COBOL-74 Language Manual

Order Number AA-5059A-TK

January 1979

This manual reflects the software of version 12 of the COBOL-74 compiler (CBL74), version 12 of the object-time system (C74OTS), and version 4A of SORT.

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, January 1979

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

FIGURES

CONTENTS (CONT.)

PREFACE

This manual describes COBOL-74 as it has been implemented on the TOPS-10 and TOPS-20 operating systems. Part 1 of this manual outlines the topics to be covered in each chapter. Part 2 describes the COBOL-74 compiler and presents the vocabulary and syntax of the language. Part 3 provides the information necessary to use the COBOL system, including performance improvement, utilities, and various features of COBOL-74. Part 4 contains appended material. Several appendixes and a glossary of COBOL-74 terms have also been included. Appendix A is a list of differences between COBOL-74 and COBOL-68. Appendix B contains a list of all COBOL-74 reserved words. Appendix C lists the character collating sequences. Appendix D describes an alternative form of numeric test which may be elected at system installation time.

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

> Carl Feingold, Fundamentals of COBOL Programming (Dubuque, Iowa, William C. Brown Company, 1977).
>
> Daniel D. McCracken, A Simplified Guide to Structured COBOL Programming (New York, John Wiley and Sons, Inc., 1976).
>
> Daniel D. McCracken and Umberto Garbassi, A Guide to COBOL Programming, Second Edition (New York, John Wiley and Sons, Inc., 1970).

The COBOL programmer should be familiar with the operating system commands and the editing language for the system in question. For users of the TOPS-10 operating system, the manuals which contain this information are:

- Operating System Commands Manual
- TECO Programmer's Reference Manual

Other manuals which contain information useful to TOPS-10 COBOL-74 programmers are:

- Monitor Calls Manual
- Hardware Reference Manual
- LINK Reference Manual

For users of TOPS-20, the information concerning the operating system commands and the system editing language is contained in the following manuals:

- DECSYSTEM-20 User's Guide
- EDIT Reference Manual

Other manuals which contain information useful to TOPS-20 COBOL-74 programmers are:

- Monitor Calls Reference Manual
- Hardware Reference Manual
- LINK Reference Manual

ACKNOWLEDGMENT

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

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

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959 by Sperry Rand Corporation;

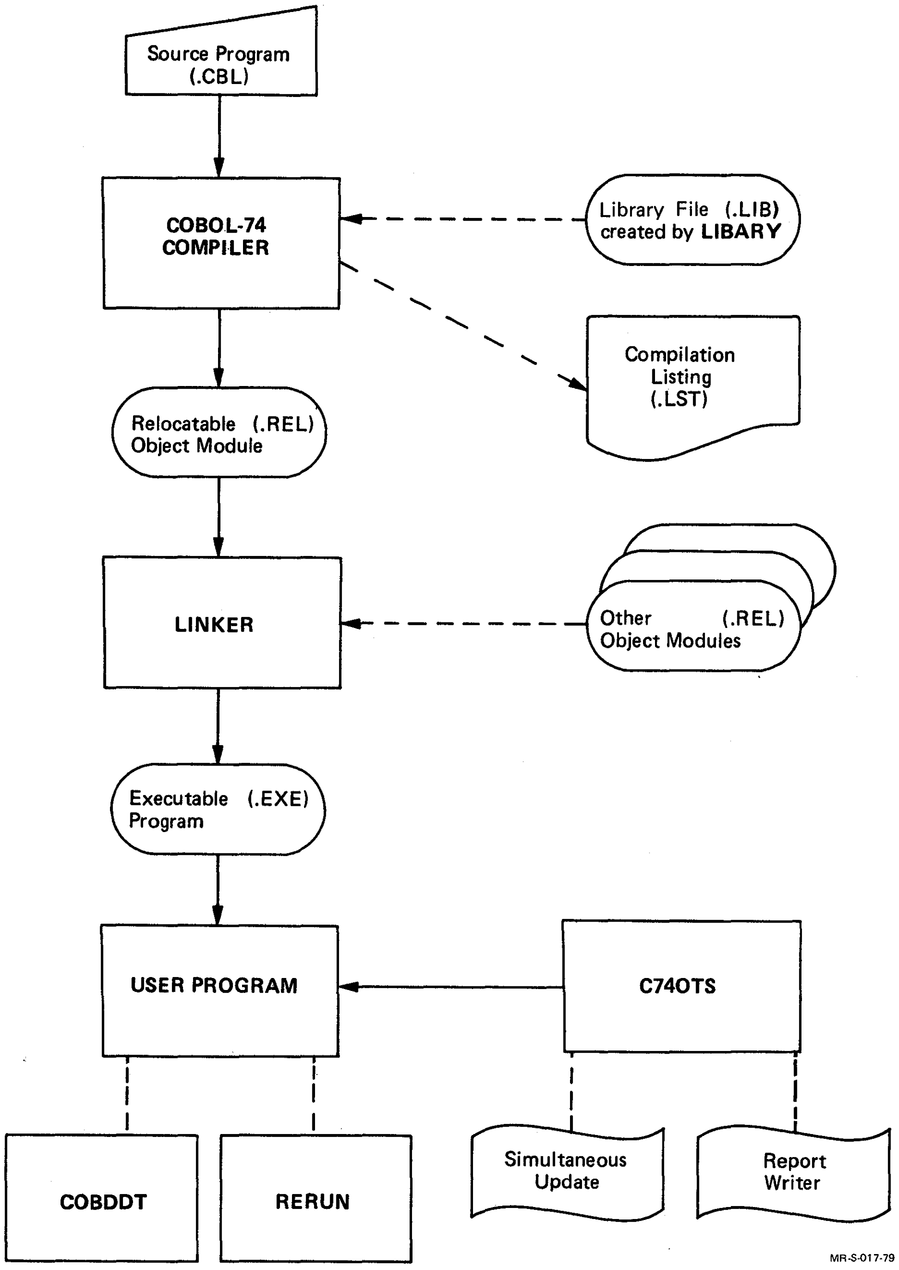
IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM;

FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

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

INTRODUCTION TO THE COBOL-74 SYSTEM AND THE STRUCTURE OF THE MANUAL

The typical COBOL program follows a fairly simple series of steps from the human-readable format in which it is written to the machine-readable format in which it is executed. The following flow chart shows the basic steps which all programs take.


Source Program (.CBL)
COBOL-74 COMPILER
Library File (.LIB) created by LIBARY
Compilation Listing (.LST)
Relocatable (.REL) Object Module
LINKER
Other (.REL) Object Modules
Executable (.EXE) Program
USER PROGRAM
C74OTS
COBDDT
RERUN
Simultaneous Update
Report Writer
MR-S-017-79

The program first sees the light of day as a source file which is either created with a text editor or entered into the system by some other means (for example, it could be punched into cards and loaded through a card reader). This file is usually given a filename whose extension is .CBL, and it is identified in the flow chart by this extension.

The COBOL-74 compiler then translates the source file into a relocatable object module. In order to do this, the compiler may sometimes copy text from user libraries which contain often-used pieces of code. These libraries, identified in the chart by the extension of .LIB, are created by the LIBARY utility. The output from the compiler, the relocatable object module, is usually given an extension of .REL, and is identified by this extension in the flow chart. The compiler can optionally produce a file which contains the compilation listing of the source program. This file is identified by its extension, .LST.

At this point the program is given to the system linker, which produces the executable version with the extension .EXE. (This manual does not contain any information on the system linker. Users of TOPS-10 should refer to the LINK Reference Manual and the LOAD command in the Operating System Commands Manual for more information about LINK. Users of TOPS-20 should refer to the the LINK Reference Manual and the LOAD command in the DECSYSTEM-20 user's Guide.)

The .EXE version of the program runs in conjunction with the object-time system, C74OTS. Among other things, the object-time system handles I/O and calls routines from the COBOL-74 library to be used at runtime. The user program is now in a format which can be executed, but there is no guarantee that it will produce the correct results. Most programs must still be debugged after they compile error-free. The COBOL-74 system provides an on-line debugging facility called COBDDT to assist the programmmer in finding out what the program is really doing. COBDDT runs along with the user program and the object-time system, and allows the steps which the program executes to be monitored by the programmer.

Many COBOL programs use indexed files during their execution. These files are convenient for many applications. The COBOL-74 system provides a program, called ISAM, to create and maintain indexed files.

There are times when the user program is running and the system operator has to shut down the system unexpectedly. Some programs are written to be restartable, but many are not. The RERUN utility is provided with COBOL-74 to help in this situation. RERUN can save enough information to allow the program to be restarted after the system is brought back up, even though no provision was made in the program for the restart.

Thus, the COBOL-74 system, in conjunction with the operating system, provides complete facilities for the creation and execution of a COBOL program. The rules regarding the creation of a COBOL-74 program, and the syntax to be used in the program, are described in Part 2, COBOL-74 Language Reference Material. The individual units of the COBOL-74 system are enumerated below.

1. The Compiler -

   The compiler copies text from user libraries and translates the COBOL-74 program into a relocatable object module. Running the COBOL-74 compiler is described in Part 3, Chapter 6.

2. The OTS -

   The object-time system runs the COBOL-74 program and allows the program to use such facilities as simultaneous update and Report Writer. Information on the file formats which the OTS accepts may be found in Part 3, Chapter 8. The simultaneous update facility is described in Part 3, Chapter 9, and Report Writer in Part 3, Chapter 10. Subprograms, segmentation and overlaying are covered in Part 3, Chapter 11. Chapter 12 of Part 3 contains information on calling non-COBOL subprograms.

3. The Utilities -

   The COBOL-74 utilities - LIBARY, COBDDT, RERUN and ISAM - are described in Part 3, Chapter 7. Information on the use of COBDDT in improving the performance of COBOL-74 programs may be found in Part 3, Chapter 13.

Part 4 of this manual contains appended material which may be of interest to some users of COBOL-74. Appendix A presents a list of differences between DIGITAL's COBOL-68 and DIGITAL's COBOL-74. Appendix B is the list of COBOL-74 reserved words. Appendix C provides ASCII, SIXBIT, and EBCDIC collating sequences, along with conversion charts for these three codes. An alternate to the usual numeric test, which may be elected at the time of installation of COBOL-74, is described in Appendix D. Finally, Appendix E contains a short description of the process of defining a logical name for TOPS-20 users of the COBOL-74 utilities.

CHAPTER 1

INTRODUCTION TO COBOL-74 LANGUAGE

This chapter describes the symbols, special terms, language elements, and source program formats acceptable to COBOL-74. The source language statements are discussed in subsequent chapters.

NOTE

In this manual the word COBOL refers to COBOL-74. Any documentation concerning DECtapes can be ignored if your system does not have them.

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 to this statement is the term BIS-compiler. This term refers to compiler implementations that compile COBOL-74 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 may 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-74 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. Its basis is the system of symbols used in the American National Standard and developed by CODASYL.

1.1.1.1 **Underline** - The underline is used to denote reserved key words. Key words (uppercase underlined words) are required when you use a function of which they are a part. The absence of an underline in an uppercase word denotes that the word is optional; you may use or omit the word at your discretion.

NOTE

Uppercase words, whether underlined or not, must be spelled correctly.

1.1.1.2 **Brackets and Braces** - When brackets, [], enclose a portion of a general format, they denote an optional portion that may be included or omitted as needed. When braces, {}, enclose a portion of a general format, you must select one of the options within the braces. Consider the following figure.

$$\left[\text{MEMORY SIZE integer} \begin{Bmatrix} \text{WORDS} \\ \text{CHARACTERS} \\ \text{MODULES} \end{Bmatrix} \right]$$

The brackets indicate that the entire clause is optional. The braces indicate that a choice of one of the words vertically stacked within the braces must be specified.

Wherever a choice is required, the possibilities are vertically stacked either within brackets or braces. Consider the following example.

$$\left[\begin{Bmatrix} \text{SYNCHRONIZED} \\ \text{SYNC} \end{Bmatrix} \begin{bmatrix} \text{LEFT} \\ \text{RIGHT} \end{bmatrix} \right]$$

The outside brackets indicate that the entire clause is optional. The braces indicate that if the clause is used, a choice of a word vertically stacked within the braces must be made. The inside brackets indicate that you may optionally select a vertically stacked word within.

NOTE

When possibilities are vertically stacked between brackets, you have the option of overriding a default condition. The default condition is described in the general rules for the clause.

1.1.1.3 **The Ellipsis** - The ellipsis (...) indicates that you may repeat the item preceding it. The preceding item is usually enclosed either by brackets or braces to remove any ambiguity as to which item may be repeated. Consider the following example.

```
[SAME [RECORD] AREA FOR file-name-1 [file-name-2] ...] ...
```

The final ellipsis indicates that the entire clause, if used, may be repeated. The initial ellipsis indicates that the item file-name-2 may also be repeated within the clause.

1.1.2 COBOL Terms

The terms block, record, and item have special meanings when used in relation to 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.

NOTE

The term "block" as defined here does not refer to a "disk block", which is 128 words of storage space on a disk.

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. Each division is made up of source language statements. Some statements are required in every program; most of them 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

The COBOL-74 compiler will recognize source line numbers up to and including 8184. If your program (including library routines) exceeds this maximum, the compiler will start numbering again at 0001. Since this causes two or more lines to have a single line number, you should exercise caution when debugging your program. The cross-reference listing may be confusing. However, the compiler will generate correct code regardless of how many lines are in the program or how they are numbered in the cross-reference listing.

1.2.2 COBOL-74 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. CTRL/Z (32(8)), which marks the end-of-file

The compiler translates the lowercase ASCII characters to uppercase 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 user-defined words, such as data-names, procedure-names, and identifiers.

The remaining ASCII characters which are acceptable to the COBOL-74 compiler are listed below.

Punctuation characters include:

Δ	(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)
/	(slash)	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)

NOTE

These special characters will not necessarily be underlined when they appear in formats. For example, an underlined minus sign might easily be confused with an equal sign. However, they are usually required items. You may not omit them, unless you are specifically told otherwise.

1.2.3 Words

A COBOL word is a character string which has not more than 30 characters and is either a user-defined word or a reserved word. For COBOL-74, as for most COBOL compilers, a word may be either user-defined or reserved, but not both.

1.2.3.1 **Reserved Words** - A reserved word is a COBOL word that is one of a specific list that may be used in COBOL source programs as specified in the general formats. You cannot use a reserved word as a user-defined word; the two types are mutually exclusive. (See Appendix B for a complete list of COBOL reserved words).

There are six types of reserved words:

1. Key words

   A key word is required when the format in which the word appears is used in a source program. Within each format, key words are uppercase and underlined. Consider the following example.

   <u>COMPUTE</u> identifier-1 [<u>ROUNDED</u>] [identifier-2 [<u>ROUNDED</u>]] ...

   =arithmetic-expression [ON <u>SIZE</u> <u>ERROR</u> imperative-statement]

   In this case, the words COMPUTE, ROUNDED, SIZE, and ERROR are key words.

2. Optional Words

   Within each format, uppercase words that are not underlined are optional words included for readability. You may use or omit these words indiscriminately. The presence or absence of an optional word does not alter the semantics of the COBOL program in which it appears. Consider the following example.

   <u>LINAGE</u> IS integer-1 LINES [WITH <u>FOOTING</u> AT integer-2]

   [LINES AT <u>TOP</u> integer-3]

   In this case, the words IS, LINES, WITH, and AT are optional words.

3. Connectives

   There are three types of connectives:

   a. Qualifier connectives that associate a data-name, a condition-name, or a text-name with its qualifiers: OF, IN (See Section 4.7, Qualification.) An example of this type is

      ```
      COPY ACTREC OF COBLIB.
      ```

   b. Series connectives that link two or more consecutive operands: separator comma, separator semicolon. An example is

      ```
      GO TO PART1, PART2, PART3 DEPENDING ON COUNTER1.
      ```

   c. Logical connectives that are used in the formation of the following conditions: AND, OR, AND NOT, OR NOT. An example is

      ```
      IF HOURS-WORKED IS GREATER THAN ZERO AND NOT
      DEDUCTION-TIME PERFORM PRINT-CHECK.
      ```

4. Figurative Constants

   A few specific constant values are used frequently and in enough different ways to make it useful to have names for them. The names given to them are called Figurative Constants. These names are reserved words and are listed below.

The values represented by figurative constants are generated by the compiler and referenced through the use of the reserved words given below. These words must not be bounded by quotation marks when used as figurative constants. The singular and plural forms of figurative constants are equivalent and may be used interchangeably to increase readability.

The values which the compiler generates for you, and the reserved words that name them, are as follows:

ZERO ZEROS ZEROES	Represent the value '0', or one or more of the character '0', depending on context.
SPACE SPACES	Represent one or more of the character "space".
HIGH-VALUE HIGH-VALUES	Represent one or more of the character that has the highest ordinal position in the computer's collating sequence (in ASCII code, this is octal 177).
LOW-VALUE LOW-VALUES	Represent one or more of the character that has the lowest ordinal position in the computer's collating sequence (in ASCII this is octal 000).
QUOTE QUOTES	Represent one or more occurences of the quote character, usually '"' (double quote).
ALL literal	Represents one or more repetitions of the string of characters which compose the literal. The literal must be either an alphanumeric literal or a figurative constant other than ALL. When a figurative constant is used, the word ALL is redundant and is an option. You may use it for readability if you wish.

Frequently a figurative constant represents a string of characters whose length is not explicitly stated. When this happens, the compiler determines the length of the string from context. The figurative constant may be associated with another data item by the context, as in the following statements:

```
MOVE SPACES TO WORK-RECORD

IF AMOUNT-OWED EQUALS ZERO PERFORM CLOSE-ACCOUNT
```

Alternatively, the figurative constant may stand by itself with no relation to any data item, as in:

```
DISPLAY "BALANCE IS" ZERO

STRING DAY-CODE, SPACE, "-", SPACE, MONTH-CODE
  DELIMITED BY SIZE INTO DSPLY-DATE
```

In cases where the figurative constant is associated with a data item, the compiler assumes that the string of characters represented by the figurative constant has the same number of characters as the associated data-item. In the case of the figurative constant ALL literal, the literal is repeated from left to right and truncated on the right, if necessary.

Thus, if WORK-RECORD in the above example contains 128 characters, the figurative constant SPACES represents a string of 128 spaces. If AMOUNT-OWED is an eight-character numeric field with two decimal places, ZERO represents the value 000000.00. In the following example:

```
MOVE ALL "ABC" TO HOLD-AREA
```

if HOLD-AREA is a ten-character alphanumeric field, its contents after the MOVE will be

A	B	C	A	B	C	A	B	C	A

If you associate a JUSTIFIED clause with the data item, the character repetition and truncation will take place before any justification.

When the figurative constant is not associated with a data item, as in the second set of examples above, the length of the character string is one character, or one occurrence of the literal in the case of ALL literal. This is true even if you use the plural form instead of the singular. That is, all of the following statements cause the same display:

```
DISPLAY ZERO.
DISPLAY ZEROS.
DISPLAY ALL ZEROS.
```

In each case, one zero will be displayed.

A figurative constant may be used whenever a literal appears in a format. However, if the literal is restricted to numeric characters, the only figurative constants permitted are ZERO (ZEROS, ZEROES), LOW-VALUE (LOW-VALUES), and HIGH-VALUE (HIGH-VALUES).

Each reserved word that is used to reference a figurative constant value is a distinct character string with the exception of the construction ALL literal, which is composed of two distinct character strings.

5. Special Registers

COBOL-74 recognizes four reserved words as special registers: DAY, DATE, TIME, and LINAGE-COUNTER. All special registers have implied data descriptions of unsigned elementary integers. The lengths of DAY, DATE, and TIME are fixed; the length of LINAGE-COUNTER depends upon the file description statement that generates the register.

DAY is five digits long. Its value represents the number of the current day of the year. Its format is:

```
YYDDD
```

where YY is the year of the century, and

DDD is the number of the day of the year.

DATE is six digits long. Its value represents the current date. Its format is:

YYMMDD

where YY is the year of the century,

MM is the number of the month, and

DD is the number of the day.

TIME is eight digits long. Its value represents the current elapsed time since midnight on a twenty-four-hour basis. Its format is:

HHMMSShh

where HH is the hours,

MM is the minutes,

SS is the seconds, and

hh is the 1/100ths of a second.

DAY, DATE, and TIME may be accessed by ACCEPT statements in the Procedure Division. See Section 5.9.1 for the correct format to use with the ACCEPT verb.

The LINAGE-COUNTER special register is generated whenever the file description of a sequential file includes the LINAGE clause. The contents of a LINAGE-COUNTER represent the current line number within the current page of output. The contents of a LINAGE-COUNTER are updated automatically by WRITE statements referring to the associated sequential file. The LINAGE clause and LINAGE-COUNTER are fully explained in Section 4.9.31.

6. Special-Character Words

   The arithmetic operators +, -, *, /, **, ^, and the relation characters <,>, and = are special-character reserved words.

1.2.3.2 **User-Defined Words** - A user-defined word is a COBOL word which is supplied by the user to satisfy the format of a clause or statement. The characters which may be used to form user-defined words are the letters of the alphabet, the digits 0 through 9, and the hyphen. The hyphen may not be used as the first or last character in the user-defined word.

There are 17 types of user-defined words:

1. alphabet-name
2. cd-name
3. condition-name
4. data-name

5. file-name
6. index-name
7. level-number
8. library-name
9. mnemonic-name
10. paragraph-name
11. program-name
12. record-name
13. report-name
14. routine-name
15. section-name
16. segment-number
17. text-name

Each of these user-defined word types is described in the Glossary which appears at the end of this manual.

1.2.4 Literals

A literal is a character string whose value is determined by the ordered set of characters of which it is composed. You can also use a figurative constant as a literal. There are two types of literals: numeric and alphanumeric.

1.2.4.1 **Numeric Literal** - A numeric literal is a character string of from 1 to 20 characters selected from the digits 0 through 9, the plus sign, the minus sign, and the decimal point. The rules for the formation of numeric literals are as follows:

1. A literal must contain at least 1 digit and no more than 18 digits.
2. A literal must not contain more than one sign character. If a sign is used, it must appear as the leftmost character of the literal. If the literal is unsigned, it is considered positive.
3. A literal must not contain more than one decimal point. The decimal point is treated as an assumed decimal point, and may appear anywhere within the literal except as the rightmost character. If the literal contains no decimal point, the literal is considered an integer.

NOTE

The word integer, appearing in a general format, represents a nonnegative numeric literal with no decimal point.

If a literal conforms to the rules for the formation of numeric literals but is enclosed in quotation marks, it is considered an alphanumeric literal and is treated as such by the compiler.

4. The value of a numeric literal is the algebraic quantity represented by the characters in the numeric literal. Every numeric literal is category numeric. (See Section 4.10.16, The PICTURE Clause.) The size of a numeric literal is equal to the number of digits specified by the user, including leading zeros, if any.

1.2.4.2 **Alphanumeric Literals** - An alphanumeric literal is a character string representing from 1 to 120 characters, delimited on both ends by quotation marks and consisting of any allowable character in the computer's character set. An opening quotation mark must be immediately preceded by a space or left parenthesis. A closing quotation mark must be immediately followed by one of the separators (space, comma, semicolon, or right parenthesis) or by the terminator, period.

NOTE

You may use either the single quote character (') or the double quote ("). Whichever one you use, you must be sure to pair them correctly - do not try to pair a single quote with a double quote or vice versa.

To represent one quotation-mark character within an alphanumeric literal, two contiguous quotation marks must be used. The value of an alphanumeric literal in the object program is the string of characters itself, except that:

1. The delimiting quotation marks are excluded, and

2. Each embedded pair of contiguous quotation marks represents a single quotation mark character.

All other punctuation characters are part of the value of the alphanumeric literal, not separators. All alphanumeric literals are category alphanumeric. (See Section 4.9.18, The PICTURE Clause.)

1.2.5 Separators

A separator is a string of one or more punctuation characters. The rules for forming separators are:

1. Space

   a. Anywhere a space is used as a separator, more than one space may be used.

   b. A space may immediately precede all separators except the closing quotation mark. Here the space is considered part of an alphanumeric literal, not a separator.

c. A space may immediately follow any separator except the open quotation mark. In this case, a following space is considered part of an alphanumeric literal, not a separator.

2. Comma and Semicolon

   The punctuation characters, the comma and semicolon, are separators. You may insert these separators only where explicitly permitted by the general formats, by format punctuation rules, by statement and sentence definitions, or by source program format rules.

3. Right Parenthesis and Left Parenthesis

   Right parenthesis and left parenthesis are separators only when used in balanced pairs to delimit subscripts or indexes.

4. Quotation Marks

   Quotation marks may be used only in balanced pairs to delimit alphanumeric literals or in adjacent pairs to pass one quotation mark in an alphanumeric literal. (See note concerning quotation marks in Section 1.2.4.2, Alphanumeric Literals.)

5. Horizontal Tab

   The horizontal tab character is governed by the same rules that govern the space character. It is normally used to vertically align statements or clauses on successive lines of the source program listing. The compiler, upon encountering a tab character, generates one or more space characters consistent with the tab character position in the source line.

6. Pseudo-text Delimiter

   Pseudo-text delimiters set off textual matter in the COPY statement from the rest of the sentence. Each delimiter consists of two contiguous equal signs (==). The opening pseudo-text delimiter must be immediately preceded by a space; the closing delimiter must be immediately followed by one of the separators space, comma, semicolon, or period. These delimiters may appear only in balanced pairs delimiting pseudo-text.

NOTE

There are certain rules for writing source programs which supersede these general rules. For a discussion of source program formats see Section 1.3.

1.3 SOURCE PROGRAM FORMAT

There are two basic types of source program formats in which you may write your COBOL-74 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-74 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 which are to be entered into the system through a terminal using a text editor. The compiler will assume 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 Appendix C).

Certain margins which begin the areas used for writing COBOL-74 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 1974 ANSI standard for source program formats. Programs written according to the rules will be more readable and transportable. The COBOL-74 compiler, however, does not do complete syntax checking to determine if you have followed all rules, and will 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 may 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 will assume 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 may simply write the first line, then continue on the next line; do not use the continuation column for this purpose.

2. Comment Lines - You may insert comment lines into your COBOL-74 program by using the continuation area. If the compiler finds an asterisk (*) in that area it will list the remainder of the line as a comment on the next line. If there is a slash (/) instead of an asterisk a new page will be started and the comment will be listed at the top of the new page.

NOTE

All formats may 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 may 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 the user. The layout of a line in this format is shown in Figure 1-1. The numbers refer to card columns or character positions.

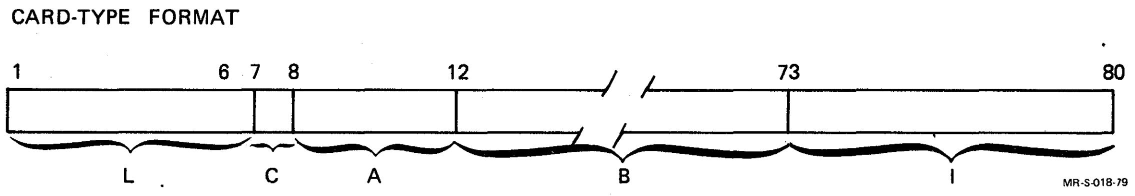


Figure 1-1(a) Card-type Format

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 the user who creates the file on a terminal or a card punch.

2. Debug Lines - You may insert debug lines into your program by putting a "D" in the continuation area (column 7). The compiler will recognize it and print it on the source listing with the spacing similar to a comment line.

3. Identification Area - This area is marked I in the figure (positions 73 through 80). These eight character positions may hold identifying information which can be composed of any eight characters. This information will be 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-74 when you specify card-type format.

The examples in Figure 1-1(b) illustrate these rules. The first two lines are simple statements, with a line number in area L, COBOL-74 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 may 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).

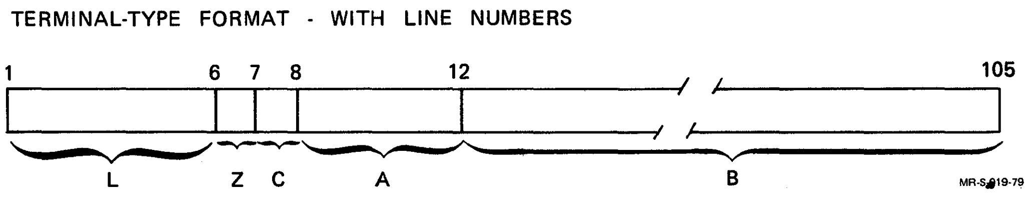


Figure 1-2(a) Terminal-type Format with Line Numbers

In this format, margin L is to the left of position 1 and margin R is to the right of position 105. 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.

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 the user. 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 may insert a tab here for purposes of making your text more readable; if so, the compiler will read 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 may 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 will assume that you meant the space to be part of the line.

4. Debug Lines - Debug lines can be inserted in your program with this format if you type "\D" (backslash D) as the first two characters on the line. If you use "D" as in card-type format, the compiler will read the "D" as the first character of a word beginning in area A.

The examples in figure 1-2(b) illustrate the use of this format. The first two lines are simple COBOL-74 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 may 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) 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).

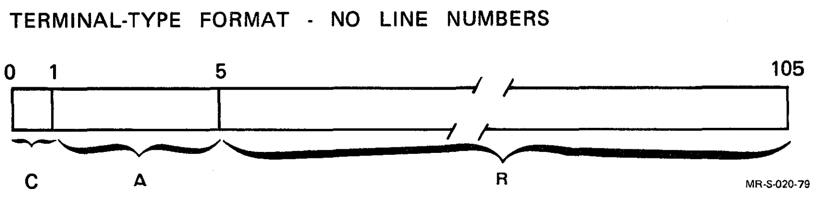


Figure 1-3(a) Terminal-type Format without Line Numbers

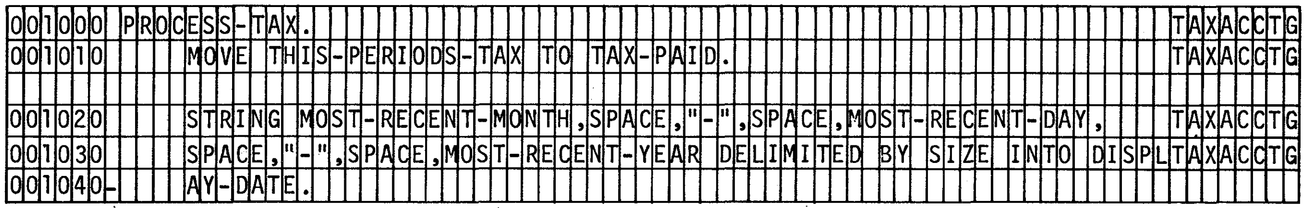
In this format, margin L is to the left of position 0, if it exists, or position 1, if position 0 does not exist. Margin R is to the right of position 105. Margin A is to the left of position 1 and begins the area labeled A. Margin B is between positions 4 and 5 and begins the area labeled B.

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

1. 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 will assume 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.

2. Debug Lines - Debug lines may be inserted into the program. To do this type a "\D" (backslash D) as the first two characters on the line.

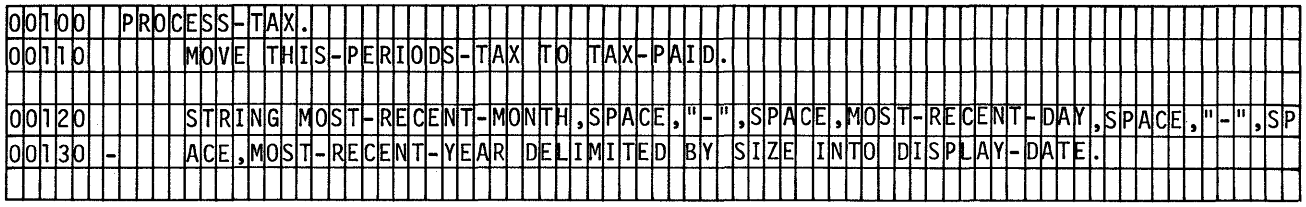
The examples in Figure 1-3(b) show this format's simplicity. The first two lines are the same simple COBOL-74 sentences as above. Note that the paragraph-name starts in the very first character position. The third line shows how to tell the compiler that the line you enter is a continuation (or a comment) 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.



```
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.
```

MR-S-021-79

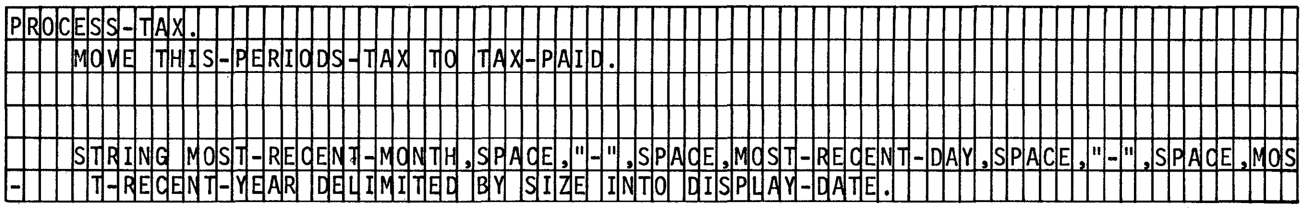
Figure 1-1 (b)



```
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.
```

MR-S-022-79

Figure 1-2 (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.
```

MR-S-023-79

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-74 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.

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-74 Usage Material, Part 3 of this manual.) The COPY statement may also be used to replace specified text in the source text being copied.

General Format

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

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

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 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, will be removed when the text is copied from the library.

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

4. String-2 may be null.

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

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

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. Debugging lines are permitted within library text and string-2. Debugging lines are not permitted within string-1; text-words within a debugging line participate in the matching rules as if the 'D' did not appear in the indicator area. If a COPY statement is specified on a debugging line, then the text that is the result of the processing of the COPY statement will also appear as though it were specified on debugging lines with the following exception: comment lines in library text will appear as comment lines in the resultant source program.

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

14. 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.

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

16. 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. It identifies the source program and the output from compilation. In addition, it may contain 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.]

[AUTHOR. comment-entry ...]

[INSTALLATION. comment-entry ...]

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

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

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

Technical Notes

1. The Identification Division must begin with the reserved words IDENTIFICATION DIVISION followed by a period and a space. Note that in COBOL-74 the reserved word ID may be substituted for IDENTIFICATION in the division header.

2. The PROGRAM-ID paragraph contains the name identifying the program. The program-name may 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, may 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 which will be replaced by the current date. Reserved words can be used in any comment paragraph.

THE IDENTIFICATION DIVISION

GENERAL FORMAT FOR IDENTIFICATION DIVISION

```
{ ID             }
{ IDENTIFICATION } DIVISION.

[PROGRAM-ID.  program-name.]

[AUTHOR.   comment-entry   ...]

[INSTALLATION.   comment-entry   ...]

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

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

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

CHAPTER 3

THE ENVIRONMENT DIVISION

The Environment Division allows you to describe the particular computer configurations you wish to use for program compilation and execution. In this division you also specify the files and devices you will use for input and output. The clauses used to do these things are presented on the following pages.

CONFIGURATION SECTION

3.1 ENVIRONMENT DIVISION CLAUSE FORMATS

3.1.1 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 3.1.2)

OBJECT-COMPUTER. (See Section 3.1.3)

SPECIAL-NAMES. (See Section 3.1.4)

Technical Notes

1. This section is optional.
2. All commas and semicolons are optional. A period must terminate the entire entry.

3.1.2 SOURCE-COMPUTER

Function

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

General Format

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

Technical Notes

1. This paragraph is optional.

2. Computer-name must be one of the list DECsystem-10, DECSYSTEM-20, PDP-10, or PDP-integer-1. Integer-1 must be in the range 1000 to 1099.

3. If the WITH DEBUGGING MODE clause is specified, all debugging lines are compiled. If it is not specified all debugging lines are treated as if they were comment lines. In either case all USE FOR DEBUGGING statements are compiled as if they were comments. This is because COBDDT accomplishes what is otherwise done with debugging statements.

Examples

```
SOURCE-COMPUTER. DECSYSTEM-1055.

SOURCE-COMPUTER. DECSYSTEM-20 WITH DEBUGGING MODE.
```

OBJECT-COMPUTER

3.1.3 OBJECT-COMPUTER

Function

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

General Format

```
OBJECT-COMPUTER.   computer-name

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

   [PROGRAM COLLATING SEQUENCE IS alphabet-name]

   [SEGMENT-LIMIT IS segment-number]
```

Technical Notes

1. This paragraph is optional.

2. Computer-name must be one of the following: PDP-10, PDP-integer-1, DECsystem-10, or DECSYSTEM-20. Integer-1 must be a number in the range 1000 through 1099. The number specified is for documentary purposes only and has no direct bearing on the object code generated by the compiler. If the compiler was installed to take advantage of the KL central processing unit's Business Instruction Set (BIS), the BIS-code will be generated automatically. (See the COBOL-74 Installation Procedures.)

