry in the File Section of the Data Division that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

File-Name

A user-defined word that names a file described in a file description entry or a sort-merge file description entry within the File Section of the Data Division.

File Organization

The permanent logical file structure established at the time that a file is created.

File Section

The section of the Data Division that contains file description entries and sort-merge file description entries together with their associated record descriptions.

Format

A specific arrangement of a set of data.

Group Item

A named contiguous set of elementary or group items.

High Order End

The leftmost character of a string of characters.

I-O-CONTROL

The name of an Environment Division paragraph in which object program requirements for specific input-output techniques, rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

I-O Mode

The state of a file after execution of an OPEN statement, with the input-output phrase specified, for that file and before the execution of a CLOSE statement for that file.

Identifier

A data-name followed as required by the syntactically correct combination of qualifiers, subscripts, and indexes necessary to make unique reference to a data item.

Imperative Statement

A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement can consist of a sequence of imperative statements.

Implementor-Name

A system-name that refers to a particular feature available on that implementor's computing system.

Index

A computer storage position or register, the contents of which represent the identification of a particular element in a table.

Index Data Item

A data item in which the value associated with an index-name can be stored in a form specified by the implementor.

Index-Name

A user-defined word that names an index associated with a specific table.

Indexed Data-Name

An identifier that is composed of a data-name followed by one or more index-names enclosed in parentheses.

Indexed File

A file with indexed organization.

Indexed Organization

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

Input File

A file that is opened in the input mode.

Input Mode

The state of a file after execution of an OPEN statement with the INPUT phrase specified for that file, and before the execution of a CLOSE statement for that file.

Input-Output File

A file that is opened in the input-output mode.

Input-Output Section

The section of the Environment Division that names the files and the external media required by an object program, and that provides information required for transmission and handling of data during execution of the object program.

Input Procedure

A set of statements that is executed each time a record is released to the sort file.

Integer

A nonnegative numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point. Where the term 'integer' appears in general formats, integer must not be a numeric data item, and must not be signed or zero, unless explicitly allowed by the rules of that format.

Invalid Key Condition

A condition at object time caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

Key

A data item that identifies the location of a record, or a set of data items that serve to identify the ordering of data.

Key Word

A reserved word whose presence is required when the format in which the word appears is used in a source program.

Language-Name

A system-name that specifies a particular programming language.

Level Indicator

Two alphabetic characters that identify a specific type of file or a position in hierarchy.

Level-Number

A user-defined word that indicates the position of a data item in the hierarchical structure of a logical record or that indicates special properties of a data description entry. A level-number is expressed as a 1- or 2-digit number. Level-numbers in the range 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

Library-Name

A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Library Text

A sequence of character-strings and/or separators in a COBOL library.

Line

(See Report Line.)

Line Number

An integer that denotes the vertical position of a report line on a page.

Linkage Section

The section in the Data Division of the called program that describes data items available from the calling program. These data items can be referred to by both the calling and called program.

Literal

A character-string whose value is implied by the ordered set of characters constituting the string.

Logical Operator

One of the reserved words AND, OR, or NOT. In the formation of a condition, both or either of AND and OR can be used as logical connectives. NOT can be used for logical negation.

Logical Record

The most inclusive data item. The level-number for a record is 01. (See Report Writer Logical Record.)

Low Order End

The rightmost character of a string of characters.

Mass Storage

A storage medium on which data can be oganized and maintained in both a sequential and nonsequential manner.

Mass Storage Control System (MSCS)

An input-output control system that directs or controls the processing of mass storage files.

Mass Storage File

A collection of records that is assigned to a mass storage medium.

MCS

(See Message Control System.)

Merge File

A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

Message

Data associated with an end of message indicator or an end of group indicator. (See Message Indicators.)

Message Control System (MCS)

A communication control system that supports the processing of messages.

Message Count

The count of the number of complete messages that exist in the designated queue of messages.

Message Indicators

EGI (end of group indicator), EMI (end of message indicator), and ESI (end of segment indicator) are conceptual indications that notify the MCS that a specific condition exists (end of group, end of message, end of segment).