3. The optional MEMORY SIZE clause specifies the maximum memory size of SORT's work area during a SORT operation. If the MEMORY SIZE clause 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)

COBOL-74 presently ignores the MEMORY SIZE clause. SORT will use its default algorithms to determine the amount of memory needed to execute a sort. (Refer to the SORT User's Guide for more information.)

4. The PROGRAM COLLATING SEQUENCE clause specifies a collating sequence for a program. When you use the PROGRAM COLLATING SEQUENCE clause the collating sequence is the one associated with alphabet-name. When you do not use the PROGRAM COLLATING SEQUENCE clause the collating sequence is ASCII. The program collating sequence determines:

   1. the results of explicit comparisons in relation-conditions and in condition-name conditions

   2. the results of implicit comparisons in CONTROL clauses of report description entries

   3. the order of records processed by SORT and MERGE statements which do not specify another collating sequence with the COLLATING SEQUENCE phrase

   4. the values of the figurative constants HIGH-VALUE and LOW-VALUE

   (See the alphabet-name IS clause in the SPECIAL-NAMES paragraph for information on how to associate a collating sequence with alphabet-name.)

5. If you use the SEGMENT-LIMIT clause, only those segments having segment numbers from 0 up to but not including the value of integer-3 are treated as resident segments of the program. Integer-3 must be a positive integer in the range 1 to 49.

   If you omit the SEGMENT-LIMIT clause, 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 Sections 5.3 and 11.1.

6. The DISPLAY clause is optional. If you include it in your program, the compiler will use the DISPLAY type you specify as the default in determining the recording mode for external files and for items described in the Data Division as DISPLAY. This allows you to change the default usage inside the program without using compiler switches. The effect of specifying DISPLAY IS DISPLAY-9 is the same as that of including a /X switch in the command string to the compiler. However, the /X switch will always override the DISPLAY clause. For example, if you include in your program the following statement

   ```
   DISPLAY IS DISPLAY-7
   ```

   all items described in the Data Division as USAGE IS DISPLAY will be considered DISPLAY-7 items.

Example

```
OBJECT-COMPUTER. DECSYSTEM-1077
     MEMORY 50000 WORDS
     SEGMENT-LIMIT IS 35
     PROGRAM COLLATING SEQUENCE IS NATIVE
     DISPLAY IS DISPLAY-7.
```

SPECIAL-NAMES

3.1.4 SPECIAL-NAMES

Function

The SPECIAL-NAMES paragraph provides a means of assigning mnemonic names to input/output devices, code sets, and collating sequences. This paragraph may also define the character used as a currency sign, and may specify the interchange of decimal point and comma functions in the program.

General Format

```
[SPECIAL-NAMES.

┌                  ┌                                                                        ┐ ┐
│                  │ STANDARD-1                                                              │ │
│                  │ NATIVE                                                                  │ │
│                  │                                                                         │ │
│                  │             ┌ {THROUGH} literal-2                                 ┐     │ │
│                  │ literal-1   │ {THRU   }                                           │     │ │
│ alphabet-name IS<              └ ALSO literal-3  [ALSO literal-4] ...               ┘     > │ ...
│                  │                                                                         │ │
│                  │  ┌            ┌ {THROUGH} literal-6                               ┐ ┐    │ │
│                  │  │ literal-5  │ {THRU   }                                         │ │... │ │
│                  │  └            └ ALSO literal-7  [ALSO literal-8] ...             ┘ ┘    │ │
└                  └                                                                        ┘ ┘
[literal-9 IS mnemonic-name]
[CURRENCY SIGN IS literal-10]
[DECIMAL-POINT IS COMMA] .]
```

Technical Notes

1. This paragraph is optional.

2. The reserved word CONSOLE refers to the user's terminal. The assigned mnemonic-name may 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 will print 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 alphabet-name IS clause associates a user-specified name with a sequence of characters that may be used as a character code set, a collating sequence, or both. This character sequence may be either one of the two sequences provided by the compiler or a sequence specified by the user.

   A character code set is specified by referencing alphabet-name in the CODE-SET clause of a file description. When defining a character code set, the alphabet-name IS clause is restricted to STANDARD-1, NATIVE, ASCII, or EBCDIC. A collating sequence is specified by referencing alphabet-name either in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph or in the COLLATING SEQUENCE phrase of a SORT or MERGE statement.

   When STANDARD-1, NATIVE, or ASCII appear in an alphabet-name IS clause, the character code set and collating sequence specified is ASCII. When EBCDIC appears in an alphabet-name IS clause, the character code set and collating sequence specified is EBCDIC.

   When the literal phrase appears in an alphabet-name IS clause, the literals define an ascending collating sequence in the order of their appearance in the phrase. Numeric literals represent the ordinal number of the character within the ASCII character set and must be in the range from 1 through 128. Nonnumeric literals in an alphabet-name IS clause represent themselves. If the literal contains multiple characters, they are assigned successive ascending positions within the collating sequence, starting with the leftmost character. Characters whose positions are not explicitly defined by the literal phrase are assigned positions higher than the specified characters and in their normal ASCII sequence.

   When you specify the THROUGH phrase, the set of contiguous ASCII characters beginning with the character specified by literal-1 and ending with the character specified by literal-2 are assigned successive ascending positions in the collating sequence. The characters specified by a THROUGH phrase may be in either ascending or descending order.

   When you specify the ALSO phrase, the characters specified by literal-1, literal-3, literal-4, ..., are all assigned to the same position in the collating sequence.

   The highest character in the collating sequence, regardless of how it is specified, becomes the figurative constant HIGH-VALUE. If more than one character occupies this position, the last character specified becomes HIGH-VALUE. The lowest character in the collating sequence, regardless of how it is specified, becomes the figurative constant LOW-VALUE. If more than one character occupies this position, the first character specified becomes LOW-VALUE.

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 Section 4.9.26). Literal-1 must be an alphanumeric 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, you must use the literal you specify (instead of the $ character) in PICTURE clauses in the Data Division. For instance, if you wish to insert a currency sign at the front of a field which is to be printed on your report, you must use the literal you specified - not the $ character - as the editing symbol.

   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.
```

3.1.5 INPUT-OUTPUT SECTION

The Input-Output Section names the files and external media required by the object program and provides information required for transmitting and handling 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 3.1.6)

I-O-CONTROL. (See Section 3.1.15)

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

3.1.6 FILE-CONTROL

Function

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

General Format

FORMAT 1:

```
SELECT [OPTIONAL] file-name

   ASSIGN TO device-name-1    [device-name-2] ...

   [ RESERVE integer-1  [AREA ] ]
   [                    [AREAS] ]

   [ORGANIZATION IS SEQUENTIAL]

   [ACCESS MODE IS SEQUENTIAL]


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

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

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

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

FORMAT 2:

```
SELECT file-name

   ASSIGN TO 'device-name-1    [ device-name-2 ] ...

  [ RESERVE integer-1  [AREA ] ]
  [                    [AREAS] ]

    ORGANIZATION IS RELATIVE

  [                 { SEQUENTIAL      RELATIVE KEY IS data-name-1 } ]
  [  ACCESS MODE IS {                                             } ]
  [                 { {RANDOM }                                   } ]
  [                 { {DYNAMIC}    RELATIVE KEY IS data-name-1    } ]

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

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

   [ {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] ] ] ] ] ] ] ]
```

FILE-CONTROL (Cont.)

FORMAT 3:

```
SELECT file-name

   ASSIGN TO  device-name-1      [  device-name-2   ] ...

  [ RESERVE integer-1  [ AREA  ] ]
  [                    [ AREAS ] ]

    ORGANIZATION IS INDEXED
  [                   { SEQUENTIAL } ]
  [ ACCESS MODE IS    { RANDOM     } ]
  [                   { DYNAMIC    } ]
    RECORD KEY IS data-name-1

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

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

   [ { 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] ] ] ] ] ] ] ]
```

Technical Notes

1. This section is optional.

2. All semicolons and commas are optional. Each SELECT clause must end with a period.

3. The SELECT and ASSIGN statements must appear before any other clause shown, and the SELECT statement must precede the ASSIGN statement. Every file described in the Data Division must be named in a SELECT clause in the Environment Division. Thus, the following clause must be specified for every such file: SELECT file-name ASSIGN TO device-name.

4. The individual clauses are described on the following pages in the order shown above.

SELECT

3.1.7 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

```
SELECT file-name

  ASSIGN TO  device-name-1      [ device-name-2 ] ...
```

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 your program tries to open a file which you have declared to be OPTIONAL, the question IS file-name PRESENT? is typed on the operator's console 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 will immediately take the AT END or INVALID KEY path.

NOTE

ISAM files may not be optional. They must be present at program start-up, even if only as dummy files. (Refer to the COBOL-74 Usage Material, Part 3 of this manual, for more information on ISAM.)

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 refer to 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 TOPS-10 Operating System Commands Manual and the TOPS-20 User's Guide.

Logical device-names are names created by the programmer. They can contain up to six characters, and can consist 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 a monitor command (refer to the COBOL-74 Usage Material, Part 3 of this manual, for an explanation of the commands).

4. You may assign more than one device to a file to avoid delay when switching from one reel or unit to the next. When you specify more than one device the object program automatically uses the next device, in a cyclic manner, when an end-of-reel condition is detected. This applies only to tape devices and SORT and ISAM files, and it is unconditional for tapes. For SORT/MERGE, any number of devices may be assigned. If the disks are specified generically, SORT/MERGE will use its internal algorithm to determine which physical devices to use. Otherwise, all devices specified will be used in a round-robin fashion. For ISAM files you may assign not more than two devices.

5. 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.

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

Examples

```
SELECT INFIL ASSIGN TO MTA1.

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

RESERVE

3.1.8 RESERVE

Function

The RESERVE clause allows you to specify the actual number of input/output buffer areas for the compiler to allocate to this file.

General Format

```
[                       [AREA ] ]
[ RESERVE integer-1     [AREAS] ]
```

Technical Notes

1. If you specified the organization for this file as RELATIVE or INDEXED, this clause is ignored and only one buffer area is assigned.
2. If you did not specify RELATIVE or INDEXED organization, the integer specifies the number of buffer areas for the compiler to assign.
3. If you omit this clause for a sequential file, two areas will be assigned.
4. Integer-1 does not have a maximum, but you may run out of available memory if you request too many areas reserved. You may also make your program run slower if you request a large number of areas, since the program will be that much bigger.

Example

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

3.1.9 ORGANIZATION

Function

The ORGANIZATION clause specifies the way in which a file will be organized.

General Format

```
                  ( SEQUENTIAL                                      )
ORGANIZATION IS  <  RELATIVE                                         >
                  ( INDEXED     { DEFERRED   }   OUTPUT             )
                                { CHECKPOINT }
```

Technical Notes

1. The ORGANIZATION clause is required for relative and indexed-sequential files. It is ignored for sequential files.

2. If ORGANIZATION IS SEQUENTIAL and the file is on a random-access device, 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 ORGANIZATION IS RELATIVE, the contents of the data item associated with the RELATIVE 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, or which record is to be deleted by a DELETE statement, or which record will be replaced by a REWRITE statement.

4. If ORGANIZATION IS INDEXED, the contents of the data item associated with the RECORD 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 will be replaced by a REWRITE statement.

5. The DEFERRED OUTPUT option of the ORGANIZATION IS INDEXED clause causes the object-time system to output a block of an indexed-sequential file only when another block must be brought into memory. Normally, to ensure integrity 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, Section 5.9.25.

ORGANIZATION (Cont.)

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

7. If you use the ORGANIZATION IS INDEXED clause, you may also specify the CHECKPOINT OUTPUT option (instead of DEFERRED OUTPUT). If you specify this option, the object-time system will force the buffers to be written out, and all pointers internal to the file to be updated, after every WRITE statement. This will naturally make your program run much more slowly. However, it will also safeguard your file against system crashes, since the file will have been updated after the last WRITE before the crash.

Example

```
SELECT INFIL ASSIGN TO DSK, DSK
     ORGANIZATION IS INDEXED DEFERRED OUTPUT.
```

3.1.10 ACCESS MODE

Function

The ACCESS MODE clause specifies the method used to access the file in question.

General Format

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

Technical Notes

1. If you do not specify the ACCESS MODE clause, ACCESS MODE IS SEQUENTIAL is assumed regardless of the organization of the file.

2. If you specify ACCESS MODE IS DYNAMIC you may access the file either sequentially or randomly.

3. When you specify ACCESS MODE IS SEQUENTIAL, the records in your file are accessed in the sequence dictated by the file organization. Sequential files are accessed in the same order they are added to the file. Relative files are accessed in ascending relative record number order. Indexed files are accessed in ascending record key order.

4. If you choose random access mode, the relative key (for relative files) or the record key (for indexed files) indicates the record to be accessed.

Example

```
SELECT INFILE ASSIGN TO DSK
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS RECKEY.
```

RECORD KEY

3.1.11 RECORD KEY

Function

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

General Format

<u>RECORD</u> KEY IS data-name-1

Technical Notes

1. The RECORD KEY clause is valid only for files whose access mode is INDEXED; it must be specified for those files (refer to the READ statement, Section 5.9.27).

2. You must define the RECORD KEY data-name 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.

3. 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 Section 5.9).

Example

```
SELECT INFIL ASSIGN TO DSK, DSK
     ORGANIZATION IS INDEXED
     RECORD KEY IS RECKEY.
```

3.1.12 RELATIVE KEY

Function

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

General Format

<u>RELATIVE</u> KEY IS data-name-1

Technical Notes

1. The RELATIVE KEY clause is valid only for a file whose organization is RELATIVE; it must be specified for this type of file. This clause cannot be used for a file whose organization is INDEXED or SEQUENTIAL.

2. The RELATIVE 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
     ORGANIZATION IS RELATIVE
     ACCESS MODE IS RANDOM
     RELATIVE KEY IS RKEY.
```

RECORDING MODE/DENSITY/PARITY

3.1.13 RECORDING MODE/DENSITY/PARITY

Function

The RECORDING 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  } ] [                     ] ]
    [ DENSITY IS   { 556  } ] [ PARITY IS { ODD  }  ] ]
    [              { 800  } ] [           { EVEN }  ] ]
    [              { 1600 } ] [                     ] ]
```

Technical Notes

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

ASCII	- The file will be read/written as ASCII records, five 7-bit characters per 36-bit word. Bit 35 (the rightmost bit) is ignored.
SIXBIT	- The file will be read/written as SIXBIT records, six 6-bit characters per 36-bit word with record headers.
BINARY	- The file will be read/written as binary records, 36 bits per word.
F	- The file will be read/written as fixed-length EBCDIC records, four 9-bit characters per 36-bit word. However, for industry-compatible magnetic tape (9-track, with at least 800 bpi density), the file will be 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 of them.

RECORDING MODE/DENSITY/PARITY (Cont.)

V - The file will be 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 at least 800 bpi density), the file will be 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 the user overwrites 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 memory 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 Sections 8.1 and 8.2 of this manual.

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, a line printer), the object-time system will always use 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 will always use 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 will always use 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. Normally, the recording mode specified is used. However, if the recording mode is not specified, the default recording mode will depend on the usage mode. If neither the recording mode nor the usage mode is specified, and the /X switch is not included in the command string to the compiler, the default recording mode is SIXBIT. If the /X switch is present, the default recording mode is F.

RECORDING MODE/DENSITY/PARITY (Cont.)

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 may default to another character set, with undesirable results (see the USAGE clause in Section 4.9.23). 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 system commands which are described in the Operating System Commands Reference Manual for users of TOPS-10, and in the TOPS-20 User's Guide for users of TOPS-20. Remember that not all drives will handle all densities. You should verify that the drive you plan to use will accept the density you specify. 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 a user 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. (It does not round up to a word boundary.) This is only valid on magnetic tape, and applies only to users of TOPS-10. 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)
none	none	EBCDIC (/X)
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

The object-time system automatically makes the conversions necessary to have the recording mode conform to the usage mode of the records. (These conversions may cause your program to run more slowly.)

FILE STATUS

3.1.14 FILE STATUS

Function

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

General Format

$$\left[\begin{Bmatrix}\underline{\text{FILE-STATUS}}\\ \underline{\text{FILE STATUS}}\end{Bmatrix}\ \text{IS data-name-1}\ \left[\text{data-name-2}\ \left[\text{data-name-3}\ \left[\text{data-name-4}\ \left[\text{data-name-5}\ \left[\text{data-name-6}\ \left[\text{data-name-7}\ \left[\text{data-name-8}\right]\right]\right]\right]\right]\right]\right]\right]$$

Technical Notes

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

FILE STATUS (Cont.)

2. You must define the data-names 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 10-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

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

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 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 MEMORY 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 MEMORY WHILE SORT IS IN PROGRESS
8	INSUFFICIENT MEMORY 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 MEMORY BETWEEN .JBFF AND OVERLAY AREA
36 INSUFFICIENT MEMORY 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 I/O RECORD
44 THE TWO LOW-ORDER BYTES OF A BLOCK/RECORD DESCRIPTOR WORD MUST BE ZERO

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. The user 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 memory or written on the disk is probably incorrect. The user does not usually set this bit.
20	IO.DTE	Hard data error. The data read or written has incorrect parity as detected by the hardware. The user's data is probably unrecoverable even after the device has been fixed. This bit is usually not set by the user.
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 the user's quota on the file structure has been exceeded. This bit is usually not set by the user. This error is also returned when the user tries to close a file that has open locks associated with it (via Enqueue/Dequeue).
22	IO.EOF	End-of-file. The user 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. This bit is usually not set by the user.
23	IO.ACT	I/O Active. The disk is actively transmitting or receiving data. This bit is always set by the monitor for its own use.
29	IO.WHD	Write disk-pack headers. This is used in conjunction with the SUSET. monitor call to format a disk pack. (Not used in COBOL)
30	IO.SYN	Synchronous mode I/O. Stop disk after every buffer is read or written. (Not used in COBOL)
31	IO.UWC	User word count, supplied by the user 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	Filename already exists (RENAME) or filename is different (ENTER after LOOKUP) or requested 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. (Not expected to occur)
10	Not enough memory.
11	Device not available.
12	No such device.
13	No 2-register relocation capability. (Not expected to occur)
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 memory 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 nonempty directory. (Not expected to occur)
23	Subdirectory 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. (Not expected to occur)
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) (Not expected to occur)
30	Cannot update file.
31	Low segment overlaps high segment. (Not expected to occur)
32	Not logged in. (Not expected to occur)

4. The FILE STATUS items are the paths of communications between the object-time system 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 the object-time system. Control returns to the object-time system at the conclusion of the USE procedure. The object-time system action is determined by the error number and by the contents of the action-code placed into data-name-3 by the USE procedure. If the action-code is set to 1, the object-time system ignores the error and continues the run. If the action-code is left set

FILE STATUS (Cont.)

to 0, the object-time system issues an error message and terminates the run. If the error-number is 17, the object-time system continues the run independent of the action-code setting. If the action-code is not 0 or 1, the object-time system action is undefined.

When the program comes to a normal termination and you have requested (by loading a "1" into the action-code) that errors be ignored, the object-time system issues the following message:

```
%n ERRORS IGNORED
```

5. Refer to the USE statement in Section 5.9.42 for details of writing USE procedures.

6. If you did not specify the FILE STATUS statement, I/O error recovery processing cannot be performed. If you specify the FILE STATUS statement with only data-name-1 included, you can examine the status of the file, but you cannot specify that the object-time system 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
    ORGANIZATION IS INDEXED
    ACCESS MODE IS RANDOM
    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 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.
```

I-O-CONTROL

3.1.15 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  )                  ]

  [      [RECORD    ]                                           ]
  [ SAME [SORT      ]  AREA FOR file-name-2  {file-name-3} ...  ]  ...
  [      [SORT-MERGE]                                           ]

  [ MULTIPLE FILE TAPE CONTAINS file-name-4  [POSITION integer-3]

      [file-name-5  [POSITION integer-4]  ...     ...]  .  ]]
```

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

   If you use the integer-1 RECORDS option, 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 files are open for input/output (updating), or if any file is open on a device other than magnetic tape, disk, line printer, or terminal, or if an indexed-sequential (ISAM) file is open. Therefore, do not attempt to have a rerun dump taken while a sort is in progress. Also, RERUN cannot be used if overlays are used or if files are open for simultaneous update.

3. The SAME AREA clause specifies that two or more files are to use the same area during processing; this overlapping applies to 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 you specify the RECORD option, the files share only the record area (that is, the area in which the current logical record is processed). All of the files mentioned in the SAME RECORD AREA clause may be open at the same time. A logical record in the SAME RECORD AREA is considered to be a logical record of each opened output file whose name appears in the SAME RECORD AREA clause, as well as the most recently read input file whose name is specified. Since the various DISPLAY usages are represented differently in memory, you must keep track of the usage of the record in the SAME RECORD AREA. You may use the record in any way you would otherwise use it. However, you must be sure that you have a record of the expected usage in the SAME RECORD AREA. If, for example, you plan to use a DISPLAY-7 record in your processing, you must have a DISPLAY-7 record in the SAME RECORD AREA, not a DISPLAY-6 record. You will not get an error message if you attempt to use a DISPLAY-6 record as if it were DISPLAY-7.

   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 several files share 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 may 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 the 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.

   No 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.
```

THIS PAGE INTENTIONALLY LEFT BLANK.

THE ENVIRONMENT DIVISION VERB FORMATS

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

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

OBJECT-COMPUTER.   computer-name

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

   [PROGRAM COLLATING SEQUENCE IS alphabet-name]

   [SEGMENT-LIMIT IS segment-number] .

[SPECIAL-NAMES.

[                  { STANDARD-1                                                            }  ]
[ alphabet-name IS { NATIVE                                                                }  ]
[                  {                                                                       }  ]
[                  {            [ {THROUGH}  literal-2                               ]     }  ]
[                  { literal-1  [ {THRU   }                                          ]     }  ]
[                  {            [ ALSO literal-3   [ALSO literal-4] ...              ]     }  ]
[                  {                                                                       }  ]
[                  {   [            [ {THROUGH}  literal-6                         ] ]     }  ]
[                  {   [ literal-5  [ {THRU   }                                    ] ] ... }  ]
[                  {   [            [ ALSO literal-7   [ALSO literal-8] ...        ] ]     }  ]

[literal-9 IS mnemonic-name]

[CURRENCY SIGN IS literal-10]

[DECIMAL-POINT IS COMMA] .]
```

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
[INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   {file-control-entry} ...

[I-O-CONTROL.

   [                 {END OF {REEL}    }                      ]
   [RERUN EVERY      {       {UNIT}    }  OF file-name-1      ]  ...
   [                 {integer-1 RECORDS}                      ]

   [     [RECORD    ]                                            ]
   [SAME [SORT      ]  AREA FOR file-name-2  {file-name-3} ...   ]  ...
   [     [SORT-MERGE]                                            ]

   [MULTIPLE FILE TAPE CONTAINS file-name-4  [POSITION integer-3]

     [file-name-5  [POSITION integer-4]] ...     ...]  .  ]]
```

GENERAL FORMAT FOR ENVIRONMENT DIVISION

FORMAT 1:

```
SELECT [OPTIONAL] file-name

   ASSIGN TO device-name-1   [device-name-2] ...

   [RESERVE integer-1 [AREA ]]
                      [AREAS]

   [ORGANIZATION IS SEQUENTIAL]

   [ACCESS MODE IS SEQUENTIAL]

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

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

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

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

GENERAL FORMAT FOR ENVIRONMENT DIVISION

FORMAT 2:

```
SELECT file-name

   ASSIGN TO  device-name-1      [  device-name-2  ] ...

   [ RESERVE integer-1  [ AREA  ] ]
   [                    [ AREAS ] ]

     ORGANIZATION IS RELATIVE

   [                    {  SEQUENTIAL        RELATIVE KEY IS data-name-1 } ]
   [  ACCESS MODE IS    {                                                } ]
   [                    { { RANDOM  }                                    } ]
   [                    { { DYNAMIC }     RELATIVE KEY IS data-name-1    } ]

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

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

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

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

GENERAL FORMAT FOR ENVIRONMENT DIVISION

FORMAT 3:

```
SELECT file-name

   ASSIGN TO device-name-1      [ device-name-2 ] ...

  [ RESERVE integer-1 [ AREA  ] ]
  [                   [ AREAS ] ]

   ORGANIZATION IS INDEXED

  [                 { SEQUENTIAL } ]
  [ ACCESS MODE IS  { RANDOM     } ]
  [                 { DYNAMIC    } ]
   RECORD KEY IS data-name-1

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

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

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

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

CHAPTER 4

THE DATA DIVISION

The Data Division, which is 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

3. Data to be sent to or received from the Message Control System or the Transactional Processing System

4. Data which is used by the program in the process of executing (This data can be constant or variable, and may be stored as part of the program or computed by the program during its operation.)

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

3. The Communication Section, which defines the special data items that link a program or subprogram to the Message Control System (MCS-10) or the Transactional Processing System (TPS-20)

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 a calling program

6. The Report Section, which describes the data and format of a report

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

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

4.1 FILE SECTION

The File Section begins with the section-header FILE SECTION. If present, it must be the first section in the Data Division. In the File Section, the characteristics of each file to be processed are described by two types of entries, the file description and the record description.

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 record description, describes the data formats of the logical records in the files.

4.1.1 Record Descriptions

Following the FD file-name entry for a file, or the SD file-name entry for a sort file, a record description is given for each different record format in the file. 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 can be found in Section 4.9.11.

A record description begins with a level-01 entry:

```
01  data-name
```

A complete record description may be as simple as

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

or it may 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.

4.1.2 Elementary Items and Group Items

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

4.1.3 Level Numbers

Level numbers indicate a hierarchy of data items. 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. 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 01 level is not required to be a group item; it may be an elementary item as long as it is referenced as a unit. For example:

```
01 EMPLOYEE-RECORD PICTURE IS X(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 data items that are defined only in the Working-Storage Section to define constant values or to store intermediate results. Defining a level-77 item is the equivalent of defining a level-01 elementary item.

Level-66 data items are those items that contain an explicitly specified portion of a record already defined, or even the whole record. A data item with a level number 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 may 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...

4.2 SCHEMA SECTION

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

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 will be found in the Data Base System Programmer's Procedures Manual.

4.3 COMMUNICATION SECTION

The Communication Section contains the definitions of input and output communication-description entries.

CD entries define records called CD records which contain special data items used to link the program to the Message Control System for users of TOPS-10 or the Transactional Processing System for users of TOPS-20.

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 your TOPS-10 installation does not include MCS, or your TOPS-20 installation does not have TPS, the Communication Section cannot be used.

Details of the Communication Section entries will be found in the Message Control System Programmer's Procedures Manual for users of TOPS-10, and the Transactional Processing System Programmer's Procedures Manual for users of TOPS-20.

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, RD, CD, or SCHEMA entries.

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

The maximum size of a record in Working Storage is 4095 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 RECORD KEY and RELATIVE KEY data items 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,
            .                                       SUB1, SUB2.
            .                                         .
     CALL ENTRPT USING MAIN,                          .
            MAIN1, MAIN2.                             .
            .                                  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, references to SUB1 refer to MAIN1, and so on through the list. See the COBOL-74 Usage Material, Part 3 of this manual, 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; any items that do not can be made to do so by means of 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.

4.6.1 Format Of 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 break happens least often and the most minor control break happens 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
        .
        .
        .
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 of 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 print queue using the PRINT command and specifying the code with the REPORT switch, as follows:

```
PRINT file-specifier/REPORT:code
```

Only the first 12 characters of the code will be 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 should be placed, and the number of lines which should be 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 programmer-specified procedure to be performed before a report group is produced.

4.7 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 is sufficient.

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 OF 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 may be used interchangeably.

The only data items which need to 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 may be qualified by the file name, as may any item contained within the record. File names must be unique.

Level-66 items may 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.

4.8 SUBSCRIPTING AND INDEXING

It may sometimes be more convenient for you to specify a set of data values as a table rather than assign 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-74, 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 will be treated as a COMPUTATIONAL data item by the compiler.

COBOL-74 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 immediately after the terminal space that follows the data-name. 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 10 times, an intermediate element occurring 5 times within each occurrence of the major element, and a minor element occurring 3 times within each intermediate element. The last major element of the table is referred to by the subscript form (10,1,1), while the final element of the table is referred to by (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 which an integer is added or subtracted. The form for direct subscript/indexing is shown in Figure 4-1.

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

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 - all enclosed in the parentheses immediately following the terminal space of the data-name. The form for relative subscripting/indexing is shown in Figure 4-2.

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

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.

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

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

Figure 4-3 Qualified Direct Subscripting/Indexing

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

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

you could specify the following:

```
ANAME OF ASUBGROUP1 OF AGROUP1 OF AREC1 (I,J,4)
```

NOTE

Subscripts may not be subscripted.

4.9 DATA DIVISION CLAUSES

The clauses which make up the Data Division are presented in the following pages. The function, syntax, and details of each clause are described, and the general format of the clause is included. The clauses are presented in the order in which they appear in the general formats at the end of this chapter, that is, in the order in which they occur in the Data Division. The formats of some clauses contain other clauses. When this is the case each clause which is subordinate is described separately on succeeding pages.

FILE DESCRIPTION (FD)

4.9.1 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

```
 DATA DIVISION.

[FILE SECTION.

[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 }

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

      [DATE-WRITTEN IS { data-name-2 }] [USER-NUMBER IS { data-name-3 }]
                       { literal-2   }                  { literal-3   }

      [DATA { RECORD IS   } data-name-4 [data-name-5] ...]
            { RECORDS ARE }

      [LINAGE IS { data-name-6 } LINES [WITH FOOTING AT { data-name-7 }]
                 { integer-5   }                        { integer-6   }

           [LINES AT TOP { data-name-8 }] [LINES AT BOTTOM { data-name-9 }]]
                         { integer-7   }                   { integer-8   }

      [CODE-SET IS alphabet-name]

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

FILE DESCRIPTION (FD) (Cont.)

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

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

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 semicolons and commas are optional. The entire FD entry must terminate with a period.

3. The clauses may 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 specify the RECORDING MODE clause for a file in the FD, you cannot also specify 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 Section 3.1.13.

5. The maximum number of files that can be open at one time is 16. ISAM files count as two files: one index (.IDX) file and one data (.IDA) file.

BLOCK CONTAINS

4.9.2 BLOCK CONTAINS

Function

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

General Format

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

Technical Notes

1. If you do not include this clause, or if you specify that integer-2 is zero, the file will not be organized into logical blocks when it is written. Rather, all records will be placed in the file with no empty space. The file is then considered to be "unblocked" or "blocked zero".

2. If you use the CHARACTERS option, you specify the logical block size 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), the compiler assumes that the size is specified in terms of ASCII characters. If the recording mode is SIXBIT (that is, the records for the file are all described, explictly or implicitly, as DISPLAY-6), the compiler assumes 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), the compiler assumes that the size is specified in terms of EBCDIC characters, either fixed- or variable-length. When variable-length EBCDIC records are used (that is, 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 will be 9.

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

4. Files whose organizations are RELATIVE or INDEXED must have a nonzero blocking factor.

4.9.3 CODE-SET

FUNCTION

The CODE-SET clause specifies the character code set used to represent data on the external media.

General Format

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

Technical Notes

1. When you specify the CODE-SET clause for a file, you must describe all data in that file as USAGE IS DISPLAY. You must also describe any signed numeric data with the SIGN IS SEPARATE clause.

2. The alphabet-name clause referenced by the CODE-SET clause must not specify the literal phrase.

3. You may specify the CODE-SET clause only for files not residing on mass storage media.

4. The CODE-SET clause is included only for compatability, since ASCII is the only alphabet-name allowed, and ASCII is also the default.

5. If you include the CODE-SET clause, alphabet-name specifies the character code convention used to represent data on the external media. It also specifies the algorithm for converting the character codes on the external media from or to the native character codes. This code conversion occurs during the execution of an input or output operation.

6. If you omit the CODE-SET clause, the ASCII character set is assumed for data on the external media.

DATA RECORD

4.9.4 DATA RECORD

Function

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

General Format

$$\left[\underline{\text{DATA}}\left\{\begin{array}{l}\underline{\text{RECORD}}\ \text{IS}\\ \underline{\text{RECORDS}}\ \text{ARE}\end{array}\right\}\ \text{data-name-4}\quad\left[\text{data-name-5}\right]\ \ldots\right]$$

Technical Notes

1. This clause is optional because all records in the FD entry are assumed to be data records.

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

3. All record-names must be specified in 01-level data entries subordinate to this FD entry. The presence of more than one such record-name indicates that the file contains more than one type of data record. These records may have different descriptions. The order in which they are listed is not significant.

4.9.5 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

4.9.6 LABEL RECORD

Function

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

General Format

$$\underline{\text{LABEL}} \left\{ \begin{array}{l} \underline{\text{RECORD}}\ \text{IS} \\ \underline{\text{RECORDS}}\ \text{ARE} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{STANDARD}} \\ \underline{\text{OMITTED}} \\ \text{record-name-1} \end{array} \right\}$$

Technical Notes

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

2. You should use the OMITTED option when the file has no header or trailer labels.

3. You should use the STANDARD option when the file has header and trailer labels that conform to the standard format. If the file you are describing is on disk or DECtape, you must either specify LABEL RECORDS ARE STANDARD, or omit the clause altogether allowing the default to take over. 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 disk 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. 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. 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.

5. If PULSAR is running on your TOPS-10 system, you must perform a MOUNT to get a tape. PULSAR writes labels in a different format from the label format explained here. (Refer to the reference material provided with PULSAR for more information.) PULSAR labeling depends on the type of labeling you specify at MOUNT time. It is recommended that you make the LABEL RECORD clause and the value specified for the /LABELS: switch on the mount agree.

   LABEL RECORDS can have two values: STANDARD and OMITTED. These values have the following equivalents in PULSAR labeling: /LABELS:STANDARD; /LABELS:NONE. You must specify the PULSAR label on the mount and the COBOL label in your program.

   If you use STANDARD in your program, that is equivalent to /LABELS:STANDARD in PULSAR.

   If you use OMITTED in your program, that is equivalent to /LABELS:NONE in PULSAR.

Table 4-1
Standard Label for Magtapes

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

4.9.7 RECORD CONTAINS

Function

The RECORD CONTAINS clause specifies the size of the data records in the file to which it refers.

General Format

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

Technical Notes

1. Since the record description entry completely defines the size of the data record, this clause is never required. However, if you use it, it replaces the record description entry in setting the size of the record.

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

3. The data record size is equal to the number of character positions required to contain the record.

4.9.8 REPORT

Function

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

General Format

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

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 you use this clause, you may omit the data record description 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

4.9.9 SD File-name

Function

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

General Format

```
[SD  file-name
     [RECORD CONTAINS  [integer-1 TO]  integer-2 CHARACTERS]

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

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

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.

4.9.10 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. The VALUE OF DATE-WRITTEN clause specifies a date which the file label must contain to be processed by the program. The VALUE OF USER-NUMBER clause provides a project-programmer number to be checked against the file label before processing.

General Format

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

  [ DATE-WRITTEN IS { data-name-2 } ] [ USER-NUMBER IS { data-name-3 } ] ]
  [                 { literal-2   } ] [                { literal-3   } ] ]
```

Technical Notes

1. ID may be substituted for IDENTIFICATION.
2. 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.
3. The three clauses can be written in any order, but only one of each can be specified for a file.
4. 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, DISPLAY-6, DISPLAY-7, or DISPLAY-9 data item nine characters in length. If a literal is specified, it must be a nonnumeric 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 period which the system prints between the file-name and the extension must not be included in the VALUE OF IDENTIFICATION clause.

VALUE OF IDENTIFICATION/DATE-WRITTEN/USER-NUMBER (Cont.)

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).
```

5. DATE-WRITTEN represents the date that a file (with STANDARD labels) was written. If a data-name is specified, it must be associated with a DISPLAY, DISPLAY-6, DISPLAY-7 or DISPLAY-9 data item six characters in length. If a literal is specified, it must be a nonnumeric 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).
```

6. 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.

7. 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.

8. 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 may 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.

VALUE OF IDENTIFICATION/DATE-WRITTEN/USER-NUMBER (Cont.)

9. 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 will be translated from decimal to binary by the COBOL conversion routine. Thus, the project-programmer will not be accurate unless you provide a conversion routine in your program to convert your octal project-programmer number to its decimal equivalent so that it will be converted to the correct binary number. The following example is a suggested method for performing the conversion.

```
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.
```

VALUE OF IDENTIFICATION/DATE-WRITTEN/USER-NUMBER (Cont.)

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 may be altered during program execution only if all files referenced have identical parameters.

c. If several files will be 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).

10. None of the data-names in the VALUE OF clauses can appear in the Linkage Section.

4.9.11 DATA DESCRIPTION ENTRY

Function

A data description entry describes a particular item of data.

General Format

FORMAT 1:

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

  [ REDEFINES data-name-2 ]

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

  [              { COMPUTATIONAL   } ]
  [              { COMP            } ]
  [              { COMPUTATIONAL-1 } ]
  [              { COMP-1          } ]
  [              { COMPUTATIONAL-3 } ]
  [              { COMP-3          } ]
  [   USAGE IS   { DISPLAY         } ]
  [              { DISPLAY-6       } ]
  [              { DISPLAY-7       } ]
  [              { DISPLAY-9       } ]
  [              { INDEX           } ]
  [              { DATABASE-KEY    } ]
  [              { DBKEY           } ]

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

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

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

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

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

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

DATA DESCRIPTION ENTRY (Cont.)

FORMAT 2:

$$66\ \text{data-name-1}\ \underline{\text{RENAMES}}\ \text{data-name-2}\ \left[\left\{\begin{matrix}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{matrix}\right\}\ \text{data-name-3}\right]$$

FORMAT 3:

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

$$\left[\text{literal-3}\left[\left\{\begin{matrix}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{matrix}\right\}\ \text{literal-4}\right]\right]\ \ldots\ .$$

The clauses shown in the General Format appear in alphabetical order along with the other Data Division clauses 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 may 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 item, DATABASE-KEY, or DBKEY.
5. The clauses SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO can be specified only at the elementary level.

4.9.12 BLANK WHEN ZERO

Function

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

General Format

[<u>BLANK</u> WHEN <u>ZERO</u>]

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. An asterisk used as a zero suppression symbol in a PICTURE clause may not appear in the same entry with the BLANK WHEN ZERO clause. More comprehensive editing features are available in the PICTURE clause.

4. When the BLANK WHEN ZERO clause is used for an elementary item whose PICTURE is numeric, the category of the item is considered to be numeric-edited.

Condition-name (level-88)

4.9.13 Condition-name (level-88)

Function

The condition-name (level-88) entry assigns a name to a value or range of values of the associated data item.

General Format

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

$$\left[\text{literal-3} \left[\left\{ \begin{array}{l} \underline{\text{THROUGH}} \\ \underline{\text{THRU}} \end{array} \right\} \text{ literal-4} \right] \right] \ldots$$

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 any 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 may 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 1 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 999 must be in the range of 000 to 999.

Data-name/FILLER

4.9.14 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

$$\text{level-number} \left\{ \begin{array}{l} \text{data-name-1} \\ \underline{\text{FILLER}} \end{array} \right\}$$

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 one 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-Defined 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 may 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.

4.9.15 JUSTIFIED

Function

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

General Format

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

Technical Notes

1. The JUSTIFIED clause cannot be specified at a group level, or for numeric or edited items. If neither RIGHT nor LEFT is specified, RIGHT is assumed.

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

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

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

   a. The receiving data item is described as numeric or numeric-edited (see definition in Notes 7 and 10 under the PICTURE clause, Section 4.9.18.)

      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. The receiving data item is described as alphanumeric or alphabetic (see definition in Notes 6 and 8 under the PICTURE clause, Section 4.9.18).

      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.)

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

6. 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 rightmost 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.
```

Procedure Division statement	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	Δ	Δ
MOVE ITEM-B TO ITEM-D.	Δ	Δ	W	X	Y	Z

4.9.16 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

$$\text{level-number} \quad \left\{ \begin{array}{l} \text{data-name-1} \\ \underline{\text{FILLER}} \end{array} \right\}$$

Technical Notes

1. A level-number is required as the first element in each data description entry.
2. Level-numbers may be placed anywhere on the source line, at or after margin A.
3. Level-number 88 is described under "condition-name (level-88)", Section 4.9.13, and level-number 66 is described under "RENAMES (level-66)", Section 4.9.20.
4. A further description of level-numbers and data hierarchy can be found in the introduction to this chapter.

OCCURS

4.9.17 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

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

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 phrase must be included. The DEPENDING option must immediately follow TIMES. Data-name-1 must be a positive integer, and for efficiency should be either USAGE INDEX or USAGE COMP. It cannot be subcripted, and if the clause appears in the Linkage Section, data-name-1 must be either USAGE INDEX or USAGE COMP.

7. The KEY IS option indicates that you have sorted the repeated data 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. Note that you must sort the data - it will not be sorted automatically.

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 Section 4.8.

11. The entire record containing the OCCURS clause must not exceed 32,767 characters in size; that is, if the record were completely full of data, the number of characters required to contain the record would have to be less than or equal to 32,767.

PICTURE

4.9.18 PICTURE

Function

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

General Format

$$\left[\left\{ \begin{array}{l} \underline{\text{PICTURE}} \\ \underline{\text{PIC}} \end{array} \right\} \text{ IS character-string} \right]$$

Technical Notes

1. A PICTURE clause may be specified only for an elementary data item. It may not be used with an item described as USAGE INDEX, COMP-1, or DATABASE-KEY (DBKEY).

2. PIC may be substituted for PICTURE in the format.

3. 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, tab, and 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)

   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 $ appears)[1]

[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 $.

, represents an insertion comma[1]
. represents an actual decimal point[1]
B represents an insertion blank
0 represents an insertion zero
/ represents an insertion slash

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. However, some editing symbols may not be used more than once in a data item: "S", "V", ".", "CR", and "DB".

4. A maximum number of 30 symbols can appear in a picture string. Note that the number of symbols in a picture string and the 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 may 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, with one exception, do not represent actual physical character positions within the data item. The exception involves the use of the SIGN IS SEPARATE clause, which causes the S (arithmetic sign) to take up a character position. If the clause is omitted, the character-string

```
S999V99
```

represents a 5-position data item. However, if the SIGN IS SEPARATE clause is included, the character-string would represent a 6-position item.

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

5. 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.

[1] If the DECIMAL-POINT IS COMMA clause appears in the SPECIAL-NAMES paragraph, the function of the comma and decimal point is reversed.

PICTURE (Cont.)

6. Definition of an Alphabetic Item

   a. Its picture string may contain only the symbol A or B.

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

7. Definition of a Numeric Item

   a. Its picture string may 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 may contain only the digits 0 through 9 and an operational sign.

8. 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).

9. 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, 0 or /.

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

10. 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 may also contain the symbols 9, V, or P. If you use the CURRENCY SIGN IS clause, the new currency sign you specify replaces the $ in the above list.

       The allowable sequences are determined by certain editing rules for each symbol and can be found in Note 11.

       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.

11. 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 will be 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 15, 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 if 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; in either case, it is redundant.

Examples:

PPP9999 (or VPPP9999) defines a data item of four character positions whose contents will be 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 will be 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 will retain 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 will be stored in the item as an absolute value and will be treated as positive in all operations. The S symbol is not counted in the size of the data item unless the SIGN IS SEPARATE clause is included, in which case it occupies one character position.

PICTURE (Cont.)

V	A V in a picture string indicates the location of the assumed decimal point and may 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	Δ Δ 0 3 0 5
MOVE A-FLD TO B-FLD	ZZZZZZ	Δ Δ Δ 3 0 5
MOVE A-FLD TO B-FLD	ZZZZ.ZZ	Δ 3 0 5 . 0 0

Also see Note 19, 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 will be 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 will be inserted. The comma is counted in the size of the item.
/	Each / in a picture string represents a character position into which the slash will be inserted. The slash is counted in the size of the item.

Examples: (A-FLD contains 362577)

	B-FLD picture string	Result
MOVE A-FLD TO B-FLD	9,999,999	0,362,577
MOVE A-FLD TO B-FLD	Z,ZZZ,ZZZ	ΔΔ362,577

Also see Note 15, Simple Insertion Editing.

. A . (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 period (.) is to be inserted. This symbol is counted in the size of the item. Only one . may 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	03,526.99
MOVE A-FLD TO B-FLD	ZZ,ZZZ.ZZ	Δ3,526.99
MOVE A-FLD TO B-FLD	99999.9999	03526.9900

See Note 4 under the MOVE clause, Section 5.9.23, for a clarification of the rule governing the third example.

Also see Note 16, 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 will be 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 will contain a + if the sending field either was unsigned (absolute) or had a positive operational sign; it will contain a - if the sending field had a negative operational sign.

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

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

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	3 4 5 6 . 2 5 Δ Δ Δ
MOVE B-FLD TO C-FLD	9999.99BCR	3 4 5 6 . 2 5 Δ C R
MOVE A-FLD TO C-FLD	+9999.99	+ 3 4 5 6 . 2 5
MOVE B-FLD TO C-FLD	+9999.99	- 3 4 5 6 . 2 5
MOVE A-FLD TO C-FLD	-9999.99	Δ 3 4 5 6 . 2 5
MOVE B-FLD TO C-FLD	-9999.99	- 3 4 5 6 . 2 5
MOVE A-FLD TO C-FLD	9999.99DB	3 4 5 6 . 2 5 Δ Δ
MOVE B-FLD TO C-FLD	9999.99DB	3 4 5 6 . 2 5 D B
MOVE B-FLD TO C-FLD	$9999.99+	$ 3 4 5 6 . 2 5 -

Also see Note 17, 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. (In order for floating to go past decimal point, all numeric positions of the item must be represented by the floating insertion symbol.)

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

PICTURE (Cont.)

Examples: (A-FLD contains $0056_{\wedge}25$; B-FLD contains $-0056_{\wedge}25$)

	C-FLD picture string	Result
MOVE A-FLD TO C-FLD	++999.99	Δ+056.25
MOVE B-FLD TO C-FLD	++++9.99	ΔΔ-56.25
MOVE ZERO TO C-FLD	++999.99	Δ+000.00
MOVE ZERO TO C-FLD	+++++.++	ΔΔΔΔΔΔΔΔ
MOVE A-FLD TO C-FLD	--999.99	ΔΔ056.25
MOVE B-FLD TO C-FLD	--999.99	Δ-056.25
MOVE ZERO TO C-FLD	---99.99	ΔΔΔ00.00
MOVE ZERO TO C-FLD	--------	ΔΔΔΔΔΔΔ

Also see Note 18, 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 $3456_{\wedge}75$)

	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 17, 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.

PICTURE (Cont.)

Examples: (A-FLD contains 005625)

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

Also see Note 18, Floating Insertion Editing.

12. 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

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

Category	Type of Editing Allowed
Alphabetic	Simple insertion: B only
Numeric	None
Alphanumeric	None
Alphanumeric-edited	Simple insertion: 0, B and /
Numeric-edited	All (except for the restriction given in Note 14)

14. 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.

15. Simple Insertion Editing (, B 0 /)

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

16. 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. Since the . cannot be the last symbol in the character-string, it must be immediately followed by one of the line-ending characters, either space or carriage return.

17. 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 $ will always be 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 $,$$$ then no digit will ever appear to the left of the comma; a $ will always be there.) A comma is omitted only when what appears to its left consists only of zeroes. (With the picture string $,$$$ the comma is never omitted.)

PICTURE (Cont.)

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

18. 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. Representing 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 will be 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. Representing 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.

19. 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.

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) will be set to the replacement character (with the exception of the decimal point if the suppresson symbol is an *).

The * and the clause BLANK WHEN ZERO may not appear in the same entry.

PICTURE (Cont.)

Example:

(A-FLD contains 023456, B-FLD contains 001200)

	R-FLD PICTURE String	Result of MOVE
MOVE A-FLD TO R-FLD	****.**	*234.56
MOVE B-FLD TO R-FLD	XXXX.XX	**12.00 (1)
MOVE A-FLD TO R-FLD	ZZZZ.ZZ	234.56 (1)
MOVE B-FLD TO R-FLD	ZZZZ.ZZ	12.00
MOVE ZERO TO R-FLD	****.**	****.** (2)
MOVE ZERO TO R-FLD	ZZZZ.ZZ	ΔΔΔΔΔΔΔ (3)
MOVE ZERO TO R-FLD	+****.**	*****.** (4)
MOVE ZERO TO R-FLD	+ZZZZ.ZZ	ΔΔΔΔΔΔΔΔ (5)

(1) Zero supression does not take place to the right of the decimal point.
(2) Decimal point is not suppressed.
(3) Decimal point is replaced by a space.
(4) Plus sign (+) is replaced by a space.
(5) Both sign and decimal point are replaced by space.

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

21. Figure 4-4 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.

$\{ \quad \}$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.

		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-024-79

Figure 4-4 Picture String Character Chart

REDEFINES

4.9.19 REDEFINES

Function

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

General Format

[REDEFINES 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 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. However, the REDEFINES clause may be used to redefine an item whose picture contains the OCCURS clause.

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 as data-name-1. FILLER items may be used to comply with this rule.

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

6. The redefinition entries cannot contain VALUE clauses.

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

8. 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).
```

REDEFINES (Cont.)

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)
	FIELD-3	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	

RENAMES (level-66)

4.9.20 RENAMES (level-66)

Function

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

General Format

$$\text{66 data-name-1 }\underline{\text{RENAMES}}\text{ data-name-2 }\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\text{ data-name-3}\right]$$

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. The words THRU and THROUGH are equivalent.

4. Data-name-2 and data-name-3 must be the names of items in the associated 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).

If data-name-3 is not specified, data-name-2 can be either a group item or an 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.

5. 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.
```

SIGN

4.9.21 SIGN

Function

The SIGN clause specifies the position and the mode of representation of the operational sign.

General Format

$$\left[\left[\underline{\text{SIGN}}\ \text{IS} \right] \left\{ \begin{array}{l} \underline{\text{LEADING}} \\ \underline{\text{TRAILING}} \end{array} \right\} \left[\underline{\text{SEPARATE}}\ \text{CHARACTER} \right] \right]$$

Technical Notes

1. The optional SIGN clause, if present, specifies the position and the mode of representation of the operational sign for the numeric data description entry to which it applies, or for each numeric data description entry subordinate to the group to which it applies. The SIGN clause applies only to numeric data description entries whose PICTURE contains the character S; the S indicates the presence of an operational sign. However, it does not indicate the representation or the position of the sign.

2. The numeric data description entries to which the SIGN clause applies must be described as USAGE IS DISPLAY.

3. At most one SIGN clause may apply to any given numeric data description entry.

4. If the CODE-SET clause is specified, any signed numeric data description entries associated with that file description entry must be described with the SIGN IS SEPARATE clause.

5. A numeric data description entry whose PICTURE contains the character S, but to which no optional SIGN clause applies, has an operational sign which is associated with the trailing digit position of the elementary item.

6. If the optional SEPARATE CHARACTER phrase is not present, the following rules apply:

   a. The operational sign will be presumed to be associated with the trailing digit position of the elementary numeric data item.

   b. The letter S in a PICTURE character-string is not counted in determining the size of the item (in terms of standard data format characters).

7. If the optional SEPARATE CHARACTER phrase is present, the following rules apply:

   a. There is no default condition for the operational sign in this case. You may specify the SEPARATE CHARACTER phrase only when either LEADING or TRAILING is also specified.

   b. The letter S in a PICTURE character-string is counted in determining the size of the item (in terms of standard data format characters).

   c. The operational signs for positive and negative are the standard data format characters + and -, respectively.

   d. The various possiblities for the SIGN and SEPARATE CHARACTER clauses are illustrated below: (value is -111)

Options	**SIXBIT Representation**
none	00000011J
SIGN LEADING	]00000111
SIGN TRAILING	00000011J
SIGN LEADING SEPARATE	-000000111
SIGN TRAILING SEPARATE	000000111-

8. Every numeric data description entry whose PICTURE contains the character S is a signed numeric data description entry. If a SIGN clause applies to such an entry and conversion is necessary for purposes of computation or comparisons, conversion takes place automatically.

SYNCHRONIZED

4.9.22 SYNCHRONIZED

Function

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

General Format

$$\left[\begin{Bmatrix}\underline{\text{SYNCHRONIZED}}\\ \underline{\text{SYNC}}\end{Bmatrix}\quad\left[\begin{matrix}\underline{\text{LEFT}}\\ \underline{\text{RIGHT}}\end{matrix}\right]\right]$$

Technical Notes

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

2. This clause is optional. If you omit it the default is SYNCHRONIZED LEFT.

3. 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.

4. SYNCHRONIZED LEFT or SYNC LEFT specifies that the item is to be positioned in such a way that it will begin 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 will terminate at the right boundary of a computer word.

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

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

SYNCHRONIZED (Cont.)

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

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

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

USAGE

4.9.23 USAGE

Function

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

General Format

```
[             { COMPUTATIONAL    } ]
[             { COMP             } ]
[             { COMPUTATIONAL-1  } ]
[             { COMP-1           } ]
[             { COMPUTATIONAL-3  } ]
[             { COMP-3           } ]
[  USAGE IS   { DISPLAY          } ]
[             { DISPLAY-6        } ]
[             { DISPLAY-7        } ]
[             { DISPLAY-9        } ]
[             { INDEX            } ]
[             { DATABASE-KEY     } ]
[             { DBKEY            } ]
```

Technical Notes

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

2. 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.

   The recording mode of a file determines how the data is recorded on the external medium. The recording mode may 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.

3. All group items are treated as DISPLAY items, either DISPLAY-6, DISPLAY-7, or DISPLAY-9. The maximum size for any such item is 4,096 characters.

4. COMPUTATIONAL (COMP)

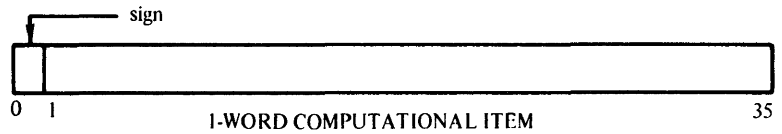
   a. COMP is equivalent to COMPUTATIONAL.

   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.

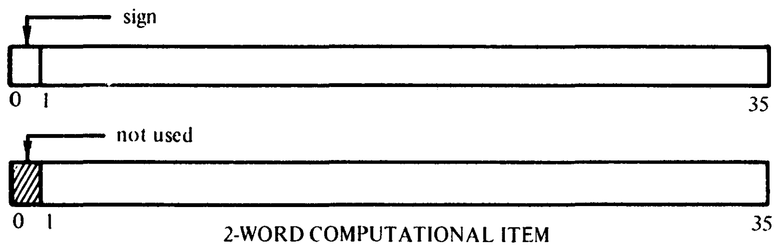
   c. If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. However, the group itself is not COMPUTATIONAL and cannot be used as an operand in arithmetic computations. See Note 3 above.

   d. COMPUTATIONAL items of 10 or fewer decimal positions will be SYNCHRONIZED RIGHT in one computer word. Computational items of more than 10 decimal positions will be SYNCHRONIZED RIGHT in two full computer words. The maximum size of a COMP item is 18 digits.

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



1-WORD COMPUTATIONAL ITEM



2-WORD COMPUTATIONAL ITEM

5. COMPUTATIONAL-1 (COMP-1)

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

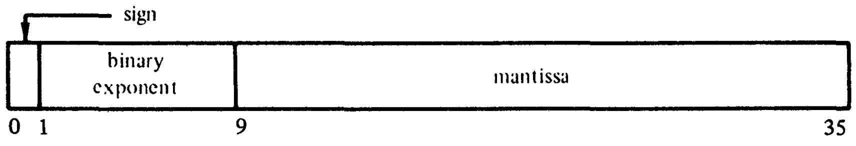
   b. A COMPUTATIONAL-1 item can contain a value, in floating point format, to be used in computations. It must be numeric. A COMP-1 item must not have a PICTURE.

   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. See Note 3 above.

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

USAGE (Cont.)

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



6. COMPUTATIONAL-3 (COMP-3)

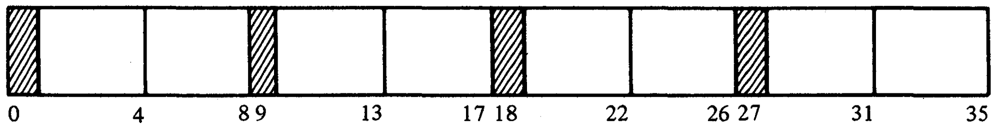
a. COMP-3 is equivalent to COMPUTATIONAL-3.

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. See Note 3 above.

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 may be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT.

g. COMP-3 items may share a computer word with other COMP-3 items or with DISPLAY-9 items. However, COMP-3 items will 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 may 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 nonprinting plus sign. Although octal 12, 14 and 16 represent plus signs, the sign given to the positive result of any arithmetic operation will be 14. Similarly, the minus sign given to the negative result of any arithmetic operation will be 15.

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

7. DISPLAY

   a. DISPLAY is equivalent to DISPLAY-6. However, you may change DISPLAY to be DISPLAY-7 or 9 with the DISPLAY IS clause. You may also cause the compiler to consider all DISPLAY items to be DISPLAY-9 by using the /X switch when compiling your program.

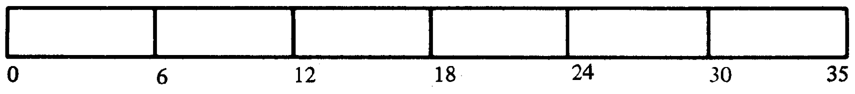
8. DISPLAY-6

   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 may contain any picture symbols. Refer to Appendix C for the SIXBIT collating sequence.

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

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



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

9. DISPLAY-7

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

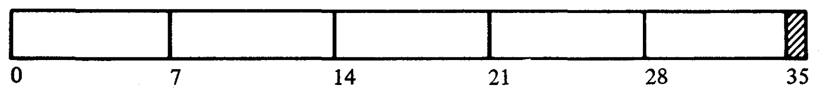
   b. DISPLAY-7 items can be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT, as desired; otherwise, they may share a computer word with other items. If the item is SYNCHRONIZED RIGHT, the last character of the item will end in bit 34 of a computer word.

   c. Bit 35 of a word represented in this format is never used.

   d. The maximum length of a DISPLAY-7 item is 4,096 characters.

USAGE (Cont.)

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



10. DISPLAY-9

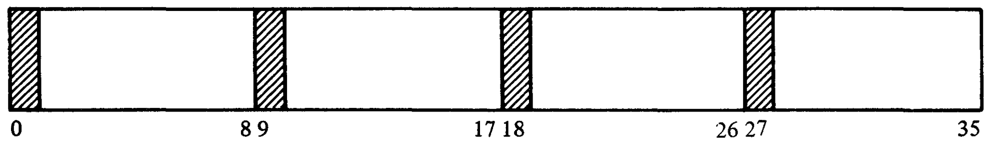
a. DISPLAY is equivalent to DISPLAY-9 when the /X switch is included in the command string to the compiler.

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

c. DISPLAY-9 items may be SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT as desired; otherwise, they may share a computer word with other DISPLAY-9 or COMP-3 items. If the item is SYNCHRONIZED RIGHT, the last character of the item will end 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 and the /X switch has been included in the compiler command string, its USAGE is assumed to be DISPLAY-9 .

11. 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.

12. 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.

   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 TOPS-10 Data Base Management System (DBMS-10), or the TOPS-20 Data Base Management System (DBMS-20). This item can be used to store the value of a data base key. All data base keys are assigned by DBMS and you cannot change them. Refer to the DBMS-10 Programmer's Procedures Manual for more information about DBMS-10, or the DBMS-20 Programmer's Procedures Manual for DBMS-20.

VALUE

4.9.24 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 ] ] ... ]
```

Technical Notes

1. Format 2 can be specified only for level-88 items.
2. The words THRU and THROUGH are equivalent.
3. In the File Section and the Linkage 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.
4. 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.
5. If no VALUE clause appears for a Working-Storage item, the initial value of the item is unpredictable.
6. More information concerning Format 2 can be found under Condition-name (Level-88) in Section 4.9.13.

7. 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 may 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 nonnumeric literals. The literal will be aligned according to the normal alignment rules (see the JUSTIFIED clause, Section 4.9.15) 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.

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

4.9.25 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  } { FINAL                                        } ]
    [ { CONTROLS ARE } { identifier-1  [ identifier-2] ...            } ]
    [                  { FINAL identifier-1  [ identifier-2]  ...     } ]

    [ PAGE { LIMIT  IS  }  integer-1  { LINE  }
    [      { LIMITS ARE }             { LINES }

        [ HEADING  integer-2 ] [ FIRST  DETAIL  integer-3 ]

        [ LAST DETAIL   integer-4 ] [ FOOTING  integer-5 ] ]  .
```

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 may 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 in this chapter separately, in alphabetical order.

4.9.26 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

```
[CODE 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, described in Section 3.1.4.
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.

4.9.27 CONTROL

Function

The CONTROL clause indicates the identifiers that control the printing of totals in the report.

General Format

$$\left[\left\{ \begin{array}{l} \underline{\text{CONTROL}}\ \text{IS} \\ \underline{\text{CONTROLS}}\ \text{ARE} \end{array} \right\} \quad \left\{ \begin{array}{l} \underline{\text{FINAL}} \\ \text{identifier-1} \quad [\text{identifier-2}]\ \ldots \\ \underline{\text{FINAL}}\ \text{identifier-1} \quad [\text{identifier-2}]\ \ldots \end{array} \right\} \right]$$

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 be defined in the File or Working-Storage Section of the Data Division. They cannot be subscripted or indexed.

Report Group Description

4.9.28 Report Group Description

Function

The Report Group Description entry specifies the characteristics and format of a particular report group.

General Format

Format 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                                                 ┘

    ┌                 ┌ DISPLAY   ┐ ┐
    │ [USAGE IS]      │ DISPLAY-6 │ │ .
    │                 │ DISPLAY-7 │ │
    └                 └ DISPLAY-9 ┘ ┘
```

Format 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 } ]
```

Report Group Description (Cont.)

Technical Notes

1. Except for the data-name, which when present must immediately follow the level-number, the clauses may 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 both 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, which is Section 4.9.11.

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 may be used in addition to the clauses in Format 1.

7. The remaining clauses are described in detail on the following pages.

4.9.29 COLUMN NUMBER

Function

The COLUMN NUMBER clause indicates the column on the printed page in which the high-order (leftmost) character of an item will be printed.

General Format

```
[COLUMN NUMBER IS integer-1]
```

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

4.9.30 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.

4.9.31 LINAGE

Function

The LINAGE clause specifies the size of a logical page in terms of number of lines. It can also specify the size of the top and bottom margins on the logical page and the line number, within the page body, at which the footing area begins.

General Format

$$\left[\underline{\text{LINAGE}}\ \text{IS} \begin{Bmatrix} \text{data-name-6} \\ \text{integer-5} \end{Bmatrix} \text{LINES} \left[\text{WITH}\ \underline{\text{FOOTING}}\ \text{AT} \begin{Bmatrix} \text{data-name-7} \\ \text{integer-6} \end{Bmatrix} \right] \left[\text{LINES AT}\ \underline{\text{TOP}} \begin{Bmatrix} \text{data-name-8} \\ \text{integer-7} \end{Bmatrix} \right] \left[\text{LINES AT}\ \underline{\text{BOTTOM}} \begin{Bmatrix} \text{data-name-9} \\ \text{integer-8} \end{Bmatrix} \right] \right]$$

Technical Notes

1. The logical page size is the sum of the values referenced by each phrase except the FOOTING phrase. (There is no necessary relationship between the size of the logical page and the size of a physical page.) If the LINES AT TOP or LINES AT BOTTOM phrases are not specified, the values for these functions are zero.

2. Data-name-1, data-name-2, data-name-3 and data-name-4 must reference elementary unsigned numeric integer data items. The value of integer-1 must be greater than zero; the value of integer-2 must not be greater than integer-1; the value of integer-3 and integer-4 may be zero.

3. The number of lines on the logical page is equal to the value of integer-1 or the data item referenced by data-name-1. The page body is that part of the logical page in which lines can be written and/or spaced.

4. The line number within the page body at which the footing area begins is equal to the value of integer-2 or the data item referenced by data-name-2. The value must not be greater than the value of integer-1, or the data item referenced by data-name-1. The footing area is the area of the logical page between the line represented by the value of integer-2 (or the data item referenced by data-name-2) and the line represented by the value of integer-1 (or the data item referenced by data-name-1) inclusive.

5. The number of lines that constitute the top margin on the logical page is equal to the value of integer-3 or the data item referenced by data-name-3.

LINAGE (Cont.)

6. The number of lines that constitute the bottom margin on the logical page is equal to the value of integer-4 or the data item referenced by data-name-4.

7. When an OPEN statement with the OUTPUT option is executed, all of the data-names or integers you have specified will refer to the areas and positions of the first logical page. When a WRITE statement with the ADVANCING PAGE options is executed, or when a page overflow condition occurs, the data-names or integers you have specified will refer to the next logical page.

8. The presence of a LINAGE clause in the FD entry for a file causes the compiler to generate a LINAGE-COUNTER. The value in the LINAGE-COUNTER at any given time represents the line number at which the device is positioned within the current page body. The rules governing the LINAGE-COUNTER are as follows:

   a. The compiler supplies a separate LINAGE-COUNTER for each file described in the File Section whose file description entry contains a LINAGE clause.

   b. You may reference LINAGE-COUNTER, but you may not modify it, with Procedure Division statements. Since more than one LINAGE-COUNTER may exist in a program, you must qualify LINAGE-COUNTER by file-name when necessary.

   c. During the execution of a WRITE statement, LINAGE-COUNTER is automatically modified according to how and whether you have specified the ADVANCING clause, as follows:

      1) When you specify the ADVANCING PAGE phrase of the WRITE statement, the LINAGE-COUNTER is automatically reset to one (1).

      2) When you specify the ADVANCING identifier-2 or integer phrase of the WRITE statement, the LINAGE-COUNTER is incremented by integer or the value of the data item referenced by identifier-2.

      3) When you omit the ADVANCING phrase of the WRITE statement, the LINAGE-COUNTER is incremented by the value one (1).

      4) For each of the succeeding logical pages, the value of LINAGE-COUNTER is automatically reset to one (1) when the device is repositioned to the first line that can be written on.

   d. When an OPEN statement is executed, the LINAGE-COUNTER associated with that file is initialized to one (1).

9. Each logical page is contiguous to the next, with no additional spacing provided.

4.9.32 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

```
[ LINE NUMBER IS   { integer-1         } ]
[                  { PLUS  integer-2   } ]
[                  { NEXT PAGE         } ]
```

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 will be done, but the LINE NUMBER clause will cause 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 may 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.

LINE NUMBER (Cont.)

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 will only be 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 will then space 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

4.9.33 NEXT GROUP

Function

The NEXT GROUP clause specifies the spacing condition following the last line of the report group.

General Format

$$\underline{\text{NEXT GROUP}}\ \text{IS}\ \left\{\begin{array}{l}\text{integer-1}\\ \underline{\text{PLUS}}\ \text{integer-2}\\ \underline{\text{NEXT PAGE}}\end{array}\right\}$$

Technical Notes

1. The NEXT GROUP clause may 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 indicates a line number to which the LINE-COUNTER is set after the group is presented.
4. PLUS integer-2 indicates 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.

4.9.34 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

$$\underline{\text{RESET}}\ \text{ON}\ \left\{ \begin{array}{l} \text{identifier-1} \\ \underline{\text{FINAL}} \end{array} \right\}$$

Technical Notes

1. Identifier must be one of the identifiers associated with the CONTROL clause in the RD entry.
2. The RESET clause may be used only in conjunction with a SUM clause at a CONTROL FOOTING elementary level.
3. Identifier 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.

4.9.35 SOURCE

Function

The SOURCE clause indicates the source of the data for a report item.

General Format

<u>SOURCE</u> IS identifier

Technical Notes

1. The SOURCE clause can only be given at the elementary level.
2. Identifier must reference an item that appears in the File or Working-Storage Section.
3. The identifier cannot be subscripted or indexed.
4. When the report group is presented, the contents of this report item are replaced by the contents of identifier.

4.9.36 SUM

Function

The SUM clause indicates the items to be summed to produce the source of data for a report item.

General Format

<u>SUM</u> identifier-1 [identifier-2] ... [<u>UPON</u> data-name-1]

Technical Notes

1. A SUM clause may 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, Section 5.9.16.)

8. The identifiers cannot be subscripted or indexed.

4.9.37 TYPE

Function

The TYPE clause specifies the particular type of report group that is described by this entry and indicates the time 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                                               ⎭
```

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 may 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 may 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 may 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 may 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 may 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 may 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.

GENERAL FORMAT FOR DATA DIVISION

```
DATA DIVISION.

[FILE SECTION.

[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 }

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

     [DATE-WRITTEN IS { data-name-2 }] [USER-NUMBER IS { data-name-3 }]]
                      { literal-2   }]                 { literal-3   }]]

     [DATA { RECORD IS   }  data-name-4  [data-name-5]  ...]
           { RECORDS ARE }

     [LINAGE IS { data-name-6 }  LINES  [WITH FOOTING AT { data-name-7 }]
                { integer-5   }                          { integer-6   }]

        [LINES AT TOP { data-name-8 }] [LINES AT BOTTOM { data-name-9 }]]
                      { integer-7   }]                  { integer-8   }]]

     [CODE-SET IS alphabet-name]

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

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

GENERAL FORMAT FOR DATA DIVISION

```
                                                      ┐
 ┌              ⎧ 200 ⎫┐┌                      ┐
 │              ⎪ 556 ⎪││              ⎧ ODD  ⎫│
 │ DENSITY IS   ⎨ 800 ⎬││ PARITY IS    ⎩ EVEN ⎭│
 │              ⎩1600 ⎭││                      │
 └                     ┘└                      ┘
                                                      ┘

[SD  file-name
    [RECORD CONTAINS  [integer-1 TO]  integer-2 CHARACTERS]

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

[{record-description-entry}  ...]  ...]

[WORKING-STORAGE SECTION.

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

[LINKAGE SECTION.

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

[COMMUNICATION SECTION.

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

GENERAL FORMAT FOR DATA DIVISION

```
REPORT SECTION.

RD  report-name

     [ CODE  mnemonic-name ]

     ┌                    ┌                                            ┐ ┐
     │ ┌ CONTROL  IS  ┐  │ FINAL                                       │ │
     │ │              │  │ identifier-1  [ identifier-2] ...           │ │
     │ └ CONTROLS ARE ┘  │ FINAL identifier-1  [ identifier-2]  ...    │ │
     └                    └                                            ┘ ┘

     ┌
     │ PAGE  { LIMIT  IS  }  integer-1  { LINE  }
     │       { LIMITS  ARE}             { LINES }
     └

          [ HEADING  integer-2 ] [ FIRST  DETAIL  integer-3 ]

          [ LAST DETAIL    integer-4 ] [ FOOTING  integer-5 ] ]  .

{record-description-entry}     ...] ...]
```

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

FORMAT 1:

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

   [REDEFINES data-name-2]

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

   [               { COMPUTATIONAL   } ]
   [               { COMP            } ]
   [               { COMPUTATIONAL-1 } ]
   [               { COMP-1          } ]
   [               { COMPUTATIONAL-3 } ]
   [               { COMP-3          } ]
   [    USAGE IS   { DISPLAY         } ]
   [               { DISPLAY-6       } ]
   [               { DISPLAY-7       } ]
   [               { DISPLAY-9       } ]
   [               { INDEX           } ]
   [               { DATABASE-KEY    } ]
   [               { DBKEY           } ]

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

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

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

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

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

  [ { JUSTIFIED } { RIGHT } ]
  [ { JUST      } { LEFT  } ]

[BLANK WHEN ZERO]

[VALUE IS literal]   .
```

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

<u>FORMAT 2</u>:

$$66\ \text{data-name-1}\ \underline{\text{RENAMES}}\ \text{data-name-2}\ \left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\ \text{data-name-3}\right]\ .$$

<u>FORMAT 3</u>:

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

$$\left[\text{literal-3}\ \left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\ \text{literal-4}\right]\right]\ \ldots\ .$$

GENERAL FORMAT FOR REPORT GROUP DESCRIPTION ENTRY

Format 1

```
01  [data-name-1]

 ┌                       ┌                  ┐ ┐
 │ LINE  NUMBER  IS      │ integer-1        │ │
 │ ────                  │ PLUS  integer-2  │ │
 │                       │ ────             │ │
 │                       │ NEXT  PAGE       │ │
 └                       └ ──────────       ┘ ┘

 ┌                       ┌                  ┐ ┐
 │ NEXT  GROUP  IS       │ integer-3        │ │
 │ ───────────           │ PLUS  integer-4  │ │
 │                       │ ────             │ │
 │                       │ NEXT  PAGE       │ │
 └                       └ ──────────       ┘ ┘

               ┌ REPORT  HEADING                                    ┐
               │ ───────────────                                    │
               │ RH                                                 │
               │ ──                                                 │
               │ PAGE  HEADING                                      │
               │ ─────────────                                      │
               │ PH  ┌ CONTROL  HEADING ┐ ┌ identifier-1 ┐          │
               │ ──  │ ──────────────── │ │ FINAL        │          │
               │     └ CH               ┘ └ ─────        ┘          │
               │       ──                                           │
  TYPE  IS     │ DETAIL                                             │
  ────         │ ──────                                             │
               │ DE  ┌ CONTROL  FOOTING ┐ ┌ identifier-2 ┐          │
               │ ──  │ ──────────────── │ │ FINAL        │          │
               │     └ CF               ┘ └ ─────        ┘          │
               │       ──                                           │
               │ PAGE  FOOTING                                      │
               │ ─────────────                                      │
               │ PF                                                 │
               │ ──                                                 │
               │ REPORT  FOOTING                                    │
               │ ───────────────                                    │
               └ RF                                                 ┘
                 ──

 ┌                     ┌ DISPLAY   ┐ ┐
 │ [ USAGE  IS ]       │ ───────   │ │
 │   ─────             │ DISPLAY-6 │ │  .
 │                     │ ───────── │ │  ─
 │                     │ DISPLAY-7 │ │
 │                     │ ───────── │ │
 │                     │ DISPLAY-9 │ │
 └                     └ ───────── ┘ ┘
```

GENERAL FORMAT FOR REPORT GROUP DESCRIPTION ENTRY

Format 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 } ]
```

CHAPTER 5

THE PROCEDURE DIVISION

The Procedure Division specifies the processing to be performed on the files and 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] ... ] ... } ...
```

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 may 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 the 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 via 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

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 INSPECT	ARITHMETIC	IMPERATIVE
ALTER CALL ENTER ENTRY EXIT PROGRAM GOBACK GO TO PERFORM STOP	SEQUENCE-CONTROL	IMPERATIVE
ACCEPT INSPECT MOVE SET STRING UNSTRING	DATA MOVEMENT	IMPERATIVE
CANCEL FREE INSPECT MERGE RELEASE RETAIN RETURN SEARCH SORT TRACE	MISCELLANEOUS	IMPERATIVE
GENERATE INITIATE TERMINATE	REPORT	IMPERATIVE
ACCEPT CLOSE DELETE DISPLAY OPEN READ REWRITE WRITE	I-O	IMPERATIVE
IF	CONDITIONAL	CONDITIONAL
COPY ENTER 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), to copy library entries (COPY), and to specify procedures for input-output errors (USE). Generally, compiler-directing sentences generate no object-program coding.

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 Section 1.3, Source Program Format) 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 and be followed by the word SECTION followed by a priority number, if desired, followed by a terminating period.