Within the hierarchy of EGI, EMI, and ESI, an EGI is conceptually equivalent to an ESI, an EMI, and an EGI. An EMI is conceptually equivalent to an ESI and an EMI. Thus, a segment can be terminated by an ESI, an EMI, or an EGI. A message can be terminated by an EMI or an EGI.

Message Segment

Data that forms a logical subdivision of a message normally associated with an end of segment indicator. (See Message Indicators.)

Mnemonic-Name

A user-defined word that is associated in the Environment Division with a specified implementor-name.

MSCS

(See Mass Storage Control System.)

Negated Combined Condition

The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated Simple Condition

The 'NOT' logical operator immediately followed by a simple condition.

Next Executable Sentence

The next sentence to which control is transferred after execution of the current statement is complete.

Next Executable Statement

The next statement to which control is transferred after execution of the current statement is complete.

Next Record

The record that logically follows the current record of a file.

Noncontiguous Items

Elementary data items in the Working-Storage and Linkage Sections that bear no hierarchical relationship to other data items.

Nonnumeric Item

A data item whose description permits its contents to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items can be formed from more restricted character sets.

Nonnumeric Literal

A character-string bounded by quotation marks. The string of characters can include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

Numeric Character

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric Item

A data item whose description restricts its contents to a value represented by characters chosen from the digits '0' through '9'; if signed, the item can also contain a '+', '-', or other representation of an operational sign.

Numeric Literal

A literal composed of one or more numeric characters that also can contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must the leftmost character.

OBJECT-COMPUTER

The name of an Environment Division paragraph in which the computer environment, within which the object program is executed, is described.

Object of Entry

A set of operands and reserved words within a Data Division entry that immediately follows the subject of the entry.

Object Program

A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone can be used in place of the phrase 'object program.'

Object Time

The time at which an object program is executed.

Open Mode

The state of a file after execution of an OPEN statement for that file, and before the execution of a CLOSE statement for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

Operand

Whereas the general definition of operand is 'that component that is operated upon,' for the purposes of this publication any lowercase word (or words) that appears in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

Operational Sign

An algebraic sign associated with a numeric data item or a numeric literal, which indicates whether its value is positive or negative.

Optional Word

A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to you when the format in which the word appears is used in a source program.

Output File

A file that is opened in either the output mode or the extend mode.

Output Mode

The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified for that file, and before the execution of a CLOSE statement for that file.

Output Procedure

A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function has selected the next record in merged order.

Page

A vertical division of a report representing a physical separation of report data, the separation being based on internal reporting requirements and/or external characteristics of the reporting medium.

Page Body

That part of the logical page in which lines can be written and/or spaced.

Page Footing

A report group that is presented at the end of a report page as determined by the Report Writer Control System.

Page Heading

A report group that is presented at the beginning of a report page and determined by the Report Writer Control System.

Paragraph

In the Procedure Division, a paragraph-name followed by a period and a space, and by zero, one, or more sentences. In the Identification and Environment Divisions, a paragraph header followed by zero, 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 and Environment Divisions. The permissible paragraph headers are:

In the Identification Division:

```
PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
REMARKS.
```

In the Environment Division:

```
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.
```

Paragraph-Name

A user-defined word that identifies and begins a paragraph in the Procedure Division.

Phrase

A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical Record

(See Block.)

Prime Record Key

A key whose contents uniquely identify a record within an indexed file.

Printable Group

A report group that contains at least one print line.

Printable Item

A data item, the extent and contents of which are specified by an elementary report entry. This elementary report entry contains a COLUMN NUMBER clause, a PICTURE clause, and a SOURCE, SUM, or VALUE clause.

Procedure

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

Procedure-Name

A user-defined word that is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which can be qualified) or a section-name.

Program-Name

A user-defined word that identifies a COBOL source program.

Pseudo-Text

A sequence of character-strings and/or separators bounded by, but not including, pseudo-text delimiters.

Pseudo-Text Delimiter

Two contiguous equal sign (=) characters used to delimit pseudo-text.