```
     section-name SECTION nn.
```

If the section-name is in the Declaratives portion, it may not have a priority number. A USE statement may 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. The method used to achieve this reduction is called segmentation.

Segmentation consists of dividing the Procedure Division sections into logically related groupings called segments. You can define 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 you wish 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 non-resident 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 non-resident 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 reloaded into memory.

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.

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 may 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 may begin with one of the following:

   (- + variable

   and may 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 path and the false path) of control depending upon the truth value of a test. Conditional expressions can be used in conditional (IF) statements and in PERFORM statements (formats 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)
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 may 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 may not be compared to a numeric operand. An alphanumeric operand may 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

$$\left\{\begin{array}{l}\text{identifier-1}\\\text{literal-1}\\\text{arithmetic-expression-1}\\\text{figurative-constant-1}\end{array}\right\}\ \text{relational-operator}\ \left\{\begin{array}{l}\text{identifier-2}\\\text{literal-2}\\\text{arithmetic-expression-2}\\\text{figurative-constant-2}\end{array}\right\}$$

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 nonnumeric, a true zero condition may be established by a nonnumeric test followed by a comparison with zero.

5.5.1.4 **Comparison of Nonnumeric Items** - For operands whose category is nonnumeric (or where one operand is numeric and the other is nonnumeric), 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 C). The size of an operand is the total number of characters in the operand. Blanks and tabs are not ignored when a nonnumeric item is compared to ZERO. The presence of either blanks, tabs, or both in the operand will cause the test result to be NOT EQUAL.

There are three cases to consider: operands of equal size, operands of unequal size, and operands with differing justification.

1. Operands of equal size - If the operands are of equal size, 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 size - If the operands are of unequal size, 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

$$\text{identifier IS } \left[\,\underline{\text{NOT}}\,\right] \left\{\begin{array}{l}\underline{\text{ALPHABETIC}}\\ \underline{\text{NUMERIC}}\end{array}\right\}$$

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 will result 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 nonnumeric 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 must have one of the four following representations: a leading graphic sign ("+" or "-"), a trailing graphic sign, a leading embedded 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, which causes leading and trailing blanks and tabs to be ignored, may be selected by a switch setting during system installation. 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 Condition** - The general format for a condition-name is

```
[NOT] 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 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

Format of a Sign Condition - The general format for a sign condition follows.

$$\left\{\begin{array}{l}\text{identifier}\\ \text{arithmetic-expression}\end{array}\right\} \text{ IS } \left[\underline{\text{NOT}}\right] \left\{\begin{array}{l}\underline{\text{POSITIVE}}\\ \underline{\text{NEGATIVE}}\\ \underline{\text{ZERO}}\end{array}\right\}$$

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 will make an item nonnumeric.

5.5.5 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.
NOT	Entire condition is true if the simple condition is false.

5.5.6 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 and/or OR. A compound-condition can contain any combination of types of conditional expressions (relational, class, condition-name, 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 may be used either to improve readability or to override the effects of 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 may 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.

   Figure 5-1 illustrates the effect of this statement and the order of evaluation.

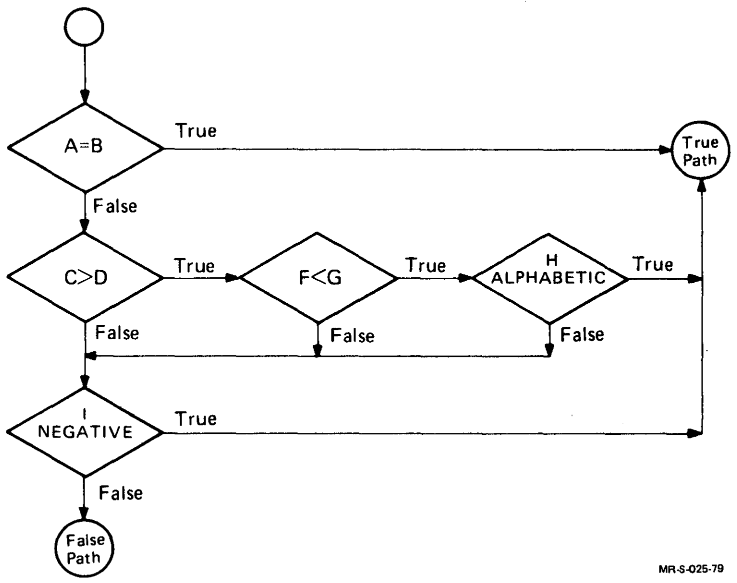


Figure 5-1 Order of Evaluation of a Conditional Expression

2. Using parentheses to override normal order of evaluation

   To illustrate this usage, a compound-conditional is shown in three forms in Figure 5-2, each accompanied by a flow diagram showing the result of each.

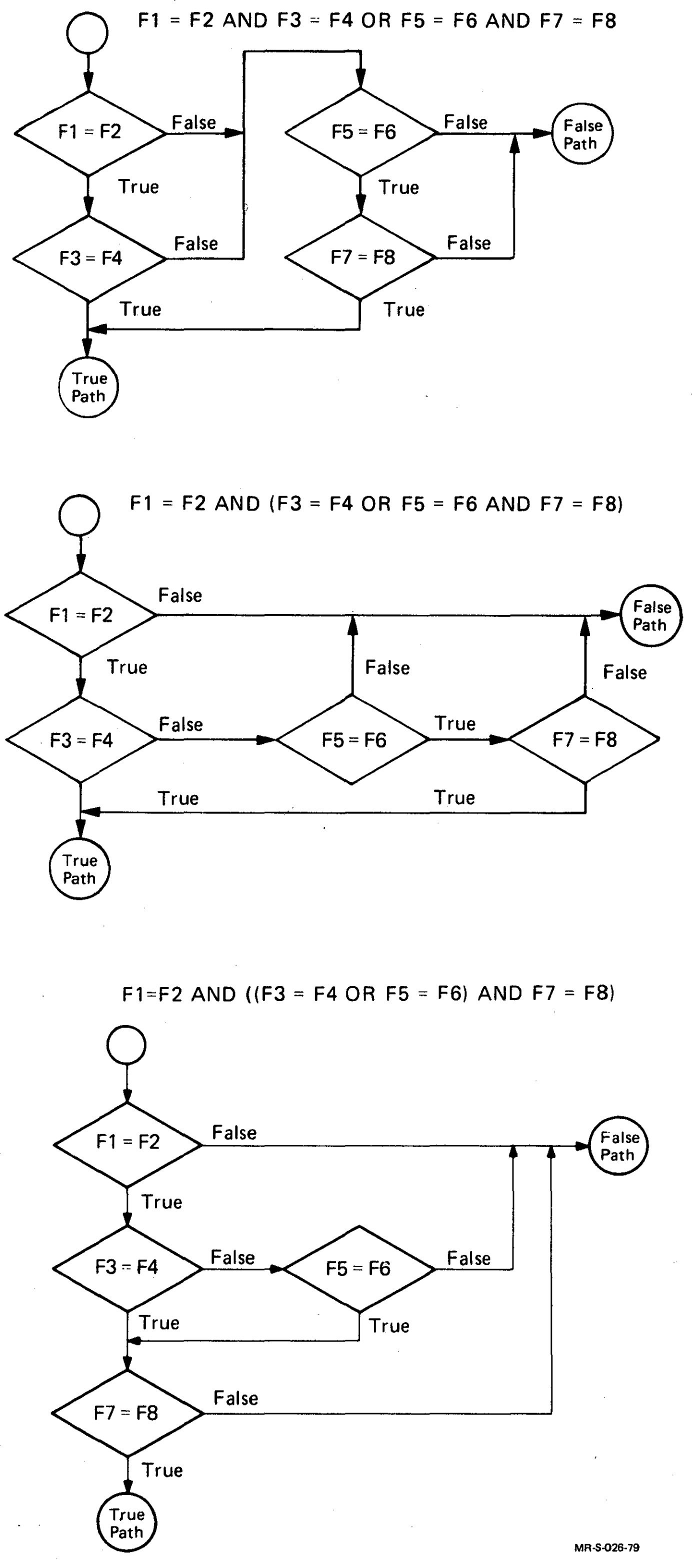

F1 = F2 AND F3 = F4 OR F5 = F6 AND F7 = F8
F1 = F2
F3 = F4
F5 = F6
F7 = F8
False
True
False Path
True Path
F1 = F2 AND (F3 = F4 OR F5 = F6 AND F7 = F8)
F1=F2 AND ((F3 = F4 OR F5 = F6) AND F7 = F8)
MR-S-026-79

Figure 5-2 Order of Evaluation of a Compound-conditional Expression

5.5.7 Combined and Negated Combined Conditions

A combined condition results from connecting conditions with one of the logical operators 'AND' or 'OR'. The general format of a combined condition follows:

$$\text{condition} \left\{ \left\{ \begin{array}{c} \underline{\text{AND}} \\ \underline{\text{OR}} \end{array} \right\} \quad \text{condition} \right\} \ldots$$

"Condition" may be one of the following:

1. A simple condition
2. A negated simple condition
3. A combined condition
4. A negated combined condition: that is, the 'NOT' logical operator followed by a combined condition enclosed within parentheses.
5. Combinations of the above, specified according to the rules summarized in Table 5-3 Combinations of Conditions, Logical Operators, and Parentheses.

Although parentheses need never be used when either 'AND' or 'OR' (but not both) is used exclusively in a combined condition, parentheses may be used to effect a final truth value when a mixture of 'AND', 'OR' and 'NOT' is used. (See Table 5-3 Combinations of Conditions, Logical Operators, and Parentheses.)

Table 5-3 indicates the ways in which conditions and logical operators may be combined and parenthesized. There must be a one-to-one correspondence between left and right parentheses so that each left parenthesis occurs to the left of its corresponding right parenthesis.

Table 5-3
Conditions, Logical Operators, and Parentheses Combinations

			In a left-to-right sequence of elements:	
Given the following element	Location in conditional expression		Element, when not first, may be immediately preceded by only:	Element, when not last, may be immediately followed by only:
	First	Last		
simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	simple-condition,)	simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	simple-condition, (
(	Yes	No	OR, NOT, AND, (	simple-condition, NOT, (
)	No	Yes	simple-condition,)	OR, AND,)

Thus, the element pair 'OR NOT' is permissible while the pair 'NOT OR' is not permissible; 'NOT' is permissible while 'NOT NOT' is not permissible.

5.5.8 Abbreviated Combined Relation Conditions

Simple or negated simple relation conditions can be combined with logical connectives in a consecutive sequence. When a succeeding relation condition contains a subject or subject and relational operator that is common with the preceding relation condition, and no parentheses are used within such a consecutive sequence, then any relation condition except the first may be abbreviated by one of the following:

1. The omission of the subject of the relation condition
2. The omission of the subject and relational operator of the relation condition

The format for an abbreviated combined relation condition follows:

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

Within a sequence of relation conditions both of the above forms of abbreviation may be used. The effect of using such abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. The result of such implied insertion must comply with the rules of Table 5-3, Combinations of Conditions, Logical Operators, and Parentheses. This insertion of an omitted subject and/or relational operator terminates once a complete simple condition is encountered within a complex condition.

The interpretation applied to the use of the word 'NOT' in an abbreviated combined relation condition is as follows:

1. If the word immediately following 'NOT' is 'GREATER', '>', 'LESS', '<', 'EQUAL', or '=', then the 'NOT' participates as part of the relational operator; otherwise
2. The 'NOT' is interpreted as a logical operator and, therefore, the implied insertion of subject or relational operator results in a negated relation condition.

Some examples of abbreviated combined and negated combined relation conditions and expanded equivalents follow.

Abbreviated Combined Relation Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a > b) AND (a NOT < c)) OR (a NOT < d)
a NOT EQUAL b OR c	(a NOT EQUAL b) OR (a NOT EQUAL c)
NOT a = b OR c	(NOT (a = b)) OR (a = c)

NOT (a GREATER b OR < c)	NOT ((a GREATER b) OR (a < c))
NOT (a NOT > b AND c AND NOT d)	NOT (((a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d))))

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 SIZE ERROR option. These two options are described here to avoid the necessity of including their descriptions with each of the arithmetic verbs.

5.6.1 The ROUNDED Option

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

5.6.2 The 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 SIZE ERROR option is specified.

If the 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 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 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. A data item in d(1) and a data item in d(2) have the same data-name and the same qualification up to, but not including, d(1) and 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) may 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) may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

See the sections ADD, MOVE, and SUBTRACT 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, and 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.

If any nonnumeric 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	need no conversion.
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 they are found in the high-order or 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 high-order or 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, stopping 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 will be 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

The ACCEPT statement causes low-volume data to be read from the user's terminal.

General Format

$\underline{\text{ACCEPT}}$ identifier-1 identifier-2 ... $\left[\underline{\text{FROM}}\text{ mnemonic-name}\right]$

$\underline{\text{ACCEPT}}$ identifier FROM $\left\{\begin{matrix}\underline{\text{DATE}}\\ \underline{\text{DAY}}\\ \underline{\text{TIME}}\end{matrix}\right\}$

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 will read a maximum of 1023 characters into each identifier.

6. The ACCEPT statement places characters into an alphanumeric item from left to right, filling any remaining characters of the item with blanks up to a maximum of 1023 positions.

7. If an alphanumeric item is longer than 1023 characters, only the leftmost 1023 characters are replaced or blanked; the remaining characters of the item are not altered.

5.9.2 ADD

Function

The ADD statement computes the sum of two or more numeric operands and stores the result.

General Format

```
ADD { identifier-1 }  [ identifier-2 ] ...  TO identifier-m [ROUNDED]
    { literal-1    }  [ literal-2    ]

  [ identifier-n [ROUNDED] ] ...  [ON SIZE ERROR imperative-statement]


ADD { identifier-1 }  { identifier-2 }  [ identifier-3 ] ...
    { literal-1    }  { literal-2    }  [ literal-3    ]

    GIVING identifier-m [ROUNDED]  [ identifier-n [ROUNDED] ] ...

    [ON SIZE ERROR imperative-statement]


ADD { CORRESPONDING }  identifier-1 TO identifier-2 [ROUNDED]
    { CORR          }

  [ON SIZE ERROR imperative-statement]
```

Technical Notes

1. Each ADD statement must contain at least two operands (that is, an addend and an augend). In formats 1 and 2, each identifier must refer to an elementary numeric item, except that identifiers appearing to the right of the word GIVING may refer to numeric-edited items. In format 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 (that is, the data item resulting from the superimposition of all operands aligned by decimal point) must not contain more than 19 decimal digits for the standard 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. (See Section 1.1 for a definition of the BIS-compiler.)

ADD (Cont.)

3. Format 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. Format 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. Format 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 SIZE ERROR options are described in Section 5.6, Common Options Associated with Arithmetic Verbs.

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-4 ] ...
```

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 nothing but 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 may not reference a procedure name within the DECLARATIVES; conversely, an ALTER statement within the DECLARATIVES may 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 will be lost when segments are overlaid.

CALL

5.9.4 CALL

Function

The CALL statement is used to transfer control to a subprogram.

General Format

```
      ( identifier-1 )  [ USING data-name-1   [data-name-2] ... ]
CALL  { program-name }
      ( entry-name   )

   [ON OVERFLOW imperative-statement]
```

Technical Notes

1. Program-name is a one to six character name (PROGRAM-ID) of the subprogram to be called. Entry-name is a one to six 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 will be 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 may 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 may be referenced (or whose subordinate parts may 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.

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 TOs, 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 or overlaid. If the subprogram has been cancelled or overlaid, its state is that of a fresh copy.

8. The CALL identifier clause works only when the following conditions are met:

   a. There is only one subprogram per overlay.

   b. Each subprogram has only one entry point.

   c. The overlay name is the same as the subprogram name.

9. Refer to the COBOL-74 Usage Material, Part 3 of this manual, for more information on subprograms.

5.9.5 CANCEL

Function

The CANCEL statement releases the memory areas occupied by the programs named in the clause.

General Format

$$\underline{\text{CANCEL}} \begin{Bmatrix} \text{identifier-1} \\ \text{subprogram-1} \end{Bmatrix} \begin{bmatrix} \text{identifier-2} \\ \text{subprogram-2} \end{bmatrix}$$

Technical Notes

1. The CANCEL statement can be used either to reload a segment of a segmented COBOL program or to cancel a subprogram that has been loaded into an overlay link by LINK. (Refer to the COBOL-74 Usage Material, Part 3 of this manual, for information on specifying LINK overlays and on subprograms.) Note 2 describes the first case while the remaining notes describe the second.

2. When you cancel a segment of a program you cause the object-time system to read your .EXE file and copy an initialized version of the segment into memory.

3. After a subprogram has been cancelled, a subsequent call to the subprogram will cause a freshly initialized copy to be brought into memory.

4. Cancellation of a subprogram causes the entire link in which it resides and all lower-level links to be cancelled.

5. 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 will be ignored and a warning message issued at runtime.

6. A subprogram cannot cancel itself or any subprogram that resides in an overlay link with it. An attempt to do either will result in the CANCEL statement being ignored and a warning message issued at runtime.

7. 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 reach an undefined location.

5.9.6 CLOSE

Function

The CLOSE statement terminates the processing of input and output files, reels, or units.

General Format

```
                                            ┌                              ┐
                      ┌ { REEL } ┐ │ { WITH      { NO REWIND }   } │
                      │ { UNIT } │ │ {           { LOCK      }   } │
CLOSE file-name-1     └          ┘ │ {           { DELETE    }   } │
                                   │ { FOR         REMOVAL       } │
                                   └                               ┘

                                   ┌                               ┐
                      ┌ { REEL } ┐ │ { WITH      { NO REWIND }   } │
    file-name-2       │ { UNIT } │ │ {           { LOCK      }   } │
                      └          ┘ │ {           { DELETE    }   } │
                                   │ { FOR         REMOVAL       } │
                                   └                               ┘

CLOSE file-name-1 [WITH LOCK] [file-name-2 [WITH LOCK]] ...
```

Technical Notes

1. Each filename must appear as the subject of an FD entry in the File Section of the Data Division.

2. The DELETE option applies only to disk and DECtape files. If this option is included, the file will be deleted from the device.

3. The REEL, UNIT, and NO REWIND options apply only to magnetic tape files; UNIT is synonymous with REEL.

4. The FOR REMOVAL option unloads magnetic tape. The file cannot be re-opened without intervention by the operator.

5. 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.

CLOSE (Cont.)

b. SINGLE REEL	A file that is entirely contained on one reel or unit.
c. MULTI-REEL	A file that may 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 5-4. 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.

Codes Used in Table 5-4

A Any subsequent reels of this file will not be processed.

B The current reel is not rewound.

C Standard CLOSE File Procedure is followed:

INPUT and I-O Files

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.

D The current reel is rewound and unloaded.

E Any attempt to subsequently OPEN this file will result in an error message being typed and the run terminated.

F Standard CLOSE REEL Procedure is followed:

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 is performed for the new reel.

OUTPUT and I-O Files

1. The standard ending reel label procedure is performed.

2. If the file is assigned to more than one device, the devices are swapped. A halt occurs to allow the operator to mount an available reel.

3. The standard beginning reel label procedure is performed.

G The tape is rewound.

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 will remain intact (that is, the original file will not be superseded nor deleted).

X Illegal. This is an illegal combination of a CLOSE option and a file type.

6. If a file is OPENed but not CLOSEd before the STOP RUN statement is executed, the file will be automatically CLOSEd. Any records still retained by a RETAIN statement will automatically be freed by a CLOSE statement.

7. 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.

8. 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.

CLOSE (Cont.)

Table 5-4
CLOSE Options and File Types

CLOSE Options	File Type		
	NON-REEL	SINGLE REEL/UNIT	MULTI-REEL
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 FOR REMOVAL	X	X	F,D,G
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

```
                                  ┌┌                          ┐            ┐
                                  ││ identifier-2             │            │
COMPUTE identifier-1 [ROUNDED]    ││ literal                  │ [ROUNDED]  │  ...
                                  ││ arithmetic-expression    │            │
                                  └└                          ┘            ┘
     ┌ IS EQUAL TO ┐
     │ EQUALS      │  arithmetic-expression   [ON SIZE ERROR imperative-statement]
     └ =           ┘
```

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. If the composite operand exceeds 19 decimal digits, the composite is converted to COMP-1 format. This will lead, however, to a loss of precision.

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 in Section 5.4, Arithmetic Expressions.

5. The ROUNDED and SIZE ERROR options are described in Section 5.6, Common Options Associated with the Arithmetic Verbs.

5.9.8 DELETE

Function

The DELETE statement removes a specified record from a file whose organization is RELATIVE or INDEXED.

General Format

```
DELETE file-name RECORD  [INVALID KEY imperative-statement]
```

Technical Notes

1. Record-name must be a record associated with a file whose organization is RELATIVE or INDEXED.

2. When the DELETE statement is executed, the object-time system removes from the file the record which has a key equal in value to the RELATIVE KEY (for relative files) or the RECORD KEY (for indexed files). 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.

4. The INVALID KEY clause must not be specified for a DELETE statement that references a file that is in sequential-access mode. It must be specified for a DELETE statement that references a file that is not in sequential-access mode, and for which no USE procedure is specified.

5. For files in the sequential-access mode, the last input-output statement executed for file-name prior to the execution of the DELETE statement must have been a successfully executed READ statement. The OTS logically removes from the file the record that was accessed by that READ statement.

6. The execution of a DELETE statement does not affect the current record pointer or the contents of the record area associated with file-name. The execution of the DELETE statement causes updating of the value of any specified FILE STATUS data item associated with file-name.

5.9.9 DISPLAY

Function

The DISPLAY statement causes low-volume data to be written to the user's terminal.

General Format

$$\underline{\text{DISPLAY}} \left\{ \begin{matrix} \text{identifier-1} \\ \text{literal-1} \end{matrix} \right\} \left[\left\{ \begin{matrix} \text{identifier-2} \\ \text{literal-2} \end{matrix} \right\} \right] \ldots \left[\underline{\text{UPON}} \text{ mnemonic-name} \right] \left[\underline{\text{WITH}}\ \underline{\text{NO}}\ \underline{\text{ADVANCING}} \right]$$

Technical Notes

1. The contents of each operand are written on the user's terminal in the order listed.

2. Each of the literals can be numeric or nonnumeric, 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. If you do not specify the WITH NO ADVANCING clause, the terminal will advance to the next line after printing the text of the DISPLAY statement.

DIVIDE

5.9.10 DIVIDE

Function

The DIVIDE statement divides one numeric item into others and sets the value of specified data item(s) equal to the quotient and the remainder.

General Format

```
DIVIDE  { identifier-1 }  INTO identifier-2  [ROUNDED]
        { literal-1    }

  [identifier-3 [ROUNDED]] ...  [ON SIZE ERROR imperative-statement]

DIVIDE  { identifier-1 }  INTO  { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

  [identifier-4 [ROUNDED]]  ...  [ON SIZE ERROR imperative-statement]

DIVIDE  { identifier-1 }  BY    { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

  [identifier-4 [ROUNDED]]  ...  [ON SIZE ERROR imperative-statement]

DIVIDE  { identifier-1 }  INTO  { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

 REMAINDER identifier-4  [ON SIZE ERROR imperative-statement]

DIVIDE  { identifier-1 }  BY    { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

 REMAINDER identifier-4  [ON SIZE ERROR imperative-statement]
```

Technical Notes

1. In all formats which include the INTO keyword, identifier-1 is the divisor and identifier-2 is the dividend. In formats which include the BY keyword, identifier-1 is the dividend and identifier-2 the divisor. In formats 1 and 2, the resulting quotient replaces the value of identifier-2. In formats 3 and 4, the resulting quotient replaces the value of identifier-3 and any data items which follow 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 may be an elementary numeric or numeric-edited item. Each literal-1 or literal-2 must be a numeric literal. Identifier-4 may be an elementary numeric or numeric-edited item.

3. The ROUNDED and SIZE ERROR options are described in Section 5.6, 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 (that is, 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. (See Section 1.1 for a definition of the BIS-compiler.)

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, the object-time system follows the procedure described in Section 5.6, 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

$$\underline{\text{ENTER}} \left\{ \begin{matrix} \underline{\text{MACRO}} \\ \underline{\text{FORTRAN}} \\ \underline{\text{COBOL}} \end{matrix} \right\} \left[\underline{\text{USING}} \left\{ \begin{matrix} \text{identifier-1} \\ \text{literal-1} \\ \text{procedure-name-1} \end{matrix} \right\} \left[\left\{ \begin{matrix} \text{identifier-2} \\ \text{literal-2} \\ \text{procedure-name-2} \end{matrix} \right\} \right] \right] \ldots$$

Technical Notes

1. MACRO refers to MACRO-10 or MACRO-20 assembly language and FORTRAN to the TOPS-10 or the TOPS-20 FORTRAN language.
2. The program-name can be enclosed in quotation marks.
3. The ENTER statement generates a subroutine call and specifies the address where the items associated with the USING clause are located. (Refer to the COBOL-74 Usage Material, Part 3 of this manual, for more information on the ENTER statement.)
4. ENTER COBOL is equivalent to CALL.

5.9.12 ENTRY

Function

The ENTRY statement establishes an entry point in a subprogram.

General Format

```
ENTRY entry-name [ USING identifier-1 [identifier-2] ... ] .
```

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 one to six 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 may 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 runtime, ENTRY statements are ignored unless there are specific calls to them.
8. Refer to the COBOL-74 Usage Material, Part 3 of this manual, for more information on subprograms.

EXIT

5.9.13 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. EXIT.

Technical Notes

1. EXIT must be the only sentence in the paragraph.

2. The EXIT statement may be used at the end of a section in the Declaratives, or to provide an end point for a series of paragraphs that are performed. When you use EXIT at the end of the range of a PERFORM or USE, you can provide a variety of exits from the performed procedure 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.

5.9.14 EXIT PROGRAM

Function

The EXIT PROGRAM statement is used to return control from a subprogram to its calling program.

General Format

```
EXIT [PROGRAM] .
```

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 the COBOL-74 Usage Material, Part 3 of this manual, for more information on subprograms.

FREE

5.9.15 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] ... ] .
```

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, Section 5.9.29, 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.

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, relative, 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 organization of the file. Each organization is described separately below.

    a. Sequential Files

       If the KEY phrase is specified, the value of the key 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 Files

       If the KEY phrase is specified, the value of the key refers to the record with that value in the RETAIN statement. For example, 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.

FREE (Cont.)

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.

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 RECORD KEY of the file. If the RECORD 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 File

```
RETAIN PART KEY 0 FOR ANY VERB.
READ PART, INVALID KEY GO TO ERR.
WRITE PARTREC.
FREE PARK KEY 0.
```

Indexed-Sequential File

```
MOVE "B" TO RECORD-KEY.
RETAIN LETTERS FOR READ.
FREE LETTERS.
```

5.9.16 GENERATE

Function

The GENERATE statement causes the Report-Writer to execute all automatic report operations, and, if required, to produce one or more report groups.

General Format

$$\underline{\text{GENERATE}} \quad \left\{ \begin{array}{l} \text{data-name} \\ \text{report-name} \end{array} \right\}$$

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. It steps and tests the LINE-COUNTER and/or PAGE-COUNTER to produce, if necessary, any PAGE FOOTING and PAGE HEADING report groups.

   b. It recognizes any specified control breaks to produce appropriate CONTROL FOOTING and CONTROL HEADING report groups, and resets appropriate summation counters.

   c. It accumulates into the summation counters all specified identifiers.

   d. It executes any routines defined by a USE statement.

   e. In detailed reporting, it 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, in 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.

6. A GENERATE statement for a particular report may 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 may not be executed until an intervening INITIATE statement is executed for the report.

5.9.17 GO TO

Function

The GO TO statement causes control to be transferred from one part of the Procedure Division to another.

General Format

```
GO TO [procedure-name-1]

GO TO procedure-name-1 [procedure-name-2] ...   procedure-name-n

   DEPENDING ON identifier
```

Technical Notes

1. Each procedure-name is the name of a paragraph or section in the Procedure Division of the program.

2. Format 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, format 1 must appear as the first sentence in a paragraph.

   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.

3. Format 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 may 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.

5.9.18 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 the COBOL-74 Usage Material, Part 3 of this manual, for more information on subprograms.

5.9.19 IF

Function

The IF statement causes a conditional expression to be evaluated and subsequent operations to be determined as a result of this evaluation.

General Format

$$\underline{\text{IF}}\ \text{condition}\ \left\{\begin{matrix}\text{statement-1}\\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}}\end{matrix}\right\}\ \left[\underline{\text{ELSE}}\quad \left\{\begin{matrix}\text{statement-2}\\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}}\end{matrix}\right\}\right]$$

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.

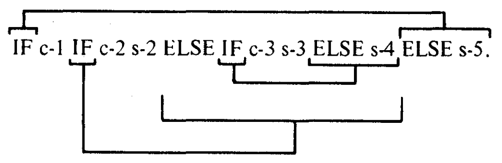
   If the conditional expression is false and either ELSE NEXT SENTENCE is given or the entire ELSE clause is omitted, control passes to the next sentence.

3. The length of compared data-items in the conditional expression of an IF statement is limited to 2047 characters.

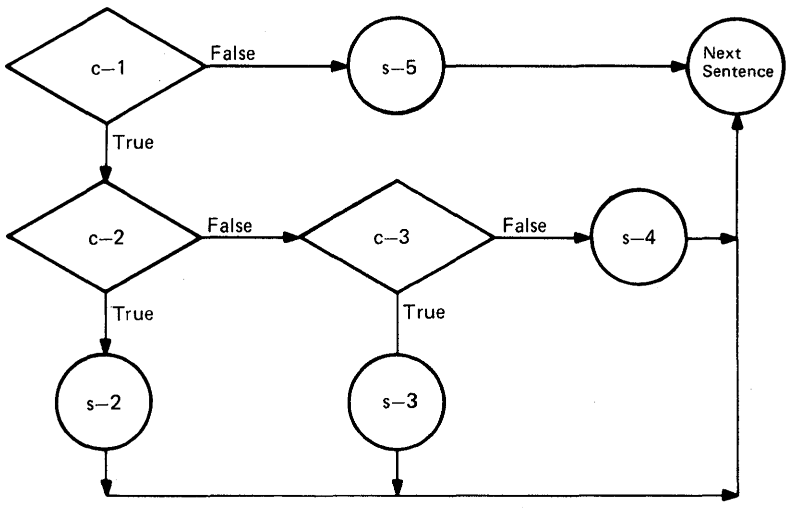
4. Statement-1, statement-2, statement-3, and statement-4 may include any statement or sequence of statements, including other IF statements. IF statements included within other IF statements are nested. Nested IF statements are paired IF and ELSE combinations and may continue up to 12 levels deep. Each 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-027-79

5.9.20 INITIATE

Function

The INITIATE statement is used to initialize all counters before a report is produced.

General Format

```
INITIATE report-name-1  [report-name-2]  ...
```

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 may 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 may not be executed until a TERMINATE statement for that report-name is executed.

5.9.21 INSPECT

Function

The INSPECT statement counts, replaces, or counts and replaces the number of occurrences of a given character or groups of characters in a data item.

General Format

```
INSPECT identifier-1 TALLYING

 {                  { {ALL       }   {identifier-3}} [{BEFORE}           {identifier-4}]}     }
 {identifier-2 FOR  { {LEADING   }   {literal-1   }} [{AFTER }  INITIAL  {literal-2   }]} ... } ...
 {                  { {CHARACTERS}                 }                                   }     }


INSPECT identifier-1 REPLACING

 {CHARACTERS BY {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]                                      }
 {              {literal-4   } [{AFTER }           {literal-5   }]                                      }
 {  { {ALL    } {identifier-5}    {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]}     }             }
 {  { {LEADING} {literal-3   } BY {literal-4   } [{AFTER }           {literal-5   }]} ... } ...         }
 {  { {FIRST  }                                                                    }     }             }

INSPECT identifier-1 TALLYING

 {                  { {ALL       }   identifier-3} [{BEFORE}           {identifier-4}]}     }
 {identifier-2 FOR  { {LEADING   }   literal-1   } [{AFTER }  INITIAL  {literal-2   }]} ... } ...
 {                  { {CHARACTERS}               }                                   }     }


REPLACING

 {CHARACTERS BY {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]                                      }
 {              {literal-4   } [{AFTER }           {literal-5   }]                                      }
 {  { {ALL    } {identifier-5}    {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]}     }             }
 {  { {LEADING} {literal-3   } BY {literal-4   } [{AFTER }           {literal-5   }]} ... } ...         }
 {  { {FIRST  }                                                                    }     }             }
```

Technical Notes

The following rules apply to Formats 1, 2 and 3:

1. Each literal must be nonnumeric and may be any figurative constant except ALL.

2. The usage of all identifiers must be DISPLAY, implicitly or explicitly. Identifier-1 must reference either a group item or any category of elementary item. Identifier-3... identifier-n must reference either an elementary, alphabetic, alphanumeric or numeric item.

The following rules apply to Format 1:

3. Identifier-2 must reference an elementary numeric data name. If either literal-1 or literal-2 is a figurative constant, it refers to an implicit one-character data item.

4. The contents of the data item referenced by identifier-2 is not initialized by the execution of the INSPECT statement.

5. The rules for tallying are as follows:

   a. If the ALL phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one (1) for each occurrence of literal-1 matched within the contents of the data item referenced by identifier-1.

   b. If the LEADING phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one (1) for each contiguous occurrence of literal-1 matched within the contents of the data item referenced by identifier-1, provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle in which literal-1 was eligible to participate.

   c. If the CHARACTERS phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one (1) for each character matched, within the contents of the data item referenced by identifier-1.

The following rules apply to Format 2:

6. The size of the data referenced by literal-4 or identifier-6 must be equal to the size of the data referenced by literal-3 or identifier-5. When a figurative constant is used as literal-4, the size of the figurative constant is equal to the size of literal-3 or the size of the data item referenced by identifier-5.

7. When the CHARACTERS phrase is used, literal-4, literal-5, or the size of the data item referenced by identifier-6 or identifier-7 must be one character in length.

8. When a figurative constant is used as literal-3, the data referenced by literal-4 or identifier-6 must be one character in length.

9. The required words ALL, LEADING and FIRST are adjectives that apply to each succeeding BY phrase until the next adjective appears.

10. The following rules for replacement are as follows:

    a. When the CHARACTERS phrase is specified, each character matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.

    b. When the adjective ALL is specified, each occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.

INSPECT (Cont.)

c. When the adjective LEADING is specified, each contiguous occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which literal-3 was eligible to participate.

d. When the adjective FIRST is specified, the leftmost occurrence of literal-3 matched within the contents of the data item referenced by identifier-1 is replaced by literal-4.

The following rules apply to Format 3:

11. Identifier-2 must reference an elementary numeric data item.

12. If either literal-1 or literal-2 is a figurative constant, the figurative constant refers to an implicit one-character data item.

13. The size of the data referenced by literal-4 or identifier-6 must be equal to the size of the data referenced by literal-3 or identifier-5. When a figurative constant is used as literal-4, the size of the figurative constant is equal to the size of literal-3 or the size of the data item referenced by identifier-5.

14. When the CHARACTERS phrase is used, literal-4, literal-5, or the size of the data item referenced by identifier-6 or identifier-7 must be one character in length.

15. When a figurative constant is used as literal-3, the data referenced by literal-4 or identifier-6 must be one character in length.

16. A Format 3 INSPECT statement is interpreted and executed as though two successive INSPECT statements specifying the same identifier-1 had been written with one statement being a Format 1 statement with TALLYING phrases identical to those specified in the Format 3 statement, and the other statement being a Format 2 statement with REPLACING phrases identical to those specified in the Format 3 statement. The general rules given for matching and counting apply to the Format 1 statement and the general rules given for matching and replacing apply to the Format 2 statement.

Examples

The field TXT-FLD contains "PSYCHOANALYSIS".

```
INSPECT TXT-FLD TALLYING COUNTER-1 FOR CHARACTERS BEFORE
INITIAL "A".
```

COUNTER-1 contains 6

```
INSPECT TXT-FLD REPLACING "A" BY "X" BEFORE INITIAL "N".
```

TXT-FLD ends with "PSYCHOXNALYSIS"

```
INSPECT TXT-FLD TALLYING COUNTER-1 FOR CHARACTERS AFTER
INITIAL "S", REPLACING ALL "S" BY "Z".
```

TXT-FLD ends with "PZYCHOANALYZIZ"
COUNTER-1 contains 12

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

```
MERGE [WITH SEQUENCE CHECK] file-name-1 ON {ASCENDING } KEY data-name-1 [data-name-2] ...
                                           {DESCENDING}

                                       [ON {ASCENDING } KEY data-name-3 [data-name-4] ...] ...
                                       [   {DESCENDING}                                  ]
 [COLLATING SEQUENCE IS alphabet-name]
  USING file-name-2  file-name-3  [file-name-4]  ...

 { OUTPUT PROCEDURE IS section-name-1 [{THROUGH} section-name-2] }
 {                                    [{THRU   }               ] }
 { GIVING file-name-5                                            }
```

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. Note that file-name-1 is actually a dummy file whose file description applies to all the files to be merged. However, file-name-2, file-name-3, file-name-4, and file-name-5 are real files. File-name-2, file-name-3, and file-name-4 are the files to be merged, and file-name-5 is the file into which the merged records will be written.

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 may be qualified but not subscripted.

5. MERGE statements may 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.

8. ALTER, GO, and PERFORM statements in the OUTPUT PROCEDURE may 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 will be 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.
    ```

    Refer to the SORT/MERGE User's Guide for more information on MERGE.

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

```
MOVE { identifier-1 }  TO identifier-2  [ identifier-3 ] ...
     { literal      }

MOVE { CORRESPONDING }  identifier-1 TO identifier-2
     { CORR          }
```

Technical Notes

1. CORR may be interchanged with CORRESPONDING.

2. 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.

3. In format 1, the data contained in identifier-1 or literal-1 is moved first to identifier-2, then to identifier-3, etc.

   In format 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 in Section 5.7, The CORRESPONDING Option.

4. 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.

   d. All other moves are legal.

5. The following rules apply to all 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.

6. Any move that is not an elementary move (that is, neither the sending or receiving items are 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 numeric data items to be multiplied and sets the values of data items equal to the results.

General Format

```
MULTIPLY {identifier-1} BY identifier-2 [ROUNDED]
         {literal-1   }

    [identifier-3 [ROUNDED]] ... [ON SIZE ERROR imperative-statement]

MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
         {literal-1   }    {literal-2   }

    [identifier-4 [ROUNDED]] ... [ON SIZE ERROR imperative-statement]
```

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 in format 2 may refer to either a numeric or a numeric-edited item. Each literal must be a numeric literal; the figurative constant ZERO is permitted.

2. Format 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. The same process happens again, with identifier-3 replacing identifier-2, then identifier-4 replacing identifier-3, until all multipliers have been used.

3. Format 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 is stored in identifier-3, identifier-4, and so on.

4. The ROUNDED and SIZE ERROR options are described in Section 5.6, Common Options Associated with Arithmetic Verbs.

5. Despite the possiblity of sequential multiplication taking place, there can never be more than two operands in use at one time. The total number of digits in both operands must not be more than 18 decimal digits for the standard 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. (See Section 1.1 for a definition of the BIS-compiler.)

5.9.25 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 [ REVERSED       ] [ file-name-2 [ REVERSED       ] ] ...
        ⎪ { OUTPUT }              [ WITH NO REWIND ] [             [ WITH NO REWIND ] ]
        ⎪
        ⎪                                   ⎡     ⎧ 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 ] filename-5 [ filename-6 ] ...
        ⎪
        ⎩ [ UNAVAILABLE statement-1 [ statement-2 ] ... ].
```

OPEN (Cont.)

Technical Notes

1. The OPEN statement must be executed for a file prior to the execution of any I/O verbs, such as READ, WRITE, DELETE, REWRITE, SEEK, or CLOSE.

2. A second OPEN statement for a file cannot be executed prior to the execution of a CLOSE statement for that file.

3. 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).

4. 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.

5. 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 the tape.

6. If a file has been described with LABEL RECORDS ARE STANDARD, standard label checking or label writing is performed. If a file has been described as LABEL RECORDS ARE OMITTED, no label checking or writinq is performed.

7. If an INPUT file is described as OPTIONAL (in the FILE-CONTROL paragraph), the object-time system will type the message

   IS file-name PRESENT?

   and wait for the operator to type YES or NO. If he types NO, the first READ statement for this file causes the imperative-statement at the AT END or INVALID KEY clause to be executed.

8. 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 to be executed. If the file does not exist when it is opened for INPUT-OUTPUT, an empty file is created.

9. 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).

10. 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 will be 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 will be denied access to the file until the first user closes it.

11. After the keyword FOR, you must give one or more verbs that you intend to execute while you have your file open. You can only execute those verbs that you have specified. Following the keywords ALLOWING OTHERS, you must give one or more verbs that you will 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.

12. 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.

13. 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.

14. 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 will be 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 will not cause the UNAVAILABLE path to be taken, but will cause an error to be returned. You can choose to ignore the error by using the FILE STATUS clause in the Environment Division (see Section 3.1.14, FILE STATUS).

15. Any valid COBOL statements (including OPEN) can be used in the UNAVAILABLE clause.

OPEN (Cont.)

16. If a user program wishes to open a file 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 a user 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.

17. A user program that violates its simultaneous update covenants is aborted. That is, if the program opens a file for READ and then issues a WRITE statement for that file, the program will be aborted.

18. 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, Section 5.9.29.

19. The EXTEND option may be specified only by users of TOPS-10.

20. When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file will add records to the file as though the file had been opened with the OUTPUT phrase.

21. When the EXTEND phrase is specified 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.

22. The REVERSED option may only be used on TU45 and TU70 tape drives. If you specify this option for a file, the file will be opened and the tape positioned at the end of the file. A READ statement will cause the final block of the file to be grabbed by the monitor. This can be slightly tricky because the record which is actually made available to you is the first record of the last block, which may not be the last record. For example, if your file is blocked 2, the record made available by the READ statement will be the next to last record in the file, not the last one.

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.
```

PERFORM

5.9.26 PERFORM

Function

The PERFORM statement is used to depart from the normal sequence of execution in order to execute one or more procedures and then return control to the normal sequence.

General Format

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                         [ { THRU    }                  ]

PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ] { identifier-1 } TIMES
                         [ { THRU    }                  ] { integer-1    }

PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ] UNTIL condition-1
                         [ { THRU    }                  ]

PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                         [ { THRU    }                  ]

   VARYING { identifier-2 } FROM { identifier-3 }
           { index-name-1 }      { index-name-2 }
                                 { literal-1    }

      BY { identifier-4 }  UNTIL condition-1
         { literal-3    }

  [ AFTER { identifier-5 }  FROM { identifier-6 }
  [       { index-name-3 }       { index-name-4 }
  [                              { literal-3    }

      BY { identifier-7 }  UNTIL  condition-2
         { literal-4    }

  [ AFTER { identifier-8 }  FROM  identifier-9
  [       { index-name-5 }        index-name-6
  [                               literal-5

      BY { identifier-10 }  UNTIL  condition-3 ] ]
         { literal-6     }
```

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 either a numeric literal or the figurative constant ZERO.

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 may occur between procedure-name-1 and procedure-name-2.

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 may terminate until all PERFORM statements that it has executed have terminated. A PERFORM statement may be executed which terminates at the same procedure-name as another active PERFORM.

6. Format 1 causes the PERFORM range to be executed once, followed by a return to the statement immediately following the PERFORM.

PERFORM(Cont.)

7. Format 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 may 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. Format 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 in Section 5.5, Conditional Expressions.

9. Format 4 is used to augment the value of one or more identifiers during the execution of a PERFORM statement. In format 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.

   The flow chart in Figure 5-3 illustrates the logic of the PERFORM cycle when two identifiers are varied.

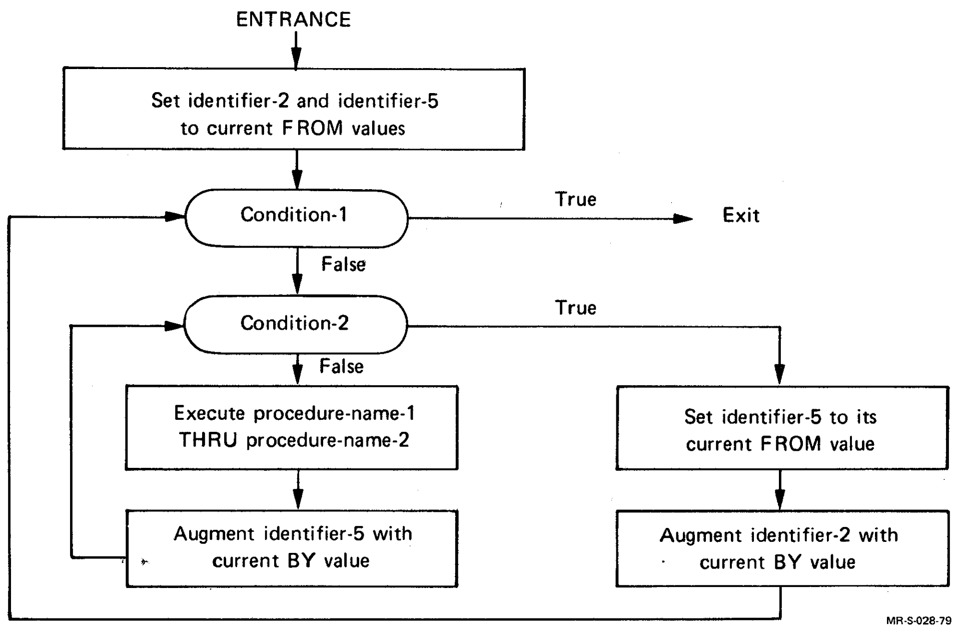

ENTRANCE
Set identifier-2 and identifier-5 to current FROM values
Condition-1
True
Exit
False
Condition-2
True
False
Execute procedure-name-1 THRU procedure-name-2
Set identifier-5 to its current FROM value
Augment identifier-5 with current BY value
Augment identifier-2 with current BY value
MR-S-028-79

Figure 5-3 PERFORM Cycle Logic - Two Variables

The flow chart in Figure 5-4 illustrates the logic of the PERFORM cycle when three identifiers are varied.

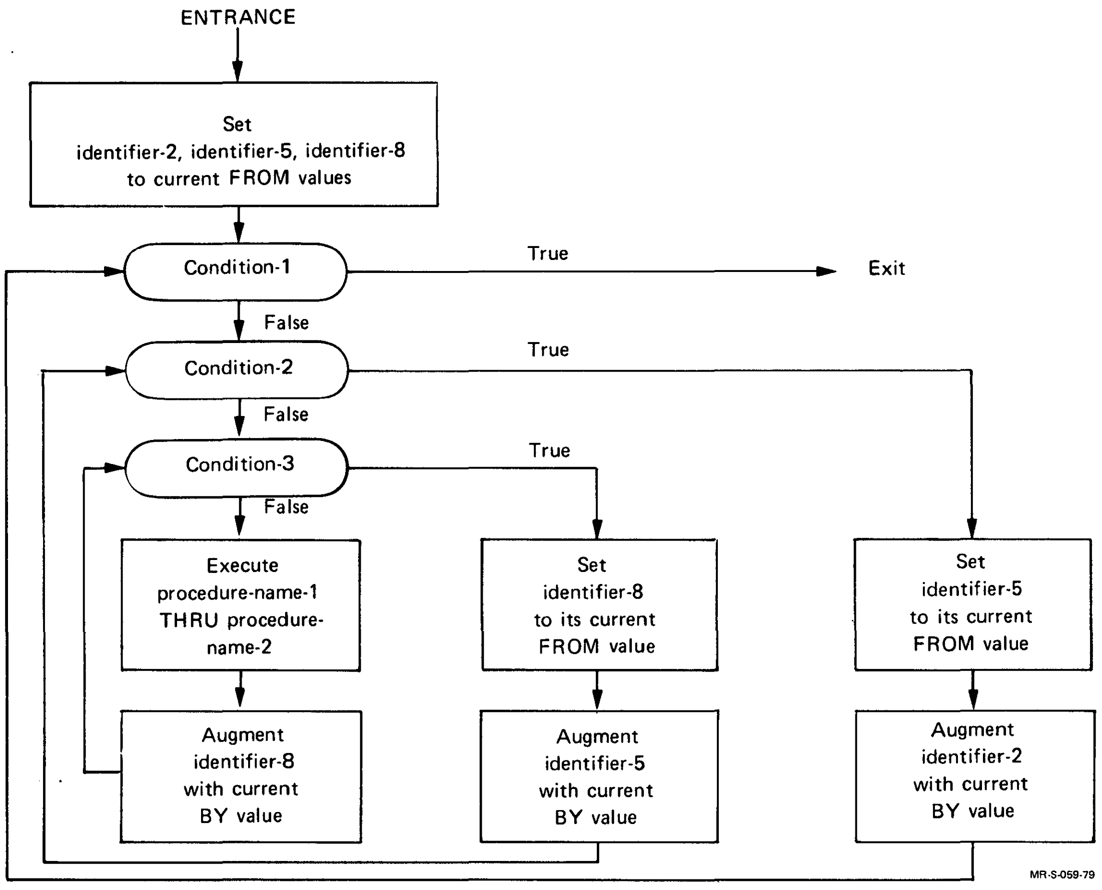

ENTRANCE
Set identifier-2, identifier-5, identifier-8 to current FROM values
Condition-1
True
Exit
False
Condition-2
True
False
Condition-3
True
False
Execute procedure-name-1 THRU procedure-name-2
Set identifier-8 to its current FROM value
Set identifier-5 to its current FROM value
Augment identifier-8 with current BY value
Augment identifier-5 with current BY value
Augment identifier-2 with current BY value
MR-S-059-79

Figure 5-4 PERFORM Cycle Logic - Three Variables

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 may have as its range procedures wholly contained within the DECLARATIVES; however, a PERFORM statement in a section within the DECLARATIVES may 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 may not have within its range any procedures outside of that INPUT or OUTPUT PROCEDURE.

5.9.27 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 [NEXT] RECORD [INTO identifier]

   [AT END imperative-statement]

READ file-name RECORD [INTO identifier] [INVALID KEY imperative-statement]

READ file-name RECORD [INTO identifier]

   [KEY IS data-name]

   [INVALID KEY imperative-statement]
```

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 organization is SEQUENTIAL (explicitly or implicitly). For those files, the AT END phrase must be specified if no applicable USE procedure is specified for file-name.

   The INVALID KEY clause is valid only for those files whose access mode is RANDOM or DYNAMIC.

   For files whose organization is RELATIVE or INDEXED, the INVALID KEY phrase or the AT END phrase must be specified if no applicable USE procedure is specified for file-name.

   If an end-of-file condition is encountered during the execution of a READ statement for a sequential file, any statements specified in the AT END clause are executed, and no logical record is made available.

   The logical end-of-file depends upon the type of device on which the file resides (users of TOPS-10 should see the Monitor Calls Manual, and users of TOPS-20 should see the Monitor Calls Reference 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.

When a READ statement is executed for a file whose organization is RELATIVE, the object-time system makes available the record whose relative record number is equal to the contents of the data item named in the RELATIVE KEY phrase. If no such record exists, the INVALID KEY statements are executed and no record is made available. For relative files whose access mode is DYNAMIC, the NEXT phrase must be specified if you wish to read the file sequentially. If you specify the NEXT phrase the record made available will be the next logical record after the one most recently read, unless there has not been a READ statement since the last OPEN or START statement. If this is the case, the record made available is the first record, in the case of OPEN, or the record specified in the START statement, whether the EQUAL, GREATER THAN, or NOT LESS THAN option is used.

When a READ statement is executed for a file whose organization is INDEXED, a search of the file is made to find the record that has a key equal to the contents of the RECORD 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 statements associated with the INVALID KEY clause are executed, and no record is made available. When a READ NEXT statement is executed for a file whose organization is INDEXED, 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 a START statement was the last reference to the file, the record made available is the one specified in the START statement, or the first of the specified range. That is, if your program contains the following sentence:

```
START MYFILE KEY IS GREATER THAN MIN-KEY INVALID  KEY  GO
TO DISPLAY-ERROR
```

the record made available to your program is the first logical record with a key value greater than MIN-KEY. If no such record exists (that is, you have reached end-of-file), the INVALID KEY statements are executed, and no record is made available. If the file has been opened but no READ, WRITE, REWRITE, DELETE or START 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.

   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.

READ (Cont.)

c. The standard beginning label procedure is executed.

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.

5.9.28 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 may 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 may no longer be available.

RETAIN

5.9.29 RETAIN

Function

The RETAIN statement specifies your intent to access one or more records in a file that is open for simultaneous update.

General Format

```
RETAIN file-name-1 { RECORD      [ KEY { identifier-1 } ] }
                   {                   { literal-1    }   }
                   { NEXT RECORD                          }

      { READ         }
      { REWRITE      }
  FOR { READ-REWRITE } [ UNTIL FREED ]
      { DELETE       }
      { WRITE        }
      { ANY VERB     }

  [ file-name-2 { RECORD      [ KEY { identifier-2 } ] }
  [             {                   { literal-2    }   }
  [             { NEXT RECORD                          }

  [   { READ         }
  [   { REWRITE      }
  [FOR { READ-REWRITE } [ UNTIL FREED ] ...             ]
  [   { DELETE       }
  [   { WRITE        }
  [   { ANY VERB     }                                  ]

[ UNAVAILABLE statement-1 [ statement-2 ] ... ]
```

Technical Notes

1. Filename-1, filename-2... must be the names of files previously opened for simultaneous update.

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 will be 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 will not cause an end-of-file condition.

6. The action 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/rewritten sequentially; the RELATIVE KEY is examined to determine the record to be read/written/rewritten/deleted in a relative file; and the RECORD 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 on these nonexistent records.

8. It is possible to mix requests for records from sequential, random, and indexed-sequential files in the same RETAIN statement.

9. 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). When you retain a record for any use other than READ, all other users are banned completely from accessing that record.

10. 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.)

11. 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.

12. 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.

13. The KEY phrase allows you to specify a particular record or to specify more than one record in a file.

14. 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. The result is that both programs hang or stall waiting for the resource to become available.

15. When attempting to retain records, the program will be 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.

16. Use of the RETAIN statement differs according to the organization of the file. Each type of file is described separately below.

17. Sequential files

    a. Records in a sequential file can be retained for READ, WRITE, READ-REWRITE, or ANY VERB. For sequential files ANY VERB means READ, WRITE, and REWRITE.

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, WRITE, or REWRITE) 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-REWRITE 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 REWRITE 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. The identifier must be numeric, and can be subscripted or qualified or both. Its USAGE should be COMPUTATIONAL or INDEX for the sake of efficiency. 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, when performing simultaneous updating on sequential files, you retain several records at a time so that the input/output overhead will be 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. When you free a record (either implicitly or explicitly), you must write the record out to the device so that other users have access to the latest copy of that record.

Example

```
OPEN INPUT-OUTPUT HISTORY FOR READ AND REWRITE
     ALLOWING OTHERS READ AND REWRITE
                    .
                    .
                    .
RETAIN HISTORY KEY 0 FOR READ-REWRITE UNTIL FREED,
     HISTORY KEY 1 FOR READ-REWRITE UNTIL FREED,
     HISTORY KEY 2 FOR READ-REWRITE;
READ HISTORY, AT END STOP RUN.
                    .
                    .
                    .
```

18. Relative files

a. Records in a relative file can only be retained for READ, WRITE, READ-REWRITE, or ANY VERB. For relative files, ANY VERB means READ, WRITE, and READ-REWRITE.

RETAIN (Cont.)

b. When the KEY phrase is specified, the value of the key designates a specific record in the file, just as the RELATIVE KEY of the file does. Thus, record 1 is always the first record in the file. If you specify the NEXT option, 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, REWRITE 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-REWRITE 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 REWRITE is defined to be the record just read. Note that the next record actually read or written depends on the value of the RELATIVE KEY, not on the record specified in the RETAIN statement.

c. If you wish to read/rewrite the file sequentially, you should select the NEXT option in the RETAIN statement, and use the READ NEXT syntax so that you are performing I/O on the same records that you are retaining. If you wish to read/rewrite the file randomly, you should set the RELATIVE 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 RELATIVE KEY specified for the file.

e. The value of a key can be specified by any identifier. The identifier must be numeric, and may be subscripted or qualified or both. For the sake of efficiency, its USAGE should be COMPUTATIONAL or INDEX. The value of the key can also be specified by a positive integer numeric literal containing ten or fewer digits.

Example

```
OPEN I-O PART FOR READ AND REWRITE 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 REWRITE,
      PART KEY 35 FOR READ AND REWRITE.
REWRITE PARTREC.
MOVE 35 TO PART-ACTUAL-KEY.
READ PART, INVALID KEY GO TO ERR.
REWRITE PARTREC.
```

19. Indexed-sequential 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.

    b. When the KEY phrase is specified, the value of the key refers to a specific record in the file, just as the RECORD 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 numeric, a positive numeric literal of ten or fewer digits can be used as the value in the KEY phrase. For the sake of efficiency the key should be USAGE COMPUTATIONAL or INDEX.

    d. If the KEY phrase is not specified, the value used for the key is taken from the current RECORD KEY for the file.

    e. If NEXT is specified, the record retained is that following the last record referenced in the same RETAIN statement or by a READ, WRITE, REWRITE, or DELETE statement.

Example

```
OPEN I-O LETTERS FOR READ ALLOWING OTHERS READ AND
REWRITE.
MOVE "B" TO RECORD KEY.
RETAIN LETTERS FOR READ.
READ LETTERS INVALID KEY GO TO ERRS.
```

5.9.30 RETURN

Function

The RETURN statement obtains sorted records from the output phase of a SORT or MERGE operation.

General Format

```
RETURN file-name RECORD [INTO identifier]  AT END imperative-statement
```

Technical Notes

1. File-name must be described by an SD file descriptor.
2. A RETURN statement may 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 may be executed until another SORT or MERGE is executed.

5.9.31 REWRITE

Function

The REWRITE statement replaces an already existing record in a file.

General Format

```
REWRITE record-name  [ FROM identifier] [INVALID KEY imperative-statement ]
```

Technical Notes

1. Record-name must be a record associated with a file whose organization is RELATIVE or 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 RECORD KEY associated with the file, 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)
   ```

5. The INVALID KEY phrase must not be specified for a REWRITE statement that references a file in sequential mode. This is because a REWRITE may only be done on a file in sequential-access mode after a successful READ statement is executed.

6. The INVALID KEY phrase must be specified in the REWRITE statement for files in the random- or dynamic-access mode for which an appropriate USE procedure is not specified.

SEARCH

5.9.32 SEARCH

Function

The SEARCH statement is used to search a table until a specified condition exists.

General Format

```
SEARCH identifier-1 [ VARYING { identifier-2 } ] [AT END imperative-statement-1]
                              { index-name-1 }

    WHEN condition-1 { imperative-statement-2 }
                     { NEXT SENTENCE          }

  [ WHEN condition-2 { imperative-statement-3 } ] ...
                     { NEXT SENTENCE          }

SEARCH ALL identifier-1  [AT END imperative-statement-1]

          { data-name-1 { IS EQUAL TO } { identifier-3            } }
     WHEN {             { IS =        } { literal-1               } }
          {                             { arithmetic-expression-1 } }
          { condition-name-1                                        }

     [     { data-name-2 {IS EQUAL TO} { identifier-4            } } ]
     [ AND {             {IS =       } { literal-2               } } ] ...
     [     { condition-name-2          { arithmetic-expression-2 } } ]

     { imperative-statement-2 }
     { NEXT SENTENCE          }
```

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 format 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 format 1, condition-1, condition-2, etc., can be any condition described in Section 5.5.

5. In format 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 will contain the value of the index at each step of the search.

8. If format 1 of the SEARCH verb is used, a serial type of 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 format 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 may 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 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.

10. In either format, 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.
```

5.9.33 SET

Function

The SET statement allows a data-item to be incremented, decremented, or set to a value.

General Format

$$\underline{\text{SET}} \left\{ \begin{matrix} \text{identifier-1} & [\text{identifier-2}] \\ \text{index-name-1} & [\text{index-name-2}] \end{matrix} \ \ldots \right\} \text{TO} \left\{ \begin{matrix} \text{identifier-3} \\ \text{index-name-3} \\ \text{integer-1} \end{matrix} \right\}$$

$$\underline{\text{SET}} \text{ index-name-4 } [\text{index-name-5}] \ \ldots \left\{ \begin{matrix} \underline{\text{UP BY}} \\ \underline{\text{DOWN BY}} \end{matrix} \right\} \left\{ \begin{matrix} \text{identifier-4} \\ \text{integer-2} \end{matrix} \right\}$$

Technical Notes

1. All identifiers must be numeric elementary items described without any positions to the right of the assumed decimal point.

2. All literals must be integers, or the figurative constant ZERO.

3. 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.

SORT

5.9.34 SORT

Function

The SORT statement orders a file or files of records according to the contents of the user-specified keys within each sorted record.

General Format

```
SORT file-name-1  ON {ASCENDING }  KEY data-name-1  [data-name-2]  ...
                     {DESCENDING}

              [ON {ASCENDING }  KEY data-name-3  [data-name-4]  ...]...
                  {DESCENDING}

 [COLLATING SEQUENCE IS alphabet-name]

{INPUT PROCEDURE IS section-name-1  [{THROUGH}  section-name-2]}
{                                   [{THRU   }                ]}
{                                                              }
{USING file-name-2  [file-name-3]  ...                         }

{OUTPUT PROCEDURE IS section-name-3 [{THROUGH}  section-name-4]}
{                                   [{THRU   }                ]}
{GIVING file-name-4                                            }
```

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. 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 may be qualified but not subscripted.

5. SORT statements may 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.

6. 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.

7. 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 in order to transfer records to the sort subroutine.

8. 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 in order to make sorted records available for processing.

9. 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.

10. 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.

11. 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,..., will be 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.
```

12. 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 will be 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.
```

SORT (Cont.)

13. An ISAM file cannot be sorted directly using the non-COBOL standalone SORT.

    ISAM files are by definition a sorted set. In designing the file you should use the order in which the file will be 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. The input procedure can release records to the sort. You can read the file in two ways - sequentially using READ NEXT, or randomly by selecting the desired record and inserting the key value in the RECORD KEY. Usually, reading the file sequentially and allowing SORT to order the records is much faster. 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.

14. The collating sequence for the comparison of the specified nonnumeric key data items is determined in the following order:

    a. First, the collating sequence established by the COLLATING SEQUENCE phrase, if specified, in the SORT statement.

    b. Second, the collating sequence established as the program collating sequence.

15. Refer to the SORT User's Guide for more information on SORT.

5.9.35 START

Function

The START statement provides for logical positioning within a relative or indexed file, for subsequent sequential retrieval of records.

General Format

```
                 ┌      ⎧IS EQUAL TO      ⎫            ┐
                 │      ⎪IS =             ⎪            │
START file-name  │ KEY  ⎨IS GREATER THAN  ⎬  data-name │
                 │      ⎪IS >             ⎪            │
                 │      ⎪IS NOT LESS THAN ⎪            │
                 └      ⎩IS NOT <         ⎭            ┘

    [INVALID KEY imperative-statement]
```

Technical Rules

1. File-name must be the name of a file with sequential or dynamic access.

2. Data-name may be qualified.

3. The INVALID KEY phrase must be the data item specified if no applicable USE procedure is specified for file-name.

4. The file associated with file-name must be open in the INPUT or I-O mode at the time the START statement is executed.

5. If you omit the KEY phrase, you imply the phrase IS EQUAL TO data-name, where data-name refers to the RECORD KEY of an indexed file or the RELATIVE KEY of a relative file.

6. If the file associated with file-name is a relative file, and you include data-name, data-name must be the data item specified in the RELATIVE KEY phrase of the file control entry. If the file is an indexed one and you include data-name, data-name must be either the data item specified as the record key, or an "approximate key". An "approximate key" is a part of a key, whose leftmost character position is the same position as the leftmost position of the RECORD KEY but which is not the entire key. Suppose, for example, you have an ISAM file whose key is of the form

   YY-MM-DD-XX

START (Cont.)

where YY is the year, MM the month, DD the day, and XX the charge sequence number. If you wished to begin processing the file at the first record of July 1978, you could write the following code:

```
SELECT CHARGE-FILE
        ASSIGN TO DSK
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS CHG-REC-KEY.
         .
         .
         .
MOVE "78-07-" TO CHG-REC-KEY.
START CHARGE-FILE KEY IS GREATER THAN CHG-REC-KEY,
        INVALID KEY GO TO ERR-RTN.
```

The effect of this would be to find the first record in the file whose key collates higher than 78-07- and then position the record pointer in front of that record. If you specified NOT LESS THAN instead of GREATER THAN the pointer would be positioned in front of the record whose key is 78-07-ΔΔΔΔΔ if such a record existed; otherwise the pointer would be positioned as in the actual example. Note that only indexed files may use the "approximate key", and that the leftmost positions of the record key and the "approximate key" must be the same character position in the record, not simply contain the same character.

7. If the comparison is not satisfied by any record in the file, the INVALID KEY condition exists, the execution of the START statement is unsuccessful, and your logical position in the file is undefined. When this is the case, the imperative-statements following the INVALID KEY phrase are executed.

8. The execution of the START statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated.

5.9.36 STOP

Function

The STOP statement halts the object program.

General Format

$\underline{\text{STOP}} \left\{ \begin{array}{l} \underline{\text{RUN}} \\ \underline{\text{literal}} \end{array} \right\}$

Technical Notes

1. The literal may be numeric or nonnumeric or may be any figurative constant except ALL.

2. If the literal is numeric, it must be an unsigned integer.

3. If the literal option is used, the literal is displayed on the user's terminal. The literal may be a figurative constant; in this case, a single character is displayed. The program waits for the operator to type

   CONTINUE

   Following receipt of this message, the program continues execution at the statement following the STOP.

4. If the RUN option is used, all files currently open are closed, and execution of the program is terminated.

STRING

5.9.37 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 imperative-statement]
```

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 (according to the rules for numeric transfers) to an intermediate unsigned numeric data item of the same size as the source whose USAGE is the same as that of identifier-7 , and then it is 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. Literal-1; literal-2... are called source literals. Source literals must be alphanumeric literals or alphanumeric figurative constants without the ALL modifier.

   e. If a source literal is a figurative constant, it refers to a single-character literal of the specified type.

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 (according to the rules for numeric transfers) to an intermediate unsigned numeric data item of the same size as the delimiter whose USAGE is the same as that of identifier-7 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, either 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 corrsponding source items is the rightmost boundary of the source item.

STRING (Cont.)

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.

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, will 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 2 of note 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 has been set to the appropriate value, 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 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.

5.9.38 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

```
SUBTRACT {identifier-1} [identifier-2] ...  FROM identifier-m [ROUNDED]
         {literal-1   } [literal-2   ]

    [identifier-n [ROUNDED]] ...  [ON SIZE ERROR imperative-statement]

SUBTRACT {identifier-1} [identifier-2] ...  FROM {identifier-m}
         {literal-1   } [literal-2   ]           {literal-m   }

    GIVING identifier-n [ROUNDED] [identifier-o [ROUNDED]] ...

    [ON SIZE error imperative-statement]

SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
         {CORR         }

    [ON SIZE ERROR imperative-statement]
```

Technical Notes

1. Each SUBTRACT statement must contain at least two operands (that is, a subtrahend and a minuend). In formats 1 and 2, each identifier must refer to an elementary numeric item, except that identifiers to the right of the word GIVING may refer to numeric edited items. In format 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 standard 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. (See Section 1.1 for a definition of the BIS-compiler.)

3. Format 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.

SUBTRACT (Cont.)

4. Format 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. Format 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 in Section 5.7, The CORRESPONDING Option.

6. The ROUNDED and SIZE ERROR options are described in Section 5.6, Common Options Associated with Arithmetic Verbs.

5.9.39 TERMINATE

Function

The TERMINATE statement ends the processing of a report.

General Format

```
TERMINATE report-name-1  [report-name-2]  ...
```

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 may 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

5.9.40 TRACE

Function

The TRACE statement causes the complier to trace paragraphs or to stop tracing paragraphs. When a paragraph is traced, its name, enclosed in angle brackets (<>), is typed each time that the paragraph is entered.

General Format

$$\underline{\text{TRACE}} \left\{ \begin{array}{l} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{array} \right\}$$

Technical Notes

1. The TRACE statement works with the COBDDT utility to help you debug your COBOL-74 program.

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 the user includes the TRACE ON statement in his program.

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.

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.

6. When compiling for a production run, you should include the /P switch in the command string so that trace calls will not be generated and TRACE statements in the program will be 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.
     .
     .
     .
```

Paragraph 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 will be ignored and trace calls will not be generated.

UNSTRING

5.9.41 UNSTRING

Function

The UNSTRING statement is used to split a single data item (for example, a text string) into several parts, depending on the occurrence of specified delimiters, and to store the parts into separate data items where they may be more easily accessed by the COBOL program.

General Format

```
UNSTRING identifier-1

 [DELIMITED BY [ALL] {identifier-2} [OR [ALL] {identifier-3}] ... ]
                     {literal-1   }           {literal-2   }

   INTO identifier-4 [DELIMITER IN identifier-5][COUNT IN identifier-6]

      [identifier-7 [DELIMITER IN identifier-8][COUNT IN identifier-9]] ...

 [WITH POINTER identifier-10][TALLYING IN identifier-11]

 [ON OVERFLOW imperative-statement]
```

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 is 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.

   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 (according to the rules for numeric transfers) to an intermediate unsigned numeric data item of the same size as the delimiter whose USAGE is the same as that of identifier-1 and then is 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 alphanumric 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 which 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.

4. Delimiter Storage Items

   a. A DELIMITER IN phrase may 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.

UNSTRING (Cont.)

5. Count Storage Items

   a. A COUNT IN phrase may be specified only if the DELIMITED BY phrase is specified.

   b. 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 size to contain a value equal to the size of the source.

   c. 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.

   d. 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 may 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, however, 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 interactive 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 will be 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.

   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.

UNSTRING (Cont.)

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 will be 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.

   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 then 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).

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 may occur during the execution of the MOVE statement. This data item may contain no character positions at all if the pointer-indicated position satisfied condition a of Note e1 or it may contain as much as the entire source.

UNSTRING (Cont.)

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 holds, 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 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 stroage 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.

k. When the UNSTRING statement has been executed, 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.

5.9.42 USE

Function

The USE statement specifies procedures for error handling that are in addition to the standard procedures provided by the input-output control system.