Punctuation Character

A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
.	period
"	quotation mark
(	left parenthesis
)	right parenthesis
	space
=	equal sign

Qualified Data-Name

An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

Qualifier

1. A data-name that is used in a reference together with another data-name at a lower level in the same hierarchy.
2. A section-name that is used in a reference together with a paragraph-name specified in that section.
3. A library-name that is used in a reference together with a text-name associated with that library.

Queue

A logical collection of messages awaiting transmission or processing.

Queue Name

A symbolic name that indicates to the MCS the logical path by which a message or a portion of a completed message can be accessible in a queue.

Random Access

An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

Random File

A file with random organization.

Random Organization

The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical position in the file.

Record

(See Logical Record.)

Record Area

A storage area allocated for processing the record described in a record description entry in the File Section.

Record Description

(See Record Description Entry.)

Record Description Entry

The total set of data description entries associated with a particular record.

Record Key

A key whose contents identify a record within an indexed file.

Record-Name

A user-defined word that names a record described in a record description entry in the Data Division.

Reference Format

A format that provides a standard method for describing COBOL source programs.

Relation

(See Relational Operator.)

Relation Character

A character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

Relation Condition

The proposition for which a truth value can be determined that the value of an arithmetic expression or data item has a specific relationship to the value of another arithmetic expression or data item. (See Relational Operator.)

Relational Operator

A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Relational Operator	Meaning
IS [NOT] GREATER THAN IS [NOT] >	Greater than or not greater than
IS [NOT] LESS THAN IS [NOT] <	Less than or not less than
IS [NOT] EQUAL TO IS [NOT] =	Equal to or not equal to

Report Clause

A clause in the Report Section of the Data Division that appears in a report description entry or a report group description entry.

Report Description Entry

An entry in the Report Section of the Data Division that is composed of the level indicator RD followed by a report name, followed by a set of report clauses, as required.

Report File

An output file whose file description entry contains a REPORT clause. The contents of a report file consist of records that are written under control of the Report Writer Control System.

Report Footing

A report group that is presented only at the end of a report.

Report Group

In the Report Section of the Data Division, an 01 level-number entry and its subordinate entries.

Report Group Description Entry

An entry in the Report Section of the Data Division that is composed of the level-number 01, the optional data-name, a TYPE clause, and an optional set of report clauses.

Report Heading

A report group that is presented only at the beginning of a report.

Report Line

A division of a page representing one row of horizontal character positions. Each character position of a report line is aligned vertically beneath the corresponding character position of the report line above it. Report lines are numbered from 1, by 1, starting at the top of the page.

Report-Name

A user-defined word that names a report described in a report description entry within the Report Section of the Data Division.

Report Section

The section of the Data Division that contains one or more report description entries and their associated report group description entries.

Report Writer Control System (RWCS)

An object-time control system provided by the implementor that constructs reports.

Report Writer Logical Record

A record that consists of the Report Writer print line and associated control information necessary for its selection and vertical positioning.

Reserved Word

A COBOL word specified in the list of words that can be used in COBOL source programs, but that must not appear in the programs as user-defined words or system-names.

Routine-Name

A user-defined word that identifies a procedure written in a language other than COBOL.

Run Unit

A set of one or more object programs that function at object time as a unit to provide problem solutions.

RWCS

(See Report Writer Control System.)

Section

A set of zero, one, or more paragraphs or entries, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section Header

A combination of words followed by a period and a space that indicates the beginning of a section in the Environment, Data, and Procedure Division.

In the Environment and Data Divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers are:

In the Environment Division:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

In the Data Division:

```
FILE SECTION.
SCHEMA SECTION.
WORKING-STORAGE SECTION.
COMMUNICATION SECTION.
LINKAGE SECTION.
REPORT SECTION.
```

In the Procedure Division, a section header is composed of a section-name followed by the reserved word SECTION, followed by a segment-number (optional), followed by a period and a space.

Section-Name

A user-defined word that names a section in the Procedure Division.