General Format

```
                       {EXCEPTION}              {file-name-1 OPEN  [file-name-2] OPEN  ...}
USE AFTER STANDARD     {ERROR    }  PROCEDURE ON {INPUT                                    }.
                                                 {OUTPUT                                   }
                                                 {I-O                                      }
                                                 {EXTEND                                   }
```

```
USE BEFORE REPORTING identifier.
```

Technical Notes

1. USE statements may appear only in the Declaratives portion of the Procedure Division. The Declaratives portion follows immediately after the PROCEDURE DIVISION header and begins with the word

   DECLARATIVES.

   The Declaratives portion ends 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 may 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. Format 1 causes the designated procedures to be executed after completing the standard input-output error routine.

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 may 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 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 will be executed for all INPUT files; if the OUTPUT option is used, they will be executed for all OUTPUT files; if the I-O or the INPUT-OUTPUT option is used, they will be executed for all INPUT-OUTPUT files; if the filename-1 format is used, they will be executed only for that particular file. If more than one USE procedure could apply in a situation, only one will actually be executed. The procedure to be used will be the one which is most restrictive, that is, the one which applies most closely to the situation in question. For example, suppose you specify the file-name-1 option and the OPEN option, and you get an error when you attempt to open file-name-1. The procedure you specified for the file-name-1 option will be executed, but the procedure for the OPEN option will not, because it is less restrictive.

USE (Cont.)

If the filename-1 OPEN format 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. Identifier-1 in Format 2 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.

5.9.43 WRITE

Function

The WRITE statement transfers a logical record to an output file.

General Format

```
WRITE record-name [FROM identifier-1]

[                    ( {identifier-2 } [LINE ] ) ]
[ {BEFORE}           ( {integer      } [LINES] ) ]
[ {AFTER } ADVANCING (                         ) ]
[                    ( {mnemonic-name}         ) ]
[                    ( {PAGE         }         ) ]

[ AT {END-OF-PAGE} imperative-statement ]
[    {EOP        }                      ]

WRITE record-name [FROM identifier] [INVALID KEY imperative-statement]
```

Technical Notes

1. An OPEN OUTPUT, OPEN I-O, OPEN INPUT-OUTPUT or OPEN EXTEND 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 may 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.

4. Format 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.

WRITE (Cont.)

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. A file mark is written, and the tape is rewound.

b. If the file was assigned to more than one tape unit, the units are advanced.

c. If labels are not OMITTED, a label is written on the new tape.

5. If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file description entry for the associated file. The words END-OF-PAGE and EOP are equivalent.

6. The ADVANCING mnemonic-name phrase cannot be specified when writing a record to a file whose file description entry contains the LINAGE clause.

7. 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:

   a. If identifier-2 is specified, it must be described as a one 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 the object-time system interprets the value 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

8. Either ADVANCING or POSITIONING can be specified for a file, but not both. Also, if either is specified, the recording mode of the file will be ASCII, regardless of the recording mode specified in the RECORDING MODE clause.

9. When a WRITE statement is executed for a file whose access mode is RANDOM and the RELATIVE KEY contains a value of 0, records will be written sequentially in the file (that is, no records will be left null). If the previous operation performed on the file was by a READ statement, the previous record will be replaced (that is, the record will be updated).

   The statement(s) in the INVALID KEY clause is executed when the RECORD KEY contains a value equal to the key of an already existing record in an INDEXED file (refer to the REWRITE statement, Section 5.9.31).

10. When executing a WRITE statement for a SEQUENTIAL file opened for 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.

11. The INVALID KEY phrase must be specified if an applicable USE procedure is not specified for the associated file.

12. 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)
```

   Note that identifier-1 must be a data-name and cannot be a figurative constant (for example, SPACES), because it is syntactically equivalent to a literal.

THIS PAGE INTENTIONALLY LEFT BLANK.

THE PROCEDURE DIVISION
VERB FORMATS

THE PROCEDURE DIVISION

GENERAL FORMAT FOR PROCEDURE DIVISION

```
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] ...] ...} ...
```

GENERAL FORMAT FOR VERBS

```
ACCEPT identifier-1   identifier-2   ... [FROM mnemonic-name]

                        { DATE }
ACCEPT identifier FROM  { DAY  }
                        { TIME }


ADD {identifier-1    [identifier-2]   ...  TO identifier-m [ROUNDED]
    {literal-1       [literal-2   ]

  [identifier-n [ROUNDED]] ...   [ON SIZE ERROR imperative-statement]

ADD {identifier-1}  {identifier-2}  [identifier-3]  ...
    {literal-1   }  {literal-2   }  [literal-3   ]

    GIVING identifier-m [ROUNDED]  [identifier-n  [ROUNDED]]   ...
    [ON SIZE ERROR imperative-statement]

ADD {CORRESPONDING}  identifier-1 TO identifier-2  [ROUNDED]
    {CORR         }
  [ON SIZE ERROR imperative-statement]

ALTER procedure-name-1 TO  [PROCEED TO]  procedure-name-2
  [ procedure-name-3 TO  [PROCEED TO]  procedure-name-4 ] ...

      { identifier-1 }
CALL  { program-name }  [USING data-name-1   [data-name-2]  ...]
      { entry-name   }
  [ON OVERFLOW imperative-statement]
```

GENERAL FORMAT FOR VERBS

```
CANCEL { identifier-1 }  [ identifier-2 ]
       { subprogram-1 }  [ subprogram-2 ]

                    [ { REEL } ] [ { WITH   { NO REWIND } } ]
CLOSE file-name-1   [ { UNIT } ] [ {        { LOCK      } } ]
                                 [ {        { DELETE    } } ]
                                 [ { FOR      REMOVAL     } ]

                    [ { REEL } ] [ { WITH   { NO REWIND } } ]
      file-name-2   [ { UNIT } ] [ {        { LOCK      } } ]
                                 [ {        { DELETE    } } ]
                                 [ { FOR      REMOVAL     } ] ...

CLOSE file-name-1 [WITH LOCK] [ file-name-2  [WITH LOCK] ] ...

                                  [ { identifier-2          }             ]
COMPUTE identifier-1 [ROUNDED]    [ { literal               } [ROUNDED]   ] ...
                                  [ { arithmetic-expression }             ]

    { IS EQUAL TO }
    { EQUALS      }  arithmetic-expression  [ON SIZE ERROR imperative-statement]
    { =           }

DELETE file-name RECORD  [INVALID KEY imperative-statement]

DISPLAY { identifier-1 } [ { identifier-2 } ] ... [UPON mnemonic-name] [WITH NO ADVANCING]
        { literal-1    } [ { literal-2    } ]
```

GENERAL FORMAT FOR VERBS

```
DIVIDE  { identifier-1 }  INTO identifier-2 [ROUNDED]
        { literal-1    }

   [ identifier-3 [ROUNDED] ] ...  [ ON SIZE ERROR imperative-statement ]

DIVIDE  { identifier-1 }  INTO  { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

   [ identifier-4 [ROUNDED] ] ...  [ ON SIZE ERROR imperative-statement ]

DIVIDE  { identifier-1 }  BY    { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

   [ identifier-4 [ROUNDED] ] ...  [ ON SIZE ERROR imperative-statement ]

DIVIDE  { identifier-1 }  INTO  { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

   REMAINDER identifier-4  [ ON SIZE ERROR imperative-statement ]

DIVIDE  { identifier-1 }  BY    { identifier-2 }  GIVING identifier-3 [ROUNDED]
        { literal-1    }        { literal-2    }

   REMAINDER identifier-4  [ ON SIZE ERROR imperative-statement ]

        { MACRO   }  [        { identifier-1     }  [ { identifier-2     } ] ]
ENTER   { FORTRAN }  [ USING  { literal-1        }  [ { literal-2        } ] ]  ...
        { COBOL   }  [        { procedure-name-1 }  [ { procedure-name-2 } ] ]


ENTRY entry-name [ USING identifier-1 [identifier-2] ... ]


EXIT.


EXIT [PROGRAM] .
```

GENERAL FORMAT FOR VERBS

```
     ⎧              ⎧ 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] ... ] .

GENERATE { data-name   }
         { report-name }

GO TO procedure-name-1 [procedure-name-2] ...  procedure-name-n

   DEPENDING ON identifier

GOBACK.

IF condition {statement-1   } [ELSE  {statement-2   }]
             {NEXT SENTENCE }        {NEXT SENTENCE }

INITIATE report-name-1  [report-name-2]  ...
```

GENERAL FORMAT FOR VERBS

```
INSPECT identifier-1 TALLYING

 {                 { {ALL       }  {identifier-3}} [{BEFORE}           {identifier-4}]}     }
 {identifier-2 FOR { {LEADING   }  {literal-1   }} [{AFTER }  INITIAL  {literal-2   }]} ... } ...
 {                 { {CHARACTERS                 }                                   }     }


INSPECT identifier-1 REPLACING

 {CHARACTERS BY {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]                                   }
 {              {literal-4   } [{AFTER }           {literal-5   }]                                   }
 {  { {ALL    } {identifier-5}    {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]}     }          }
 {  { {LEADING} {literal-3   } BY {literal-4   } [{AFTER }           {literal-5   }]} ... } ...      }
 {  { {FIRST  }                                                                     }     }          }


INSPECT identifier-1 TALLYING

 {                 { {ALL       }  identifier-3} [{BEFORE}           {identifier-4}]}     }
 {identifier-2 FOR { {LEADING   }  literal-1   } [{AFTER }  INITIAL  {literal-2   }]} ... } ...
 {                 { {CHARACTERS               }                                  }     }

REPLACING

 {CHARACTERS BY {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]                                   }
 {              {literal-4   } [{AFTER }           {literal-5   }]                                   }
 {  { {ALL    } {identifier-5}    {identifier-6} [{BEFORE}  INITIAL  {identifier-7}]}     }          }
 {  { {LEADING} {literal-3   } BY {literal-4   } [{AFTER }           {literal-5   }]} ... } ...      }
 {  { {FIRST  }                                                                     }     }          }


MERGE [WITH SEQUENCE CHECK] file-name-1 ON {ASCENDING } KEY data-name-1 [ data-name-2 ] ...
                                           {DESCENDING}

                                      [ ON {ASCENDING } KEY data-name-3 [ data-name-4 ] ... ] ...
                                      [    {DESCENDING}                                     ]

   [COLLATING SEQUENCE IS alphabet-name]
    USING file-name-2  file-name-3  [file-name-4] ...

    OUTPUT PROCEDURE IS section-name-1   THROUGH   section-name-2
                                         THRU

    GIVING file-name-5
```

GENERAL FORMAT FOR VERBS

```
MOVE  { identifier-1 }  TO identifier-2  [ identifier-3 ] ...
      { literal      }

MOVE  { CORRESPONDING }  identifier-1 TO identifier-2
      { CORR          }


MULTIPLY  { identifier-1 }  BY identifier-2 [ ROUNDED ]
          { literal-1    }

    [ identifier-3 [ ROUNDED ] ] ...  [ ON SIZE ERROR imperative-statement ]

MULTIPLY  { identifier-1 }  BY  { identifier-2 }  GIVING identifier-3 [ ROUNDED ]
          { literal-1    }      { literal-2    }

    [ identifier-4 [ ROUNDED ] ] ...  [ ON SIZE ERROR imperative-statement ]


PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ]
                           { THRU    }

PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ] { identifier-1 }  TIMES
                           { THRU    }                     { integer-1    }

PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ]  UNTIL condition-1
                           { THRU    }
```

THE PROCEDURE DIVISION

GENERAL FORMAT FOR VERBS

```
PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ]
                           { THRU    }

   VARYING { identifier-2 }  FROM  { identifier-3 }
           { index-name-1 }        { index-name-2 }
                                   { literal-1    }

      BY { identifier-4 }  UNTIL condition-1
         { literal-3    }

 [ AFTER { identifier-5 }  FROM  { identifier-6 }
 [       { index-name-3 }        { index-name-4 }
                                 { literal-3    }

      BY { identifier-7 }  UNTIL  condition-2
         { literal-4    }

 [ AFTER { identifier-8 }  FROM  identifier-9
 [       { index-name-5 }        index-name-6
                                 literal-5

      BY { identifier-10 }  UNTIL  condition-3 ] ]
         { literal-6     }

READ file-name  [NEXT]  RECORD  [INTO identifier]

   [AT END imperative-statement]

READ file-name RECORD [INTO identifier] [INVALID KEY imperative-statement]

READ file-name RECORD [INTO identifier]

   [KEY IS data-name]

   [INVALID KEY imperative-statement]

RELEASE record-name [FROM identifier]
```

GENERAL FORMAT FOR VERBS

```
                       ┌           ┌ identifier-2 ┐┐
SEARCH identifier-1    │ VARYING   ┤              ├│ [AT END imperative-statement-1]
                       └           └ index-name-1 ┘┘

     WHEN condition-1  ┌ imperative-statement-2 ┐
                       └ NEXT SENTENCE          ┘

   ┌                   ┌ imperative-statement-3 ┐┐
   │ WHEN condition-2  │                        ││ ...
   └                   └ NEXT SENTENCE          ┘┘

SEARCH ALL identifier-1   [AT END imperative-statement-1]

          ┌                ┌ IS EQUAL TO ┐  ┌ identifier-3           ┐ ┐
          │ data-name-1    │             │  │ literal-1              │ │
   WHEN   │                └ IS =        ┘  └ arithmetic-expression-1┘ │
          │                                                            │
          └ condition-name-1                                           ┘

          ┌        ┌                ┌ IS EQUAL TO ┐ ┌ identifier-4            ┐ ┐ ┐
          │        │ data-name-2    └ IS =        ┘ │ literal-2               │ │ │
          │  AND   │                                └ arithmetic-expression-2 ┘ │ │ ...
          └        └ condition-name-2                                           ┘ ┘

          ┌ imperative-statement-2 ┐
          └ NEXT SENTENCE          ┘

                                                          ┌ identifier-3 ┐
SET  ┌ identifier-1   [identifier-2]      ┐  TO            │ index-name-3 │
     └ index-name-1   [index-name-2] ...  ┘               └ integer-1    ┘

SET index-name-4   [index-name-5]  ...  ┌ UP BY   ┐ ┌ identifier-4 ┐
                                        └ DOWN BY ┘ └ integer-2    ┘
```

GENERAL FORMAT FOR VERBS

```
SORT file-name-1  ON  {ASCENDING }  KEY data-name-1  [data-name-2]  ...
                      {DESCENDING}

                 [ON  {ASCENDING }  KEY data-name-3  [data-name-4]  ...]...
                      {DESCENDING}

    [COLLATING SEQUENCE IS alphabet-name]

   {INPUT PROCEDURE IS section-name-1  [{THROUGH}  section-name-2]}
   {                                    {THRU   }                 }
   {USING file-name-2  [file-name-3]  ...                         }

   {OUTPUT PROCEDURE IS section-name-3 [{THROUGH}  section-name-4]}
   {                                    {THRU   }                 }
   {GIVING file-name-4                                            }


                        {IS EQUAL TO      }
                        {IS =             }
START file-name  [ KEY  {IS GREATER THAN  }  data-name ]
                        {IS >             }
                        {IS NOT LESS THAN }
                        {IS NOT <         }

      [INVALID KEY imperative-statement]


STOP {RUN    }
     {literal}


STRING {identifier-1} [identifier-2]  ...  DELIMITED BY {identifier-3}
       {literal-1   } [literal-2   ]                    {literal-3   }
                                                        {SIZE        }

     [{identifier-4} [identifier-5]  ...  DELIMITED BY {identifier-6}]...
     [{literal-4   } [literal-5   ]                    {literal-6   }]
     [                                                 {SIZE        }]

      INTO identifier-7 [WITH POINTER identifier-8]
     [ON OVERFLOW imperative-statement]
```

GENERAL FORMAT FOR VERBS

```
SUBTRACT {identifier-1} [identifier-2] ... FROM identifier-m [ROUNDED]
         {literal-1   } [literal-2   ]

    [identifier-n [ROUNDED]] ... [ON SIZE ERROR imperative-statement]

SUBTRACT {identifier-1} [identifier-2] ... FROM {identifier-m}
         {literal-1   } [literal-2   ]          {literal-m   }

    GIVING identifier-n [ROUNDED] [identifier-o [ROUNDED]] ...

    [ON SIZE error imperative-statement]

SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
         {CORR         }

    [ON SIZE ERROR imperative-statement]

TERMINATE report-name-1 [report-name-2] ...

TRACE {ON }
      {OFF}

UNSTRING identifier-1

   [DELIMITED BY [ALL] {identifier-2} [OR [ALL] {identifier-3}] ... ]
                       {literal-1   }           {literal-2   }

    INTO identifier-4 [DELIMITER IN identifier-5][COUNT IN identifier-6]

       [identifier-7 [DELIMITER IN identifier-8][COUNT IN identifier-9]] ...

   [WITH POINTER identifier-10][TALLYING IN identifier-11]

   [ON OVERFLOW imperative-statement]
```

GENERAL FORMAT FOR VERBS

```
                                                 ( file-name-1  OPEN[file-name-2]OPEN...)
                        {EXCEPTION}              ( INPUT                                 )
USE AFTER STANDARD      {ERROR    }  PROCEDURE ON< OUTPUT                                >.
                                                 ( I-O                                   )
                                                 ( EXTEND                                )


USE BEFORE REPORTING identifier.


WRITE record-name [FROM identifier-1]

  [                          ({identifier-2}[LINE ])]
  [ {BEFORE}   ADVANCING     ({integer     }[LINES])]
  [ {AFTER }                 ({mnemonic-name}      )]
  [                          ({PAGE         }      )]

  [ AT {END-OF-PAGE}  imperative-statement ]
  [    {EOP        }                       ]

WRITE record-name [FROM identifier] [INVALID KEY imperative-statement]
```

CHAPTER 6

COMPILER COMMAND STRINGS

The general form of the compiler command string is as follows:

```
relfil,lstfil= libfil/l, src1,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 the optional library file referenced by COPY verbs in the source files.
src1,src2	are one or more source files required to form one 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 may 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 E, 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 .COB. If that file is not found, the compiler looks for the named file with the extension .CBL. 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, the user's 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.

Table 6-1
COBOL Switch Summary

Switch	Action by Compiler
A	List the machine code generated in the lstfil.
B	Generate code for all DEBUG lines (those with /D in col. 7) which otherwise would be treated as comments.
C	Produce a cross-reference table of all user-defined symbols.
D:nnnnnn	Increment, in octal words, 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-74 command strings and switches.
I	Suppress output of start address (program is to be used only by CALL's).
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 the user defined items in the lstfil.
N	Do not type compilation errors on the user's 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.
R	Produce a two-segment object program. The high segment will contain the procedure division; the low segment all else.
S	The source file is in "conventional" format (with sequence numbers in cols. 1-6 and comments starting in col. 73).
U	Produce a one-segment object program.

Table 6-1 (Cont.)
COBOL Switch Summary

Switch	Action by Compiler
W	Rewind the device before reading or writing (magtape only).
X	Give a usage of DISPLAY-9 to items whose usage is either omitted or declared as DISPLAY.
Y	Flag DIGITAL extensions to ANS-74 standard.
Z	Zero the directory of the device before writing (DECtape only).

CHAPTER 7

COBOL-74 UTILITY PROGRAMS

COBOL-74 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-74 Programs**

  RERUN provides you with the ability to restart a COBOL-74 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 may 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 which you may use to perform these functions:

B	Build an indexed file from a sequential one
I	Ignore errors in packing a file (this switch may only be used with the P switch)
L	Read or write standard tape labels (this switch may only be used with the B or P switches)
M	Maintain your indexed file by reorganizing it
P	Pack your indexed file for backup storage

Figure 7-1 shows the COBOL-74 ISAM File Environment.

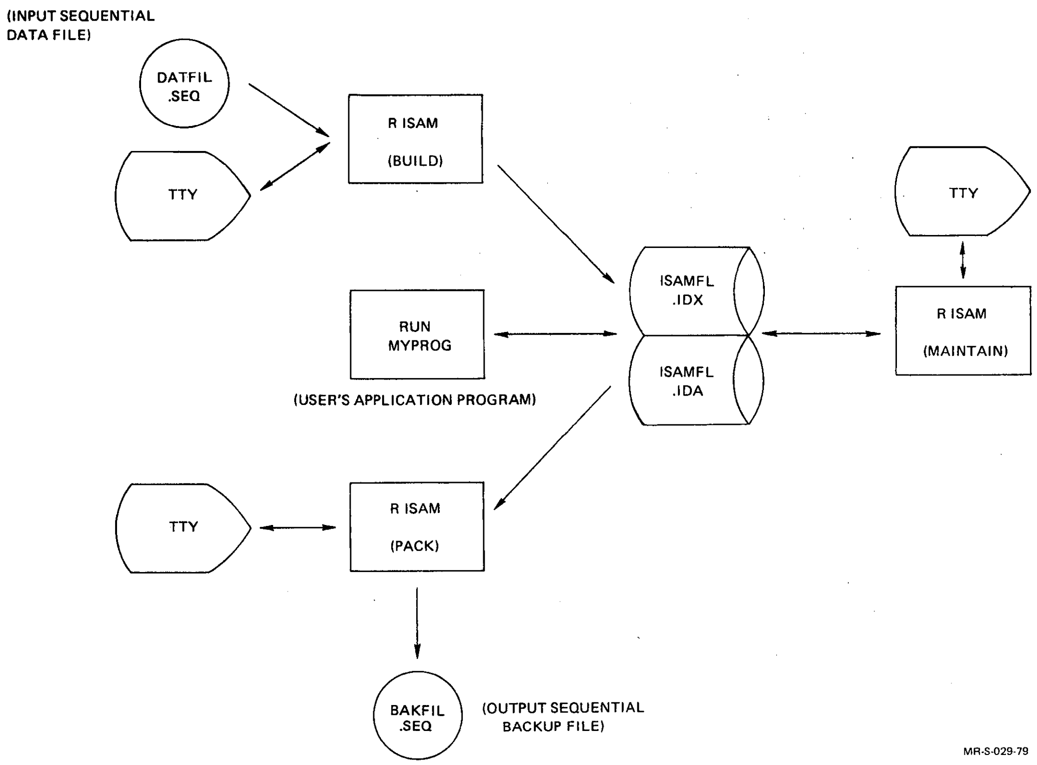


Figure 7-1 COBOL-74 ISAM File Environment

7.1.1 Building an Indexed-Sequential File

To build an indexed-sequential file you must provide a sequential file in which the record keys are arranged in ascending order. The ISAM program will use 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
```

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 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 may 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 E, Defining Logical Names under TOPS-20.
>
> /B is the switch signifying that ISAM will be 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.

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 will be converted in the same manner as they are converted in standard COBOL-74 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 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 number of the character in the record where the key begins.

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, 0 should be specified.

TOTAL RECORDS PER DATA BLOCK:

Give the total number of records to be contained in each block of the data file.

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:

Specify the total number of index entries to be contained in each block of the index file.

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 must equal or exceed two.

PERCENTAGE OF DATA FILE TO LEAVE EMPTY:

Give, as a percentage of the total number of blocks, the number of blocks to be initially left empty in the data file.

PERCENTAGE OF INDEX FILE TO LEAVE EMPTY:

Give, as a percentage of the total number of blocks, the number of blocks to be initially left empty in the index file.

MAXIMUM NUMBER OF RECORDS FILE CAN BECOME:

Reply with the maximum number of records that the data file can possess before the file is next maintained. This number sets the upper limit of the size of the data file. It is required because storage allocation tables must be set up in the index when the file is created. There is no harm in making this number excessively large because the index data blocks are allocated in the storage allocation tables, but not actually assigned until needed.

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: 40
     KEY DESCRIPTOR: SN37.4
     (The key is signed numeric display; it begins in the
      thirty-seventh byte; and it is four bytes long.)
     RECORDS PER INPUT BLOCK: 3
     TOTAL RECORDS PER DATA BLOCK: 2
     EMPTY RECORDS PER DATA BLOCK: 1
     TOTAL ENTRIES PER INDEX BLOCK: 3
     EMPTY ENTRIES PER INDEX BLOCK: 1
     PERCENTAGE OF DATA FILE TO LEAVE EMPTY: 60
     PERCENTAGE OF INDEX FILE TO LEAVE EMPTY: 10
     MAXIMUM NUMBER OF RECORDS FILE CAN BECOME: 12000
```

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 may be added to the file and the number of index levels may 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 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 may 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 E, 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.

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.

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.

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.

Example - Maintaining an indexed-sequential file

```
.R ISAM<RET>

*test.idx, test.ida=test /m
total records per data block (2):
empty records per data block (1):
total entries per index block (3): 32
empty entries per index block (1): 10
percentage of data file to leave empty (60): 50
percentage of index file to leave empty (10): 40
maximum number of records file can become (12000) 25000
```

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. 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

     *dev1:seqfil.ext[ppn1]=dev2:indfil.ext[ppn2] /P<RET>
```

where:

dev1 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.

[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 may 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 E, 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 THE 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, the user answers 0.

Example - Packing an indexed-sequential file

```
.R ISAM
*MTA2:TEST.SEQ=TEST.IDX /P
MODE OF THE OUTPUT FILE: SIXBIT
RECORDS PER OUTPUT BLOCK: 0
```

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 will continue the packing operation. The /I switch can be used only with the /P switch. It cannot be used alone.

The command string when using the /I and /P switches 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/I<RET>
```

where:

devl 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.

[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 may 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 E, 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 may 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 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-74 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.

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

*devl: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 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 may 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 E, Defining Logical Names under TOPS-20.

/B is the switch signifying that ISAM will be 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 will read 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 devl 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 devl 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 may 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 E, 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 will write standard tape labels. It must be included.

7.1.6 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 may 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 E, 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.

7.1.7 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. 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.       ORGANIZATION IS INDEXED
   3.       ACCESS MODE IS DYNAMIC
   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).

                  .
                  .
        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.
11.       READ ISAM-FILE NEXT RECORD, 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 ORGANIZATION clause is required.
3. The ACCESS MODE clause is required if you wish to access the file in random fashion, since the ACCESS MODE defaults to sequential. When DYNAMIC is specified, as here, either random or sequential access may take place.
4. The RECORD KEY clause is required in the Environment Division and refers to the data-item designated as the record key which appears in the Data Division within the FD area record description for the indexed-sequential file.
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. When the READ statement is executed, the record accessed is the first record whose record key has a value higher than the last record processed by a READ, WRITE, REWRITE or DELETE statement. If the file has been opened but no READ, WRITE, DELETE or REWRITE statement has been executed, the first record of the file is read.

7.2 LIBARY - SOURCE LIBRARY MAINTENANCE PROGRAM

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 lines or whole 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 Part 2, Section 1.4, for information on the COPY verb.)

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-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. Enter the file specification command line according to the following 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.
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 may 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 E, 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.)

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 7.2.6.4, 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, will be unchanged and the scratch file will be 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 will be 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.
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 may 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 E, 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 the user terminal all of the library-names contained on the input library file.

/H - List on the user 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 conventional 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 (*). 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 will keep you from having to restart LIBARY and reprocess your file.

LIBARY is organized so that you can optionally create new library files, insert or delete statement groups into an existing file, or make line-by-line changes to an existing file. It has, therefore, two major modes of operation: group mode or edit mode. Group mode provides a means of inserting, replacing, extracting, and deleting entire statement groups; edit mode provides a means of inserting new lines or deleting or modifying existing ones.

NOTE

Edit mode in LIBARY acts as a text editor for the library. However, this editor is not as powerful or as useful as the text editors provided with the operating system (such as TECO and EDIT). LIBARY edit mode is there for historical reasons and its use is not recommended.

7.2.6 LIBARY Commands

The following sections describe the commands available with LIBARY. LIBARY commands are divided into three classes of commands:

- Group mode (See Section 7.2.6.1)
- Edit mode (See Section 7.2.6.2)
- LIBARY-directing (See Section 7.2.6.4)

These commands may be abbreviated as long as you supply a unique abbreviation.

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:

NOTE

For the remainder of this chapter, the words "line number" refer to the line numbers generated by a system standard editor; the words "COBOL line number" refer to the conventional line numbers as described in Part 2, Section 1.3, Source Program Format.

DELETE, library-name

Delete the statement group identified by library-name from the library file. The library-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-name. When library-name is reached, LIBARY positions itself at the next sequential library entry and waits for another command.

EXTRACT, library-name, file-specification

Extract the complete library entry specified by library-name from the input library file and generate a new file named file-name. LIBARY searches the input library file for the library entry specified by library-name. When library-name is found, it creates a file or overwrites an existing file with the attributes specified by file-name and copies the library entry onto it. The input library file remains unchanged.

INSERT, library-name, file-specification

Insert the statement group contained on the file specified by file-name into the output library file. The statement group is inserted alphabetically according to the name specified by library-name. The file specified by file-name must be an ASCII file. LIBARY assumes that the entire file is to be inserted under library-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-name, file-specification

Replace the library entry identified by library-name with the statement group contained on the file specified by file-name. The file specified by file-name must be an ASCII file. LIBARY assumes that the entire file is to replace the statements currently associated with library-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 **Edit Mode Commands** - Edit mode commands allow you to create a library file or modify an existing one with line-by-line edits from your terminal. To edit your file, you must first specify one of the following commands to enter edit mode; after which, you can enter an appropriate edit command to affect the actual editing you wish to perform:

CORRECT, library-name

Positions LIBARY to the group of statements specified by library-name and enters edit mode. Any of the commands described in Section 7.2.6.3, Edit Commands, can be entered at this time. If the /N switch is specified, LIBARY puts new line numbers on the output (corrected) statements. (See Section 7.2.4, LIBARY Switches.)

INSERT, library-name

Positions LIBARY at the place in the library file that the specified library-name will be inserted. It then enters edit mode and waits for you to enter statements that will compose the module. The I command, described in Section 7.2.6.3, is used for this purpose.

REPLACE, library-name

Positions LIBARY at the statement group specified by library-name and deletes it. It then enters edit mode and waits for you to insert source lines by means of the I command. (See Section 7.2.6.3, Edit Commands.)

7.2.6.3 **Edit Commands** - The commands given in this section allow you to insert, delete, and replace individual source lines in a statement group. Source lines should be edited in numeric order within a statement group because LIBARY can only move forward in the file. The following edit commands are provided:

Dnnnnnn

Delete the line specified by nnnnnn. The line number can be entered without leading zeros. That is, you need not enter six characters unless there are that many characters actually in the line number.

Innnnnn COBOL statement

> Insert the COBOL statement into the statement group according to the line number specified by nnnnnn. The line number can be entered without leading zeros. A space or tab must be included between the line number and the COBOL statement; the space will not be included in the statement, but the tab will.

Rnnnnnn COBOL-statement

> Replace the source line identified by nnnnnn with the specified COBOL-statement. The line number can be entered without leading zeros. A space or tab must be included between the line number and the statement; the space will not be included in the statement, but the tab will.

7.2.6.4 **LIBARY-Directing Commands** - LIBARY-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.

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.

NOTE

> LIBARY maintains source modules in ascending order. Line numbers within modules are also in ascending order. If you want to go back in processing to a line previously passed, use the RESTART command.

7.2.6.5 **Example of Command Usage** - A library on disk contains the routines PAYCOMP, FIND-MP, and MP-DESCR. This example shows you how 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
*I10            LABEL RECORDS ARE STANDARD;
*I20            VALUE OF ID IS "JOBS  DAT";
*I30            DATA RECORD IS JOB-RECORD.
*CORRECT        MP-DESCR/N
*I5             BLOCK CONTAINS 5 RECORDS
*DELETE         PAYCOMP
*END
```

The file LIBARY.NEW now contains the following:

1. FIND-MP

2. JOB-DESC

3. MP-DESCR, altered to appear as follows:

```
000010    BLOCK CONTAINS 5 RECORDS
000020    LABEL RECORDS ARE OMITTED
000030    DATA RECORD IS MP-RECORD.
```

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 program behavior

7.3.1 Loading and Starting COBDDT

COBDDT is run after it is loaded and started with a compiled program.

NOTE

> The program being debugged must not have been compiled with the /P switch. The /P switch suppresses the user symbols that are necessary for COBDDT.

The program and COBDDT can be loaded by either the monitor LOAD command or direct commands to LINK. In either case, LINK must load user symbols along with the program and COBDDT. The /LOCALS switch in the LINK command string causes the necessary user symbols to be loaded. After loading, the user issues a monitor command to start the program. The monitor command DEBUG can also be used to load and start COBDDT with a COBOL program. You can specify the name of the source file or the relocatable binary file. If the program cannot be recognized as a COBOL program (i.e., its extension is not .CBL), the /COBOL switch must be included in the DEBUG command string. When COBDDT is loaded with the user program, COBDDT is started, not the program. The three methods of loading and starting are shown below. Although all system prompts shown are for TOPS-10, TOPS-20 acts exactly the same way.

1.
```
.LOAD % "LOCALS" file spec, SYS:COBDDT<RET>
.START<RET>
```

2.
```
.DEBUG file spec [/COBOL]<RET>
```

3.
```
.R LINK<RET>
*/LOCALS file spec, SYS:COBDDT /GO<RET>
.START<RET>
```

When the program is started, COBDDT is entered. This is shown by the message:

```
STARTING COBDDT
*
```

You can now issue any COBDDT command (described below). If you want to run your program at this time, enter the PROCEED command. This will cause 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 REENTER. For example:

```
^C^C REENTER
```

This will cause COBDDT to display the following message:

```
DO YOU WANT TO ENTER COBDDT (Y or N)
```

If you enter Y, COBDDT will be entered at the next TRACE entry in the COBOL program. If you enter N, however, your COBOL program will be reentered at the reenter address.

7.3.2 COBDDT Commands

The commands to COBDDT are described below. Only the first letter of each command needs to be typed for COBDDT to recognize the 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 may be qualified by section-names, and data-names may 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.

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 forms:

```
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:

```
ACCEPT VAR1<RET>
16.25<RET>
```

BREAK

The BREAK command sets a breakpoint (or pause) at the beginning of the specified paragraph. A breakpoint cannot be set on a section. The form of the BREAK command is:

```
BREAK paragraph-name
```

Not more than 20 breakpoints can be set in a program. Breakpoints cannot be set in the high segment of a reentrant program.

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.

Example:

```
BREAK PAR1 IN COMPUTING
```

CLEAR

The CLEAR command removes the breakpoint at a specified paragraph. The CLEAR command has the forms:

```
CLEAR paragraph-name
CLEAR
```

If the paragraph-name is not specified, all breakpoints that have been set in the program are removed.

Example:

```
CLEAR PAR1 IN COMPUTING
```

DISPLAY

The DISPLAY command causes the contents of a data item to be displayed on the user's terminal. The forms of the DISPLAY command are:

```
DISPLAY
DISPLAY data-name
```

If no data-name is specified, COBDDT uses the last data-name specified in an ACCEPT or DISPLAY command.