Segment-Number

A user-defined word that classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers can contain only the characters '0', '1', .., '9'. A segment-number can be expressed either as a 1- or 2-digit number.

Sentence

A sequence of one or more statements, the last of which is terminated by a period followed by a space.

Separator

A punctuation character used to delimit character-strings.

Sequential Access

An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

Sequential File

A file with sequential organization.

Sequential Organization

The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

Sign Condition

The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

Simple Condition

Any single condition chosen from the set:

relation condition
class condition
condtion-name condition
switch-status condition
sign condition
(simple-condition)

Sort File

A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

Sort-Merge File Description Entry

An entry in the File Section of the Data Division that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses, as required.

Source

The symbolic identification of the originator of a transmission to a queue.

SOURCE-COMPUTER

The name of an Environment Division paragraph in which the computer environment, within which the source program is compiled, is described.

Source Item

An identifier designated by a SOURCE clause that provides the value of a printable item.

Source Program

Although it is recognized that a source program can be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements beginning with an Identification Division and ending with the end of the Procedure Division. In contexts where there is no danger of ambiguity, the word 'program' alone can be used in place of the phrase 'source program.'

Special Character

A character that belongs to the following set:

Character	Meaning
+	plus sign
-	minus sign
*	asterisk
/	stroke (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

Special-Character Word

A reserved word that is an arithmetic operator or a relation character.

SPECIAL-NAMES

The name of an Environment Division paragraph in which implementor-names are related to user-specified mnemonic-names.

Special Registers

Compiler-generated storage areas whose primary use is to store information produced in conjunction with the user of specific COBOL features.

Standard Data Format

The concept used in describing the characteristics of data in a COBOL Data Division under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

Statement

A syntactically valid combination of words and symbols written in the Procedure Division and beginning with a verb.

Sub-Queue

A logical hierarchical division of a queue.

Subject of Entry

An operand or reserved word that appears immediately following the level indicator or the level-number in a Data Division entry.

Subprogram

(See Called Program.)

Subscript

An integer whose value identifies a particular element in a table.

Subscripted Data-Name

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

Sum Counter

A signed numeric data item established by a SUM clause in the Report Section of the Data Division. The sum counter is used by the Report Writer Control System to contain the result of designated summing operations that take place during production of a report.

Switch-Status Condition

The proposition, for which a truth value can be determined, that an implementor-defined switch, capable of being set to an 'on' or 'off' status, has been set to a specific status.

System-Name

A COBOL word that is used to communicate with the operating environment.

Table

A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Table Element

A data item that belongs to the set of repeated items comprising a table.

Terminal

The originator of a transmission to a queue, or the receiver of a transmission from a queue.

Text-Name

A user-defined word that identifies library text.

Text-Word

Any character-string or separator, except space, in a COBOL library or in pseudo-text.

Truth Value

The representation of the result of the evaluation of a condition in terms of one of two values:

true
false

Unary Operator

A plus (+) or a minus (-) sign that precedes a variable or a left parenthesis in an arithmetic expression, and that has the effect of multiplying the expression by +1 or -1, respectively.

Unit

A module of mass storage the dimensions of which are determined by each implementor.

User-Defined Word

A COBOL word that must be supplied by you to satisfy the format of a clause or statement.

Variable

A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Verb

A word that expresses an action to be taken by a COBOL compiler or object program.

Word

A character-string of not more than 30 characters that forms a user-defined word, a system-name, or a reserved word.

Working-Storage Section

The section of the Data Division that describes working storage data items, which is composed either of noncontiguous items or of working storage records or of both.

77-Level-Description-Entry

A data description entry that describes a noncontiguous data item with the level-number 77.

INDEX

TOPS–10/TOPS–20
COBOL–68
Language Manual
AA–5057B–TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____________

Name ____________ Date ____________

Organization ____________ Telephone ____________

Street ____________

City ____________ State ____________ Zip Code or Country ____________

-------Do Not Tear - Fold Here and Tape---

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1-2/E37
MARLBOROUGH, MASSACHUSETTS 01752

------ Do Not Tear - Fold Here and Tape---

Cut Along Dotted Line