Example:

```
DISPLAY ALPHA
```

MODULE

The MODULE command causes COBDDT to look for data names and procedure names in the specified program. The form of the MODULE command is:

```
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 will take place. All subsequent searches for data names and procedure names will be 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:

```
MODULE MYPROG
```

OVERLAY

The OVERLAY command either causes a break when an overlay is entered or clears the breakpoint. The forms of the OVERLAY command are:

```
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 forms:

```
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:

```
PROCEED 3
```

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 form:

```
STOP
```

TRACE

The TRACE command either starts or stops tracing, depending on the form of the command. The forms of the TRACE command are:

```
TRACE ON
TRACE OFF
```

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 the user's 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:

```
TRACE ON
!!*!**<PARA>
```

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.

WHERE

The WHERE command causes COBDDT to list the names of all paragraphs at which breakpoints were set. The form of the WHERE command is:

```
WHERE
```

If more than one module is in memory, the module name is included with the paragraph name.

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, the user's 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 may 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 E, Defining Logical Names under TOPS-20.

The title is the one that will be 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, the user's 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 may 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 E, Defining Logical Names under TOPS-20.

The title is the one that will be 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 will add 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 elapsed time and CPU time that is unaccounted for

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 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, the user's 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 may 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 E, Defining Logical Names under TOPS-20.

The title is the one that will be 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 the user 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 will continue 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 will be 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-74 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. They 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, however, if any files are open for input/output (updating), if an indexed-sequential file is open when the dump is requested, or if a sort is in progress. Each time the checkpoint file is written, the COBOL system types the message DUMP COMPLETED on the user's 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 may respond RERUN). For example:

```
     .R RERUN<RET> for users of TOPS-10

                 or

     @RERUN<RET> for users of TOPS-20
```

The program responds with the message:

```
     TYPE CHECKPOINT FILENAME
```

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 a logical device name is encountered in the program, RERUN types the following message:

```
     ASSIGN device name
     TYPE CONTINUE WHEN DONE
```

and exits to monitor command level. The appropriate ASSIGN command should be given to assign the logical device to a specific one. Then a CONTINUE monitor command will reenter RERUN.

7.4.2 Examples of Using RERUN

In the following example, the user has 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, the user employs the RERUN program to restart his program from the last checkpoint written before the program stopped. He types:

```
.R RERUN<RET>
```

and RERUN responds:

```
TYPE CHECKPOINT FILENAME
```

The user types:

```
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, a user running a COBOL program is notified that the system is going down. He does not have any RERUN statements in his program, yet he wishes to create a checkpoint file so that the processing done by his COBOL program up to that point is not wasted. He creates 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, the user can restart the program by running the RERUN program. The dialogue is as follows:

```
@RERUN<RET>
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:

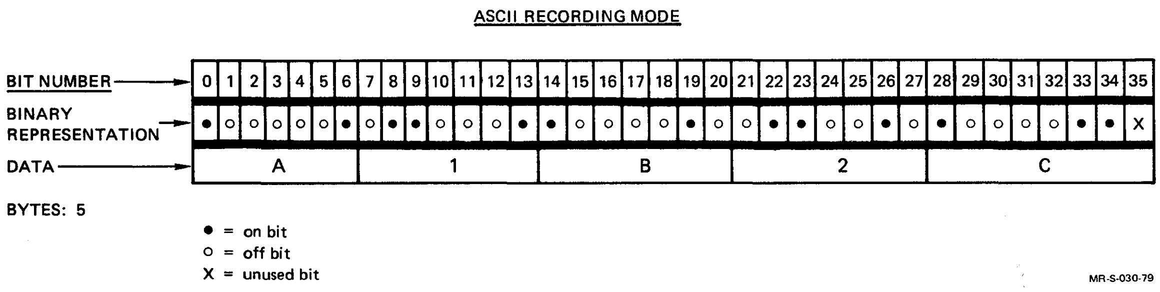


Figure 8-1 ASCII Recording Mode

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:

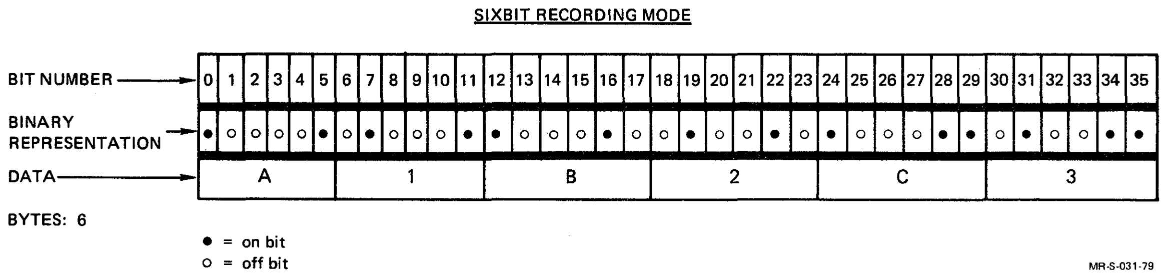


Figure 8-2 SIXBIT Recording Mode

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:

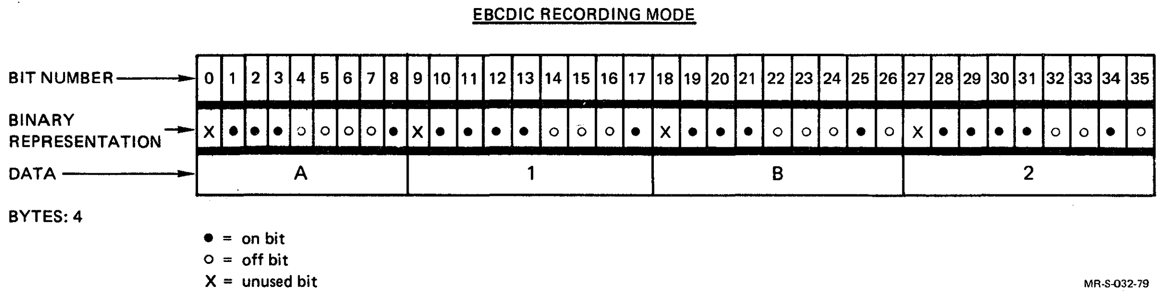


Figure 8-3 EBCDIC Recording Mode

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:

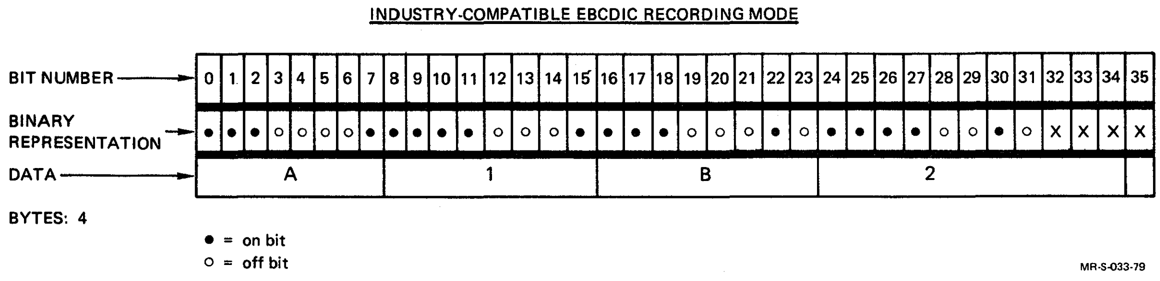


Figure 8-4 EBCDIC Recording Mode - Industry-Compatible

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:

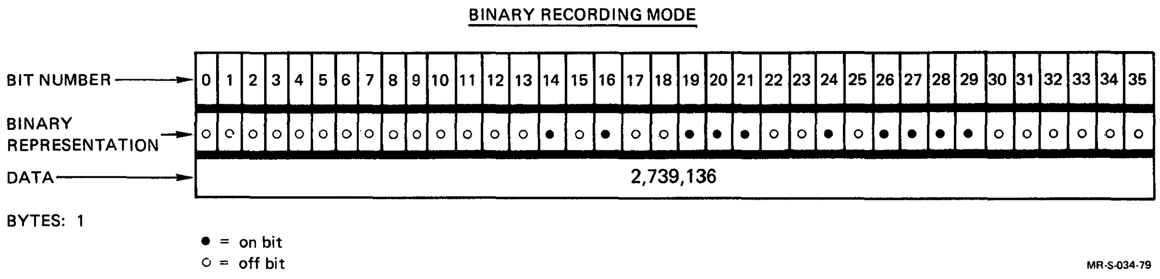


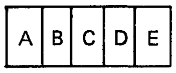
Figure 8-5 Binary Recording Mode

8.2 FILE FORMATS

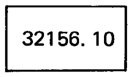
The file format specifies the structure of the record used to store the data. The following sections describe all major file formats. Each section includes a diagram of the file format and a COBOL code segment that will generate the file format.

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:



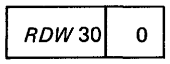
2. Binary data in a word (fixed- and floating-point numbers) is shown by a number in the word:



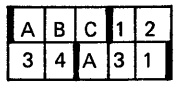
3. EBCDIC packed-decimal values are shown as two decimal digits per EBCDIC byte. The right half of the rightmost byte contains the sign. Neither the digits nor the sign are EBCDIC characters.

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.

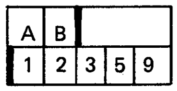
5. Italicized characters in a diagram do not depict data; they label or clarify parts of the diagram:



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 5 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:

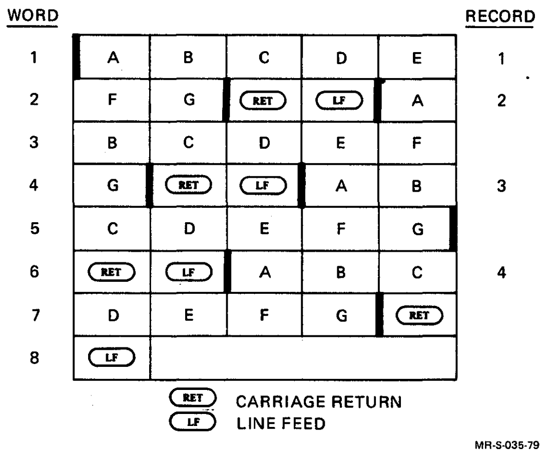


Figure 8-6 Fixed-Length ASCII

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) VALUE "AB12EF".
    02  field-2   PIC A(3) VALUE "GHI".
    02  field-3   PIC 9(4) VALUE 3249.
    02  field-4   PIC S9(6) VALUE -481253.
    02  field-5   PIC S9(6) V9999 VALUE +31458.5012.
```

Figure 8-7 illustrates the record produced by the code segment shown above:

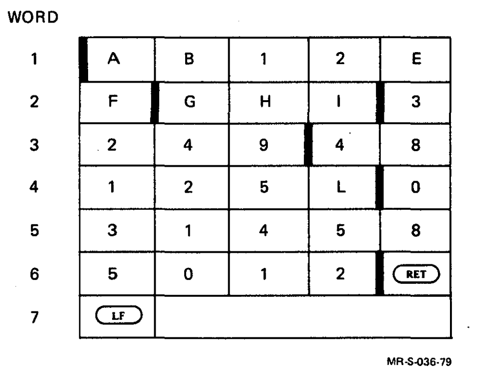


Figure 8-7 COBOL Fixed-Length ASCII

8.2.2 Variable-Length ASCII

Variable-length ASCII consists of records containing five characters per 36-bit word, with each group of 5 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:

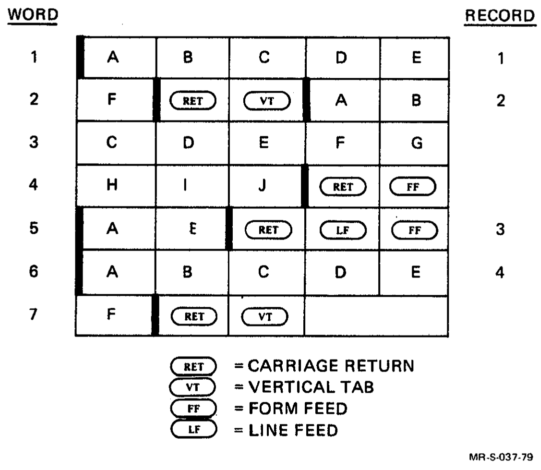
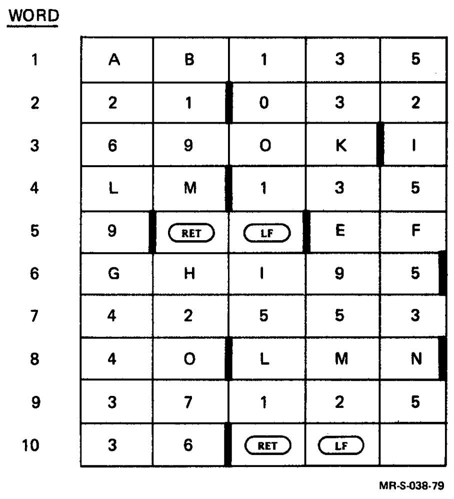


Figure 8-8 Variable-Length ASCII

CODE SEGMENT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

SELECT filename ASSIGN TO DSK
     RECORDING MODE IS ASCII.

FILE SECTION.

FD   filename   VALUE OF ID 'DATA  FIL'.
01   record-1   DISPLAY-7.
     02  field-1  PIC X(7) VALUE "AB13521".
     02  field-2  PIC S9(7) V99 VALUE -3269.02.
     02  field-3  PIC A(3) VALUE "ILM".
     02  field-4  PIC 9(4) VALUE 1359.

01   record-2   DISPLAY-7.
     02  field-1  PIC X(7) VALUE "EFGHI95".
     02  field-2  PIC S9(7) V99 VALUE 42553.40.
     02  field-3  PIC A(3) VALUE "LMN".
     02  field-4  PIC 9(7) VALUE 3712536.

PROCEDURE DIVISION.

     WRITE record-1.
     WRITE record-2.
```

Figure 8-9 illustrates the record produced by the code segment shown above:



WORD					
1	A	B	1	3	5
2	2	1	0	3	2
3	6	9	O	K	I
4	L	M	1	3	5
5	9	RET	LF	E	F
6	G	H	I	9	5
7	4	2	5	5	3
8	4	O	L	M	N
9	3	7	1	2	5
10	3	6	RET	LF	

MR-S-038-79

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:

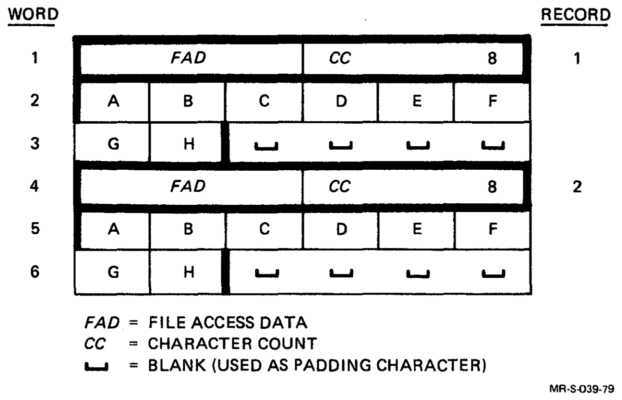


Figure 8-10 Fixed-Length SIXBIT

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) VALUE "A13B".
     02  field-2  PIC A(5) VALUE "CDEFG".
     02  field-3  PIC 9(10) COMP VALUE 9654839218.
     02  field-4  PIC X(2) VALUE "HI".
     02  field-5  PIC 9(11) COMP VALUE 34567982314.
     02  field-6  PIC 9(4) VALUE 1289.
     02  field-7  PIC 9(5) COMP-1 VALUE 123.45.
     02  field-8  PIC 9(11) COMP VALUE 12398756983.
```

Figure 8-11 illustrates the record produced by the code segment shown above:

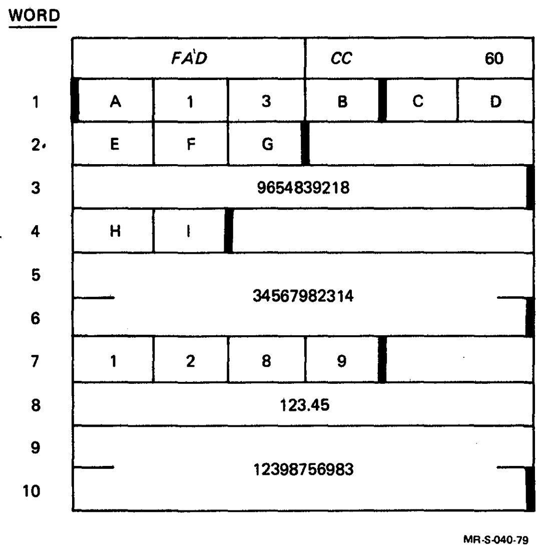


Figure 8-11 COBOL Fixed-Length SIXBIT

8.2.4 Variable-Length SIXBIT

This format is the same as fixed-length SIXBIT, except that the character count may vary from record to record. The following diagram illustrates the format of variable-length SIXBIT records:

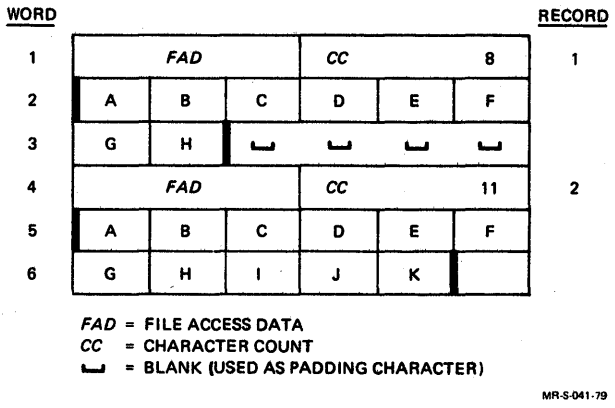


Figure 8-12 Variable-Length SIXBIT

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  PIC 9(7) COMP-1 VALUE  123.4567.
    02  field-2  PIC X(3) VALUE "A3C".
    02  field-3  PIC A(3) VALUE "DEF".
    02  field-4  PIC 9(3) VALUE -55.
    02  field-5  PIC 9(10) COMP VALUE 1234567809.
    02  field-6  PIC 9(11) COMP VALUE 98765432108.
    02  field-7  PIC X(2) VALUE "A2".
    02  field-8  PIC 9(5) COMP VALUE 32571.

01  record-2  DISPLAY-6.
    02  field-1  PIC 9(7) COMP-1 VALUE 1395.678.
    02  field-2  PIC X(3) VALUE "B5L".
    02  field-3  PIC A(3) VALUE "LMN".
    02  field-4  PIC 9(3) VALUE 79.
    02  field-5  PIC 9(10) COMP VALUE 8176596821.
    02  field-6  PIC 9(11) COMP VALUE 18976532150.
    02  field-7  PIC X(2) VALUE "M5".
    02  field-8  PIC 9(11) COMP VALUE 12357986183.

PROCEDURE DIVISION.

    WRITE record-1.
    WRITE record-2.
```

Figure 8-13 illustrates the record produced by the code segment shown on the previous page.

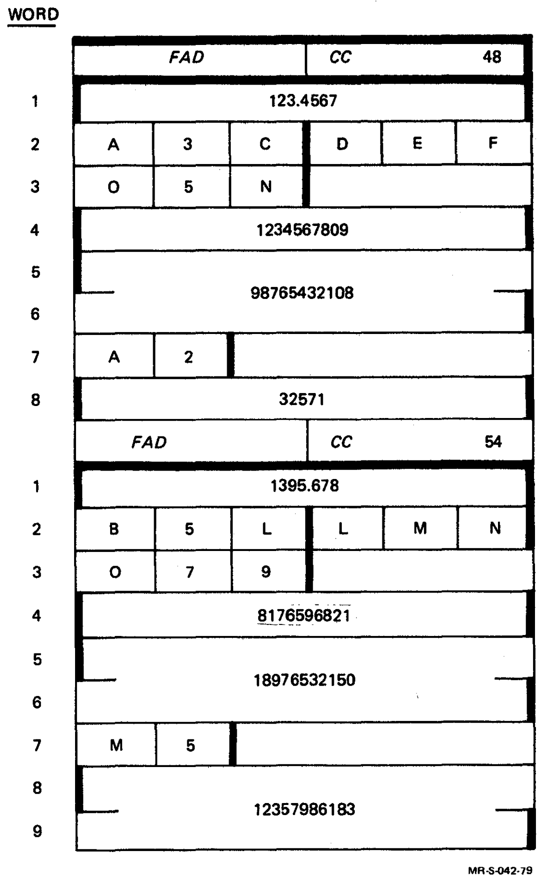


Figure 8-13 COBOL Variable-Length SIXBIT

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 may be either fixed or variable length, and may 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:

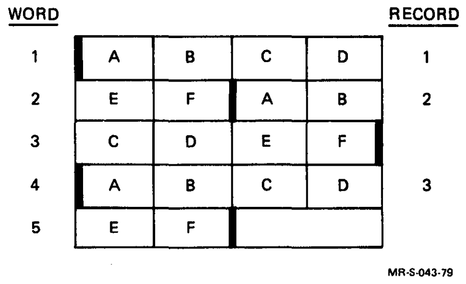


Figure 8-14 Fixed-Length EBCDIC

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) VALUE 123.
    02  field-2  PIC X(5) VALUE "ABCDE".
    02  field-3  PIC A(2) VALUE "LM".
    02  field-4  PIC 9(9) COMP-3 VALUE 137958795.
    02  field-5  PIC S9(6) COMP-3 VALUE -351235.
```

Figure 8-15 illustrates the record produced by the code segment shown above:

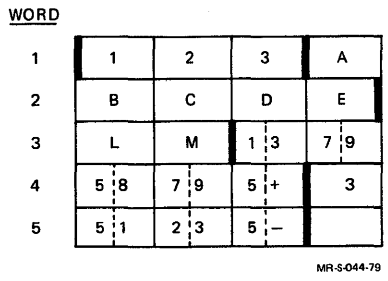


Figure 8-15 COBOL Fixed-Length EBCDIC

In a file written in variable-length EBCDIC format, the record lengths may 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:

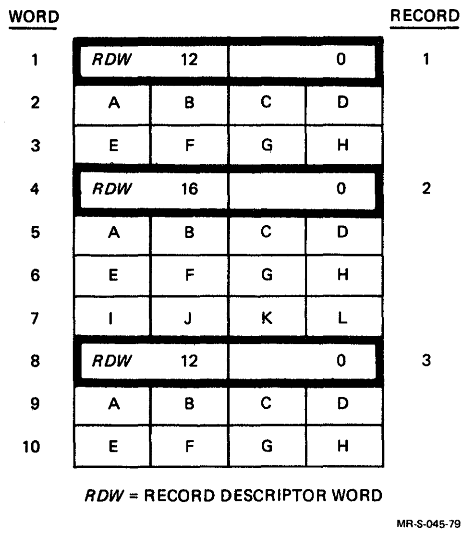
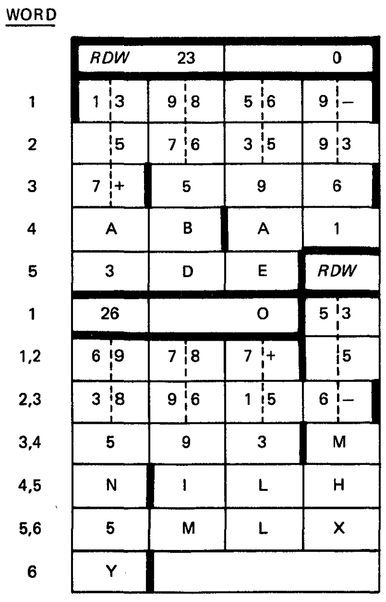


Figure 8-16 Variable-Length EBCDIC

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 VALUE -1398569.
    02  field-2  PIC S9(8) COMP-3 VALUE 57635937.
    02  field-3  PIC 9(3) VALUE 596.
    02  field-4  PIC A(2) VALUE "AB".
    02  field-5  PIC X(5) VALUE "A13DE".

01  record-2  DISPLAY-9.
    02  field-1  PIC S9(7) COMP-3 VALUE 5369787.
    02  field-2  PIC S9(8) COMP-3 VALUE -53896156.
    02  field-3  PIC 9(3) VALUE 593.
    02  field-4  PIC A(2) VALUE "MN".
    02  field-5  PIC X(8) VALUE "ILH5MLXY".

PROCEDURE DIVISION.

    WRITE record-1.
    WRITE record-2.
```

Figure 8-17 illustrates the record produced by the code segment shown above:



WORD				
	RDW	23		0
1	1 ¦ 3	9 ¦ 8	5 ¦ 6	9 ¦ –
2	¦ 5	7 ¦ 6	3 ¦ 5	9 ¦ 3
3	7 ¦ +	5	9	6
4	A	B	A	1
5	3	D	E	*RDW*
1	26		O	5 ¦ 3
1,2	6 ¦ 9	7 ¦ 8	7 ¦ +	¦ 5
2,3	3 ¦ 8	9 ¦ 6	1 ¦ 5	6 ¦ –
3,4	5	9	3	M
4,5	N	I	L	H
5,6	5	M	L	X
6	Y			

MR-S-046-79

Figure 8-17 COBOL Variable-Length EBCDIC

Fixed-length EBCDIC records may 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 magtape 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 RECORDS.
01   record-1   DISPLAY-9.
     02 field-1   PIC 9(3) VALUE "194".
     02 field-2   PIC X(5) VALUE "BDEFG".
     02 field-3   PIC A(2) VALUE "MN".
     02 field-4   PIC 9(5) COMP-3 VALUE 13796.
     02 field-5   PIC S9(4) COMP-3 VALUE 1985.

02   record-2   DISPLAY-9.
     02 field-1   PIC 9(3) VALUE "762".
     02 field-2   PIC X(5) VALUE "LANBH".
     02 field-3   PIC A(2) VALUE "AB".
     02 field-4   PIC 9(5) COMP-3 VALUE 76543.
     02 field-5   PIC S9(4) COMP-3 VALUE -9764.

PROCEDURE DIVISION.

     WRITE record-1.
     WRITE record-2.
```

Figure 8-18 illustrates the record produced by the code segment shown above:

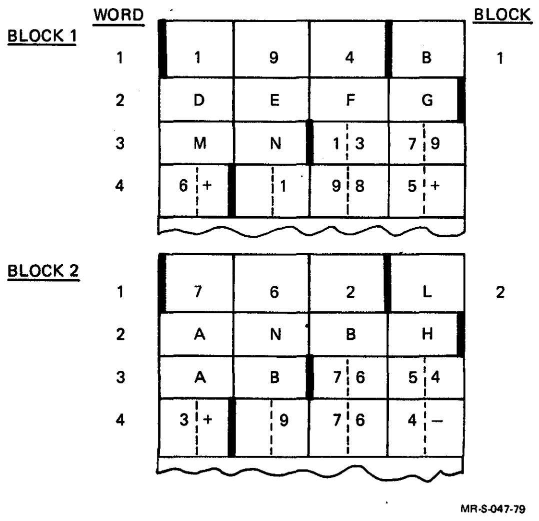


Figure 8-18 COBOL Blocked Fixed-Length EBCDIC

Variable-length EBCDIC records may be blocked as well. In this file format, the record length may 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:

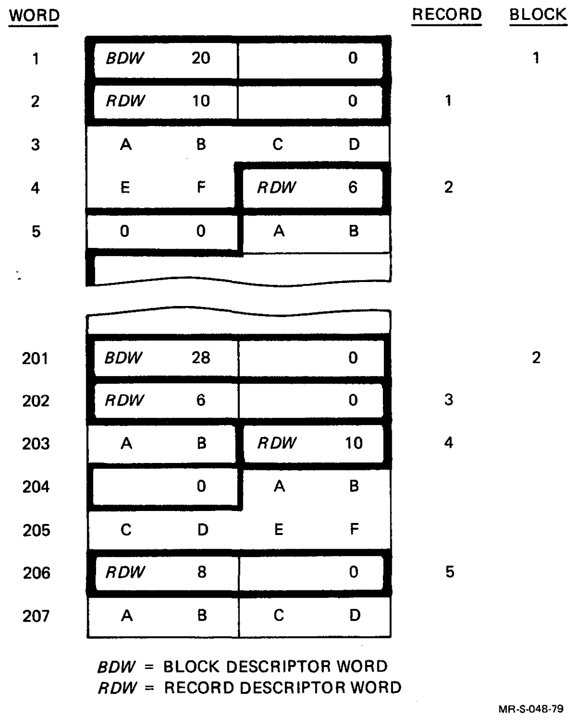


Figure 8-19 Blocked Variable-Length EBCDIC

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 RECORDS.
01  record-1  DISPLAY-9.
    02  field-1  PIC S9(7) COMP-3 VALUE +9356127.
    02  field-2  PIC 9(7) COMP-3 VALUE 3987156.
    02  field-3  PIC 9(3) VALUE "198".
    02  field-4  PIC A(2) VALUE "MN".
    02  field-5  PIC S9(9) COMP-3 VALUE -569138279.
    02  field-6  PIC X(6) VALUE "ABCDEF".

01  record-2  DISPLAY-9.
    02  field-1  PIC S9(7) COMP-3 VALUE -3295865.
    02  field-2  PIC 9(7) COMP-3 VALUE 9378518.
    02  field-3  PIC 9(3) VALUE "196".
    02  field-4  PIC A(2) VALUE "AL".
    02  field-5  PIC 9(9) COMP-3 VALUE 569138279.
    02  field-6  PIC X(9) VALUE "ABCDEFGHI".

PROCEDURE DIVISION.

    WRITE record-1.
    WRITE record-2.
```

Figure 8-20 illustrates the record produced by the code segment shown on the previous page.

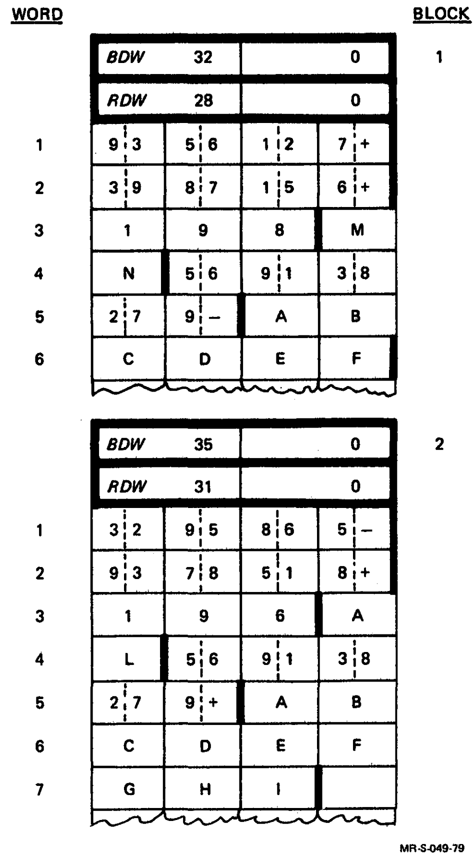


Figure 8-20 COBOL Blocked Variable-Length EBCDIC

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 may only be interpreted as COMPUTATIONAL and COMP1 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 on the following pages.

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 'DATA  FIL'.
01  BINARY-REC DISPLAY-7.
    02  field-1  PIC S9(10) COMP VALUE 12345678910.
    02  field-2  PIC S9(10) COMP-1 VALUE 1246.597892.
    02  field-3  PIC X(7) VALUE "ABCDE12".
    02  field-4  PIC 9(11) COMP VALUE 12345678954.
    02  field-5  PIC 9(3) VALUE "532".
    02  field-6  PIC 9(14) COMP VALUE 12345678954.
    02  field-7  PIC A(2) VALUE "LM".
```

Figure 8-21 illustrates the record produced by the code segment shown above:

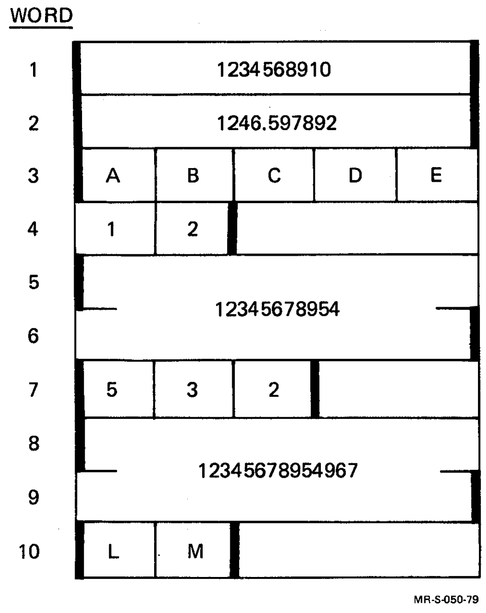


Figure 8-21 COBOL Standard Binary and ASCII Mixed-Mode Binary

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 'DATA  FIL'.
01  BINARY-REC DISPLAY-6.
    02  field-1  PIC S9(10) COMP VALUE 12345678910.
    02  field-2  PIC S9(10) COMP-1 VALUE 1234.592175.
    02  field-3  PIC X(7) VALUE "ABCDE12".
    02  field-4  PIC 9(11) COMP VALUE 12345678954.
    02  field-5  PIC 9(3) VALUE "532".
    02  field-6  PIC 9(14) COMP VALUE 12345678954967.
    02  field-7  PIC A(2) VALUE "LM".
```

Figure 8-22 illustrates the record produced by the code segment shown above:

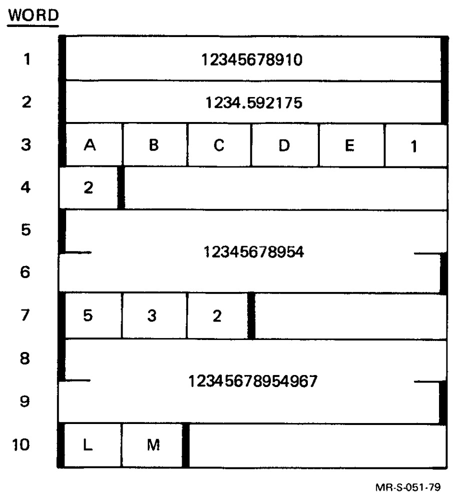


Figure 8-22 COBOL Standard Binary and SIXBIT Mixed-Mode Binary

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 'DATA  FIL'.
01  BINARY-REC DISPLAY-9.
    02  field-1  PIC S9(10) COMP VALUE 12345678910.
    02  field-2  PIC S9(10) COMP-1 VALUE 1246.597861.
    02  field-3  PIC X(7) VALUE "ABCDE12".
    02  field-4  PIC 9(11) COMP VALUE 12345678954.
    02  field-5  PIC 9(3) VALUE "532".
    02  field-6  PIC 9(14) COMP VALUE 12345678954967.
    02  field-7  PIC A(2) VALUE "LM".
    02  field-8  PIC S9(5) COMP-3 VALUE -72539.
    02  field-9  PIC 9(8) COMP-3 VALUE 36193586.
```

Figure 8-23 illustrates the record produced by the code segment shown above:

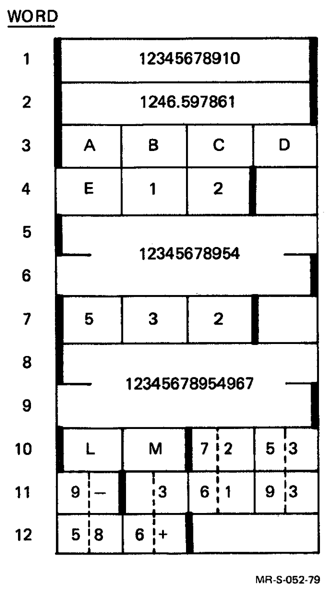


Figure 8-23 COBOL Standard Binary and EBCDIC Mixed-Mode Binary

8.3 FILE ORGANIZATION AND ACCESS

File organization refers to the manner in which the records are arranged in the file. Three types of file organization are available with COBOL-74: sequential, relative, and indexed-sequential. File organization is specified in a COBOL program by means of the ORGANIZATION clause.

COBOL-74 provides three methods by which files can be accessed: sequential, random, and dynamic. File access refers to the way in which records from a file are read and/or written. The method of access for a file is specified in a COBOL program by the ACCESS MODE clause. The chart below shows file organizations and the methods by which they can be accessed.

File Organization	Method of Access	ACCESS MODE
Sequential	Sequential	SEQUENTIAL
Relative	Sequential	SEQUENTIAL
	Random	RANDOM
	Sequential and Random	DYNAMIC
Indexed	Sequential	SEQUENTIAL
	Random	RANDOM
	Sequential and Random	DYNAMIC

In the following sections, file organizations are described 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, 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 may 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 RELATIVE FILES

Relative files are arranged like sequential files, but differ from sequential files in the method by which they are accessed and by the devices on which they must be stored. The following requirements must be fulfilled for a file to be relative:

1. It must be on a random-access device.

2. It must be blocked.

You can use the ACCESS MODE clause in the SELECT statement of the Environment Division to specify the access method.

You must also specify the RELATIVE KEY in the Environment Division. The data-name specified by the relative key must be described in the Working-Storage section as a COMPUTATIONAL item of 10 or fewer characters. Its picture can only contain the characters S and 9 (or their equivalent, such as S9(4)). The RELATIVE KEY specifies to the object-time system the location of a record relative to the beginning of the file. That is, the first record in the file is record 1 and the last record in the file is 1+n where n is the number of remaining records in the file.

Some records may be zero-length, that is, they do not have anything written in them because the file was created randomly. These records have RELATIVE 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 will be taken.

A relative file can be created in one of two ways - randomly or sequentially. To create a file randomly (that is, by writing into scattered or random records), you need only open the file, move an integer value into the RELATIVE KEY for each record to be written randomly, and write each record. To create a relative file sequentially, open the file for output and begin writing records. The RELATIVE KEY will default to the next record in the file, and the records will be entered sequentially. No zero-length records will be in the file if it is written sequentially.

8.5.1 Sequential Access Of Relative Files

A file with relative organization may still be accessed sequentially if you specify ACCESS MODE IS SEQUENTIAL in the File Control paragraph. Read operations on such a file will retrieve 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 file opened for input or I/O may be repositioned using the START statement. An existing record may be updated using the REWRITE statement, assuming the file was opened for I/O and the immediately preceding I/O operation was a READ. A sequential READ or WRITE will update the file's RELATIVE 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-existant record - this is logical End-of-File

2. A WRITE is made to a location containing a non-zero-length record

3. A REWRITE is made to a zero-length record, or a REWRITE is not the first I/O operation after a READ

8.5.2 Random Access Of Relative Files

A relative file may be accessed at scattered locations by specifying the clause ACCESS MODE IS RANDOM. In this case the record accessed is the one indicated by the current value of the RELATIVE KEY. The first record on the file is assigned the relative 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 RELATIVE KEY for the file. Non-zero-length records may be updated by the use of the REWRITE clause, assuming that the file is open for I/O and that the previous I/O operation was a READ to the file.

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
3. A REWRITE is made to a zero-length record, or the last I/O operation before the REWRITE was not a valid READ to the file

8.5.3 Dynamic Access Of Relative Files

Often you will want to access a file both randomly and sequentially. You may accomplish this by indicating that your file's ACCESS MODE IS DYNAMIC. If you specify this mode, you may read your relative file randomly in the normal way, then issue a READ NEXT command and switch to sequential access. This READ statement acts just as it would in sequential access mode, obtaining the next non-zero-length record. As in sequential mode, the RELATIVE KEY will be reset to indicate the relative location of the record just obtained. The first READ NEXT you issue will use the current value of the RELATIVE KEY as its starting point. You may alter this by using the START verb to position the record pointer in the file. You may also update records using the REWRITE verb, with the same considerations as before. Figure 8-24 presents an example program which positions the file pointer to a starting location and updates records sequentially thereafter.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT RELOUT ASSIGN TO DSK
     ORGANIZATION IS RELATIVE
     ACCESS MODE IS DYNAMIC
     RELATIVE KEY IS RELKEY.
          .
          .
          .
DATA DIVISION.
FILE SECTION.
FD RELOUT BLOCK CONTAINS 8 RECORDS DATA RECORD IS RELREC
     VALUE OF ID IS "RELFILDAT"
01 RELREC PIC X(80)
          .
          .
          .
WORKING-STORAGE SECTION.
77 RELKEY PIC 9(10) VALUE IS 1.
          .
          .
PROCEDURE DIVISION.
START.
     OPEN INPUT-OUTPUT RELOUT.
          .
          .
          .
UPDATE.
     MOVE 5 TO RELKEY.
     START RELOUT, INVALID KEY GO TO STRT-ERR.
     READ RELOUT NEXT, AT END GO TO FINISH.
          .
          .
          .
     REWRITE RELREC, INVALID KEY GO TO ERROR.
     GO TO UPDATE.
          .
          .
          .
FINISH.
     CLOSE RELOUT, STOP RUN.
ERROR.
     DISPLAY "ERROR REPLACING RECORD", DISPLAY RELREC.
     GO TO FINISH.
STRT-ERR.
     DISPLAY "ERROR IN START - KEY=", RELKEY.
     GO TO FINISH.
```

Figure 8-24 Statements Used to Sequentially Access a Relative File

A relative file can be treated as a sequential file. That is, you can declare its ACCESS MODE as SEQUENTIAL and read or write the file sequentially. However, the file cannot be read or written randomly when it has been declared as ACCESS MODE SEQUENTIAL. If you wish to be allowed to access the file both randomly and sequentially you should specify the ACCESS MODE IS DYNAMIC option.

8.6 INDEXED-SEQUENTIAL FILES

Indexed-sequential files (also called ISAM files) 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:

- o payroll (key is employee number)
- o inventory control (key is part number)
- o production control (key is job or batch number)

An indexed-sequential file consists of two files: the data file containing the actual data and the index file containing pointers to record keys within the 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 5.9, ISAM - Indexed-Sequential File Maintenance Program.) ISAM then copies the data from the sequential file and creates a data file and an index file to reference the 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 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 data file and one for the index file.

8.6.1 Data File

The 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 will request additional blocks from the monitor as needed. If the monitor cannot allocate additional blocks (that is, because the user's quota on the file structure is exceeded or the system's limit was reached), an error message is issued.

The format of the data file is similar to that of relative 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.

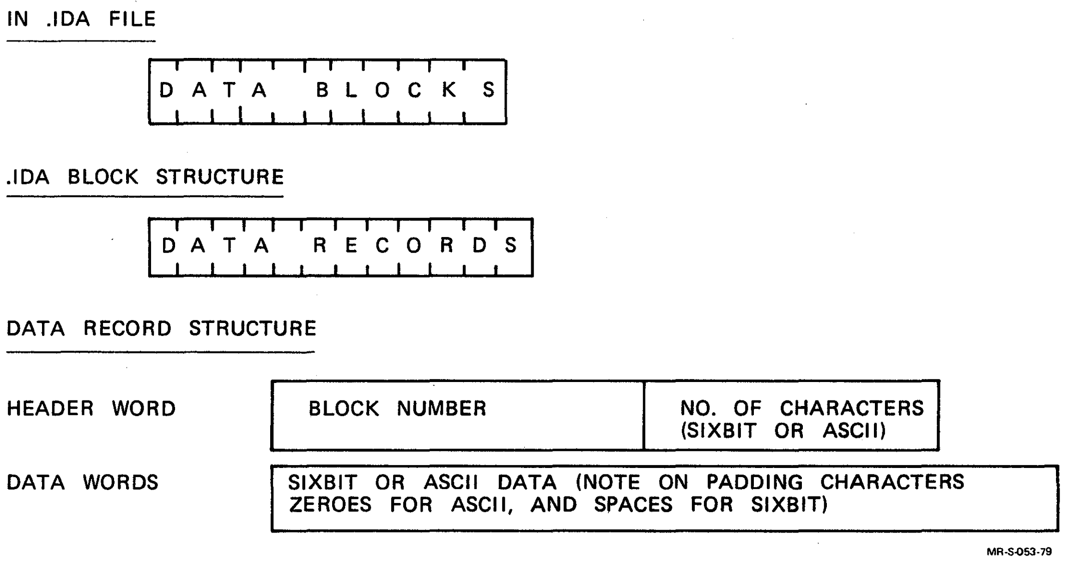

IN .IDA FILE
DATA BLOCKS
.IDA BLOCK STRUCTURE
DATA RECORDS
DATA RECORD STRUCTURE
HEADER WORD
BLOCK NUMBER
NO. OF CHARACTERS (SIXBIT OR ASCII)
DATA WORDS
SIXBIT OR ASCII DATA (NOTE ON PADDING CHARACTERS ZEROES FOR ASCII, AND SPACES FOR SIXBIT)
MR-S-053-79

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 data file and parameters specified by the user. It contains up 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 of the 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.

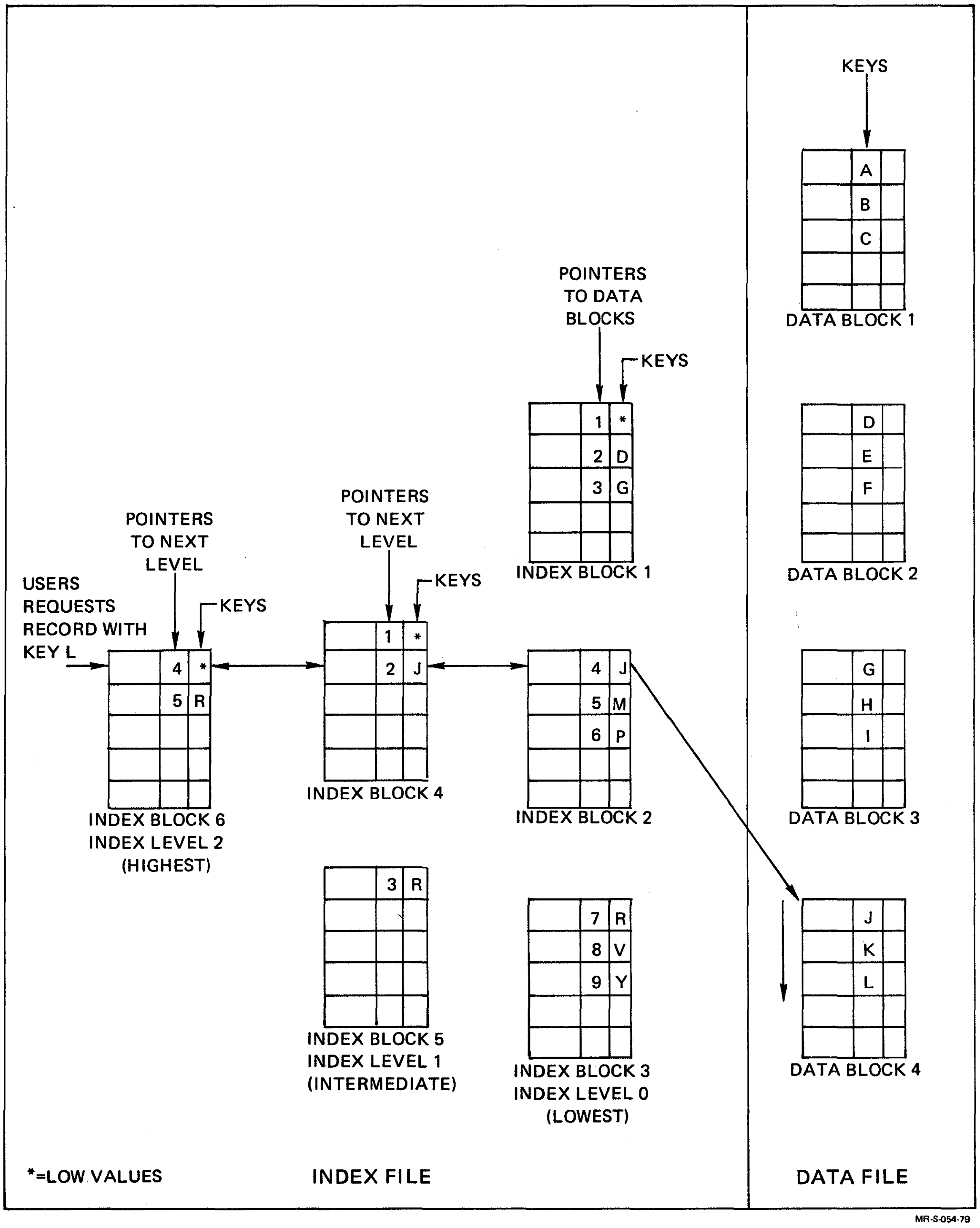


Figure 8-26 Locating a Record in an Indexed-Sequential File

The format of the index file is more complex than that of the data file. Figure 8-27 shows the structure of the index file.

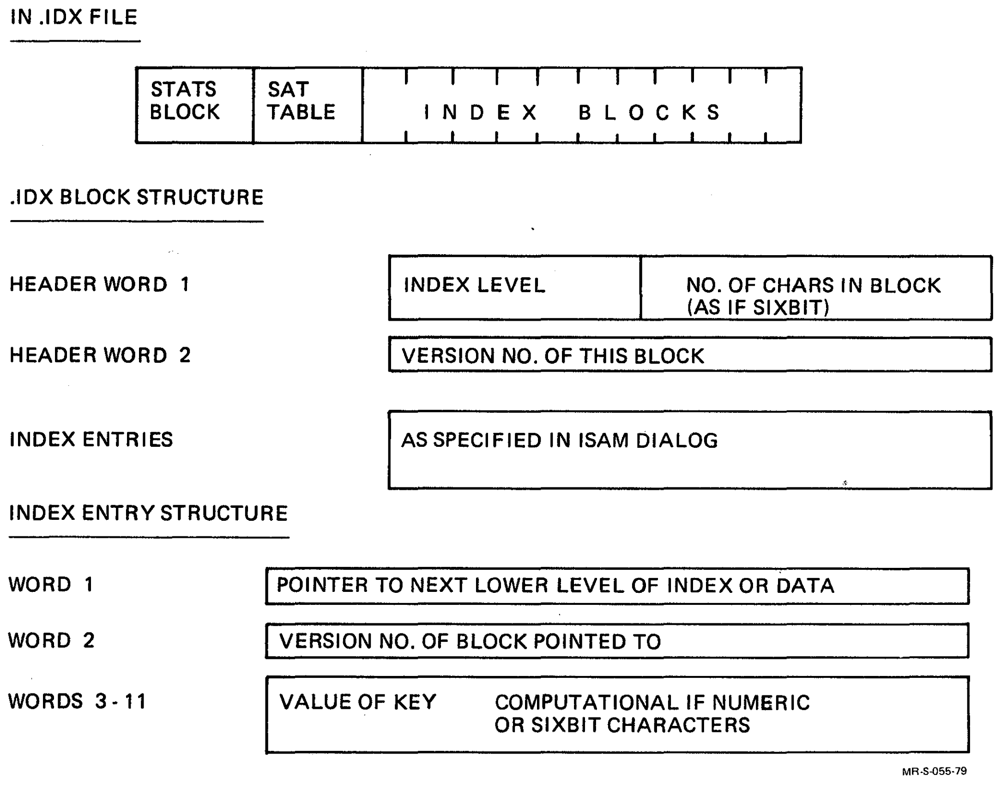

IN .IDX FILE
STATS BLOCK
SAT TABLE
INDEX BLOCKS
.IDX BLOCK STRUCTURE
HEADER WORD 1
INDEX LEVEL
NO. OF CHARS IN BLOCK (AS IF SIXBIT)
HEADER WORD 2
VERSION NO. OF THIS BLOCK
INDEX ENTRIES
AS SPECIFIED IN ISAM DIALOG
INDEX ENTRY STRUCTURE
WORD 1
POINTER TO NEXT LOWER LEVEL OF INDEX OR DATA
WORD 2
VERSION 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 will ensure 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 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 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 general, 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 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 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 will never need 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-74 simultaneous update facility allows sequential, relative, 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-74, 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-74 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 non-simultaneous-update users from accessing it until you are willing to release it. 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 exhaustion of 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.

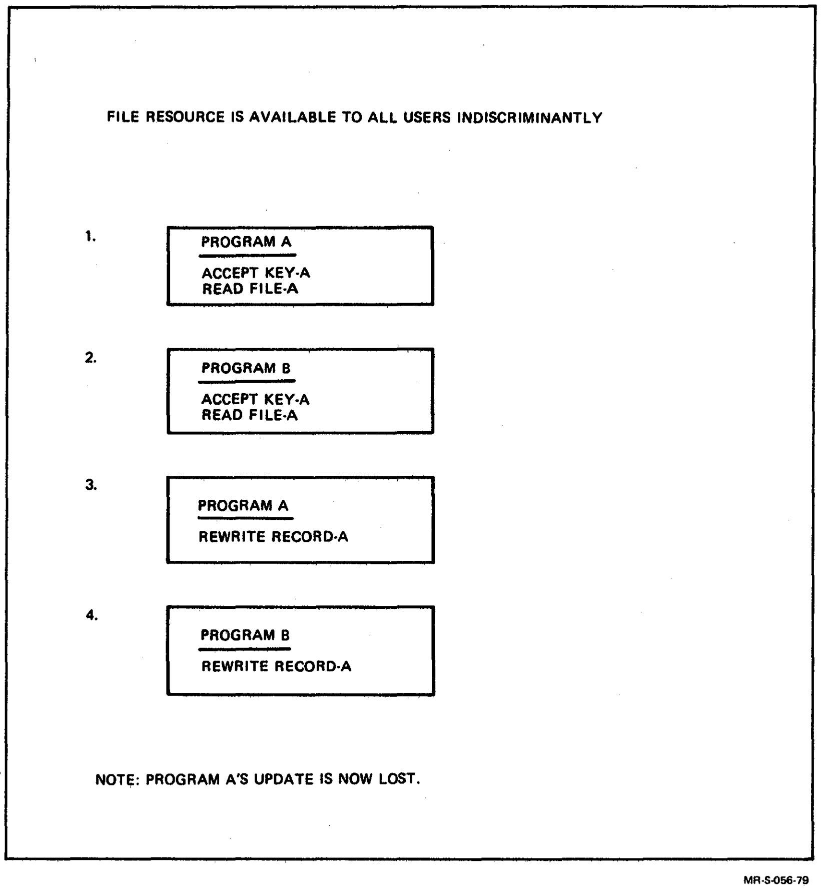
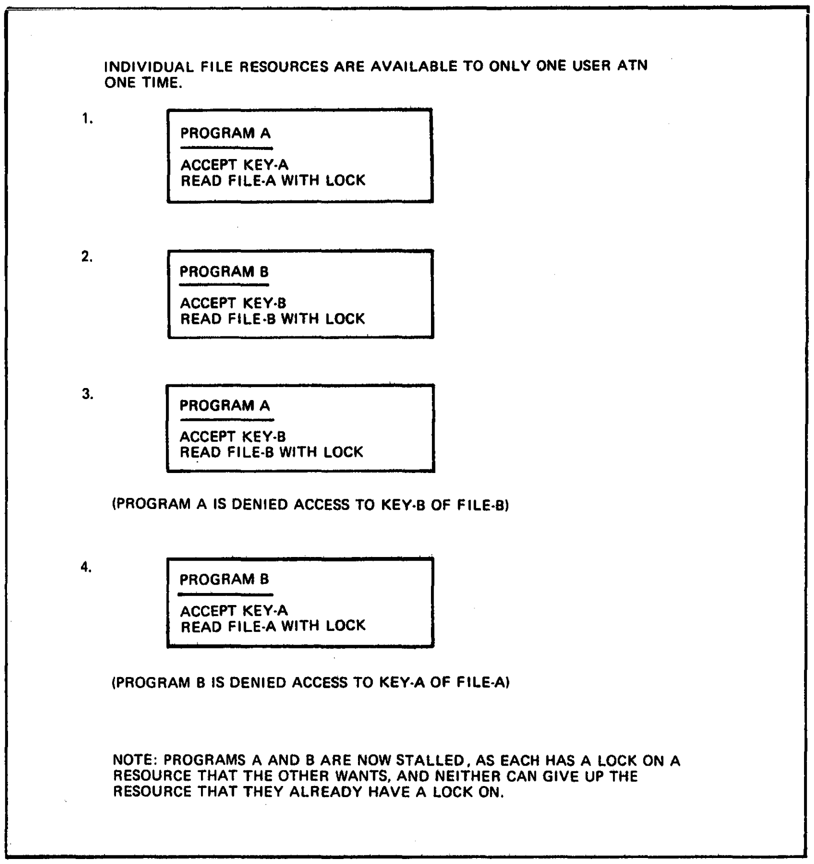


Figure 9-1 The Problem of Buried Update



INDIVIDUAL FILE RESOURCES ARE AVAILABLE TO ONLY ONE USER ATN 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 project the usage you want at both the file and record level. It also allows you to project the usage you will allow others to have while you have control of the file. A central clearing house in the COBOL-74 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 project file usage by specifying which of the COBOL-74 input/output verbs you will execute during your tenure of the file or record and which you will 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 will be stopped from doing so. See Figure 9-3 for an outline of how resources can be projected 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 Projecting 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.

```
     ┌ { INPUT  }  file-name-1 [ REVERSED       ] [ ,file-name-2 [ REVERSED       ] ] ...
     │ { OUTPUT }              [ WITH NO REWIND ] [              [ WITH NO REWIND ] ]
     │
     │                         ┌     { 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 ] filename-5 [ filename-6 ] ...
     │
     └ [ UNAVAILABLE statement-1 [ ,statement-2 ] ... ].
```

Figure 9-4 The OPEN Statement

SIMULTANEOUS UPDATE

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 either open the file under simultaneous update or for input only. Other uses will be 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. Conversely, if a file is open for normal processing, users attempting to open it under simultaneous update will be 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-74 I/O verbs you intend to execute must be entered following the key word FOR.

5. The COBOL-74 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.

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 will cause 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.
```

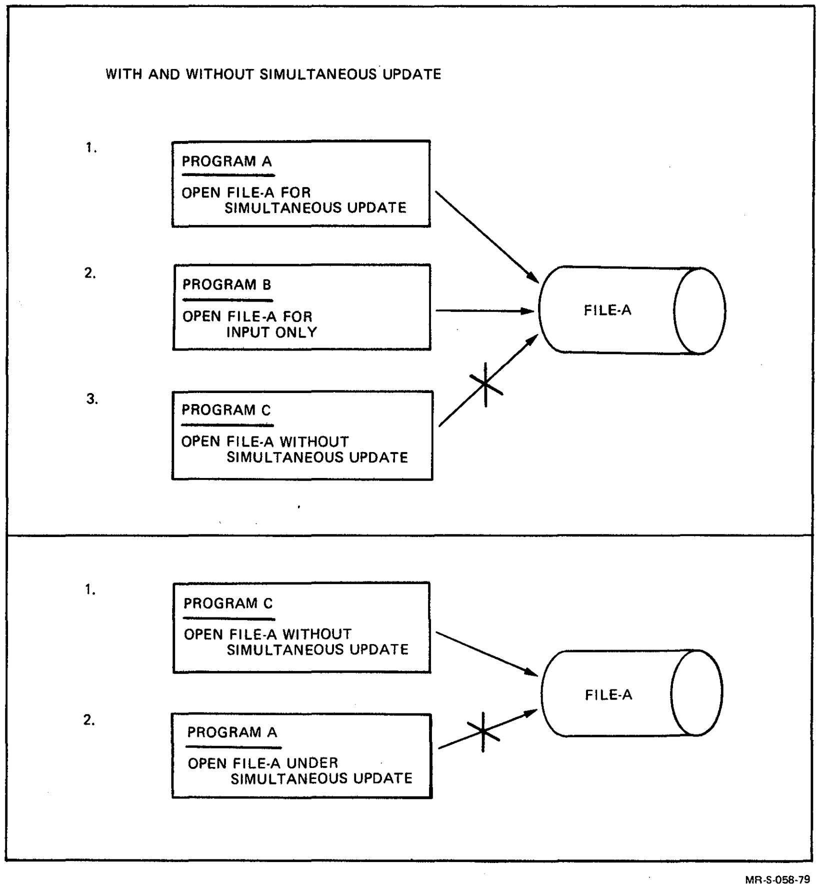

WITH AND WITHOUT SIMULTANEOUS UPDATE
1.
PROGRAM A
OPEN FILE-A FOR SIMULTANEOUS UPDATE
2.
PROGRAM B
OPEN FILE-A FOR INPUT ONLY
3.
PROGRAM C
OPEN FILE-A WITHOUT SIMULTANEOUS UPDATE
FILE-A
1.
PROGRAM C
OPEN FILE-A WITHOUT SIMULTANEOUS UPDATE
2.
PROGRAM A
OPEN FILE-A UNDER SIMULTANEOUS UPDATE
FILE-A
MR-S-058-79

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    }   }
                   { NEXT RECORD                       }

    FOR { READ         } [ UNTIL FREED ]
        { REWRITE      }
        { READ-REWRITE }
        { DELETE       }
        { WRITE        }
        { ANY VERB     }

  [ file-name-2 { RECORD   [ KEY { identifier-2 } ] }
                {                { literal-2    }   }
                { NEXT RECORD                       }

    FOR { READ         } [ UNTIL FREED ] ...  ]
        { REWRITE      }
        { READ-REWRITE }
        { DELETE       }
        { WRITE        }
        { ANY VERB     }

[ UNAVAILABLE statement-1 [ statement-2 ] ... ]
```

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, relative, indexed-sequential) see Sections 9.1.4, 9.1.5, and 9.1.6 respectively. (See also the COBOL-74 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 will abort 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, relative, 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 may 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 may, 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, will 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 will not be 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. If no key is provided, KEY 0 is assumed.

11. The value of the key may 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 may 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 may 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.

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.

```
     { file-name-1 { RECORD [ KEY { identifier-1 } ] }                                }
     {             {              { literal-1    }   }                                }
     {             { EVERY RECORD                    }                                }
FREE {                                                                                }
     {    [ file-name-2 { RECORD [ KEY { identifier-2 } ] } ]                         }
     {    [             {              { literal-2    }   } ]                         }
     {    [             { EVERY RECORD                    } ]                         }
     {                                                                                }
     { EVERY RECORD                                                                   }

[ NOT RETAINED statement-1 [ statement-2] ... ] .
```

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, relative, and indexed-sequential, see Sections 9.1.4, 9.1.5, and 9.1.6 respectively. (See also the COBOL-74 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 will occur 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 will cause 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** - 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 projects 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.

     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 may or may 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 **Sophisticated Access to Sequential Files** - 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 will not be 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 relative 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.

SIMULTANEOUS UPDATE

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 defeat 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 will normally be suspended if any of the requested files or records are unavailable. You will not be 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 will proceed 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 Relative Files

Accessing records in a relative 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 RELATIVE KEY specified in the FD for the file is used.
2. Positive keys, whether specified directly or via RELATIVE 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: that is, 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 relative file sequentially.

Example 9-4

```
     A.  RETAIN FILE-A FOR READ.

         READ FILE-A NEXT 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 RELATIVE-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 RELATIVE KEY for processing a chain of related records in a relative 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 will not be 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 may retain records for REWRITE, DELETE, and READ-REWRITE, in addition to READ, WRITE, and ANY VERB. You may 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 may 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 the user wishes)
                 .
                 .

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-74 Language Reference Material, Part 2 of this manual.

```
P R O G R A M   R E P E X M  COBOL-74 12(601) BIS
                      20-OCT-78  11:16        PAGE 1
REPORT.CBL    20-OCT-78  11:16

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-74 12(601) BIS
                     20-OCT-78  11:16               PAGE 2
REPORT.CBL    20-OCT-78  11:16

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,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-74 12(601) BIS
                 20-OCT-78  11:16                PAGE 3
REPORT.CBL    20-OCT-78  11:16

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
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-74 12(601) BIS
                     20-OCT-78  11:16              PAGE 4
REPORT.CBL     20-OCT-78  11:16

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 will occur 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-74 12(601) BIS
                  20-OCT-78  11:16                 PAGE 5
REPORT.CBL    20-OCT-78  11:16

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 will appear at  *
0175  *         the beginning of the report.                         *
0176  *                                                              *
0177  *     REPORT FOOTING (or RF) is a footing that will appear at  *
0178  *         the end of the report.                               *
0179  *                                                              *
0180  *     PAGE HEADING (or PH) is a page heading that will appear  *
0181  *         at the top of each page of the report.               *
0182  *                                                              *
0183  *     PAGE FOOTING (or PF) is a page footing that will appear  *
0184  *         at the bottom of each page.                          *
0185  *                                                              *
0186  *     CONTROL HEADING (or CH) is a heading that will appear    *
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 will appear    *
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-74 12(601) BIS
  20-OCT-78  11:16                              PAGE 6
REPORT.CBL    20-OCT-78  11:16

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 will be printed on      *
0240  *         line 1; otherwise it will be 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 will not be printed.                                   *
0252  *                                                            *
0253  * ************************************************************
```

```
0254  P R O G R A M   R E P E X M   COBOL-74 12(601) BIS
                    20-OCT-78  11:16                PAGE 7
REPORT.CBL     20-OCT-78  11:16

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 will be 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-74 12(601) BIS
                    20-OCT-78  11:16                PAGE 8
REPORT.CBL    20-OCT-78  11:16

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 will not be 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-74 12(601) BIS
                     20-OCT-78  11:16              PAGE 9
REPORT.CBL    20-OCT-78  11:16

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-74 12(601) BIS
                    20-OCT-78  11:16                 PAGE 10
REPORT.CBL    20-OCT-78  11:16

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.
```

```
     P R O G R A M   R E P E X M   COBOL-74 12(601) BIS
                   20-OCT-78  11:16                PAGE 11
REPORT.CBL    20-OCT-78  11:16

         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-74 12(601) BIS
                    20-OCT-78  11:16                 PAGE 12
REPORT.CBL    20-OCT-78  11:16

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-74 12(601) BIS
                  20-OCT-78  11:16                PAGE 13
REPORT.CBL    20-OCT-78  11:16

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          ACCEPT UNEDITED-DATE FROM 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-74 12(601) BIS
  20-OCT-78  11:16                                 PAGE 14
REPORT.CBL    20-OCT-78  11:16

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-74 12(601) BIS
                  20-OCT-78  11:16                PAGE 15
REPORT.CBL     20-OCT-78  11:16

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.

NO ERRORS DETECTED
```

CHAPTER 11

PROGRAM SEGMENTS, SUBPROGRAMS, AND OVERLAYS

You may 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.

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 writeable or nonwriteable, 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 nonwriteable; 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 writeable; 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 writeable 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-74 12(601) BIS
                    24-OCT-78  09:22                  PAGE 1
SEGMNT.CBL    24-OCT-78  09:22

IDENTIFICATION DIVISION.
PROGRAM-ID. SEGMNT.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER. DECSYSTEM-20
SEGMENT-LIMIT IS 25.

DATA DIVISION.

PROCEDURE DIVISION.
SECT1 SECTION 20.
CALL A.
SECT2 SECTION 65.
CALL A.
SECT3 SECTION 22.
CALL A.
SECT4 SECTION 20.
CALL A.
SECT5 SECTION 60.
CALL A.
SECT6 SECTION 30.
CALL A.
SECT7 SECTION 35.
CALL A.
SECT8 SECTION 35.
CALL A.
SECT9 SECTION 60.
CALL A.
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 writeable.

2. Segment 22 contains section SECT3; it is resident and writeable.

3. Segment 30 contains section SECT6. Since its segment number is above the SEGMENT-LIMIT but less than 50, it is nonresident and writeable; 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 writeable.

5. Segment 60 contains sections SECT5 and SECT9. Since its segment number is above 50, it is nonresident and nonwriteable; changes made to the segment are lost when it leaves and returns to memory.

6. Segment 65 contains section SECT2. It is nonresident and nonwriteable.

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 will also permit 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 which will perform 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 which wishes to do I/O. The CALL statement does not have to be executed to provide a link - in fact, it may be in such a position that it will never be executed. This requirement will of course be 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 may 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 may 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 may 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) may 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 in it 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 may not have a COBOL subprogram doing I/O which 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 the user 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 C74LIB.REL is searched automatically when LINK loads programs compiled with COBOL. This search occurs at the end of loading.

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 were 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 C74LIB.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 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
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
CBL0 saved
```

```
.RUN CBL0
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 virtual address 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 virtual address space. If the program can fit in your virtual 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.

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 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 writeable. 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.

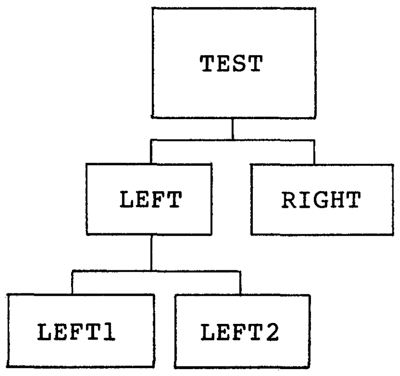


Figure 11-1 Example of an Overlay Structure

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 virtual address space 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 will not begin execution or it will fail 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 are:

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)

```
.TY SEGPRG.TTY
P R O G R A M   C B L 0         COBOL-74 12(600) BIS
        26-OCT-78  10:59        PAGE 1
CBL0.CBL    22-NOV-77  19:00

0001    ID DIVISION.
0002    PROGRAM-ID. CBL0.
0003    DATA DIVISION.
0004    WORKING-STORAGE SECTION.
0005    01      INFO.
0006           02      LEVMSG PIC X(15) USAGE IS DISPLAY-7 VALUE "We're at level ".
0007           02      LEVEL PIC 9V VALUE 0.
0008           02      PGMMSG PIC X(12) USAGE IS DISPLAY-7 VALUE " in program ".
0009           02      CALMSG PIC X(9) USAGE IS DISPLAY-7 VALUE " calling ".
0010           02      RETMSG PIC X(12) USAGE IS DISPLAY-7 VALUE "Returned to ".
0011           02      B PIC X(8) VALUE "        ".
0012    01      PGMNAM PIC X(6) VALUE "CBL0".
0013    01      ENDMSG PIC X(18) USAGE IS DISPLAY-7 VALUE "Execution ends in ".
0014    PROCEDURE DIVISION.
0015           DISPLAY LEVMSG,LEVEL,PGMMSG,PGMNAM.
0016           DISPLAY PGMNAM,CALMSG,"CBL2A".
0017           CALL CBL2A USING INFO.
0018           DISPLAY RETMSG,PGMNAM.
0019           DISPLAY PGMNAM,CALMSG,"CBL4".
0020           CALL CBL4 USING INFO.
0021           DISPLAY RETMSG,PGMNAM.
0022           DISPLAY PGMNAM,CALMSG,"CBL2B".
0023           CALL CBL2B USING INFO.
0024           DISPLAY RETMSG,PGMNAM.
0025           DISPLAY ENDMSG,PGMNAM.
0026           STOP RUN.

NO ERRORS DETECTED
```

```
S U B   C B L 1         COBOL-74 12(600) BIS
26-OCT-78  10:59        PAGE 1
CBL1.CBL    22-NOV-77  19:00

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-74 12(600) BIS
26-OCT-78  10:59        PAGE 1
CBL2.CBL    22-NOV-77  19:00

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-74 12(600) BIS
26-OCT-78  11:00        PAGE 1
CBL3.CBL    16-NOV-77  19:00

ID DIVISION.
PROGRAM-ID. CBL3.
DATA DIVISION.
WORKING-STORAGE SECTION.
01      PGMNAM PIC X(6) VALUE "CBL3".
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,B,LEVMSG,LEVEL,PGMMSG,PGMNAM.
        DISPLAY B,B,B,"CBL3 doesn't call anything".
        SUBTRACT 1 FROM LEVEL.
        GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 4         COBOL-74 12(600) BIS
26-OCT-78  11:00        PAGE 1
CBL4.CBL    16-NOV-77  19:00

0001    ID DIVISION.
0002    PROGRAM-ID. CBL4.
0003    DATA DIVISION.
0004    WORKING-STORAGE SECTION.
0005    01      PGMNAM PIC X(6) VALUE "CBL4".
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,LEVMSG,LEVEL,PGMMSG,PGMNAM.
0017           DISPLAY B,PGMNAM,CALMSG,"CBL1".
0018           CALL CBL1 USING INFO.
0019           DISPLAY B,RETMSG,PGMNAM.
0020           SUBTRACT 1 FROM LEVEL.
0021           GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 5         COBOL-74 12(600) BIS
26-OCT-78  11:00        PAGE 1
CBL5.CBL    16-NOV-77  19:00

0001    ID DIVISION.
0002    PROGRAM-ID. CBL5.
0003    DATA DIVISION.
0004    WORKING-STORAGE SECTION.
0005    01      PGMNAM PIC X(6) VALUE "CBL5".
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,LEVMSG,LEVEL,PGMMSG,PGMNAM.
0017            DISPLAY B,B,"CBL5 doesn't call anything".
0018            SUBTRACT 1 FROM LEVEL.
0019            GOBACK.

NO ERRORS DETECTED
```

```
S U B   C B L 6         COBOL-74 12(600) BIS
26-OCT-78  11:00        PAGE 1
CBL6.CBL    16-NOV-77  19:00

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
*TEST/LOG/LOGLEVEL:5                            ;Define TEST.LOG
*/ERRORLEVEL:5                                  ;Important msgs
*TEST/OVERLAY                                   ;Define TEST.OVL
*TEST/MAP                                       ;Define TEST.MAP
*CBL0,CBL1/LINK:TEST                            ;Root link
[LNKLMN Loading module CBL0]
[LNKLMN Loading module CBL1]
[LNKLMN Loading module OVRLAY]
[LNKLMN Loading module LILOWS]
[LNKLMN Loading module CON012]
[LNKLMN Loading module TRACED]
[LNKLMN Loading module USRDSL]
[LNKELN End of link number 0, name TEST]
*       /NODE:TEST      CBL2,CBL3/LINK:LEFT     ;Left branch
[LNKLMN Loading module CBL2]
[LNKLMN Loading module CBL3]
[LNKELN End of link number 1, name LEFT]
*               /NODE:LEFT      CBL5/LINK:LEFT1 ;Left-left branch
[LNKLMN Loading module CBL5]
[LNKELN End of link number 2, name LEFT1]
*               /NODE:LEFT      CBL6/LINK:LEFT2 ;Left-right branch
[LNKLMN Loading module CBL6]
[LNKELN End of link number 3, name LEFT2]
*       /NODE:TEST      CBL4/LINK:RIGHT         ;Right branch
[LNKLMN Loading module CBL4]
[LNKELN End of link number 4, name RIGHT]
*TEST/SAVE
*/E/GO
[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 12(600)         page 1
        Produced by LINK version 4(765) on  6-Dec-78 at 13:31:10

        Overlay no.     0       name    TEST
        Low  segment starts at       0 ends at    3106 length     3107 =   4P
        High segment starts at       0 ends at    3462 length     3463 =   4P
        Control Block address is   3047, length     30 (octal), 24. (decimal)
        441 words free in Low segment, 211 words free in high segment
        322 Global symbols loaded, therefore min. hash size is 358
        Start address is 400010, located in program CBL0

                *************

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.

                *************

CBL0    from DSK:CBL0.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:29:00
        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)

        CBL0        400022    Entry   Relocatable

                *************

CBL1    from DSK:CBL1.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:30:00
        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[1,4]        created by MACRO on 28-Aug-78 at 14:38:00
        Low  segment starts at    2170 ends at    2671 length      502 (octal),   322. (decimal)
        High segment starts at  400441 ends at  403462 length     3022 (octal),  1554. (decimal)

        BOUT    104000000051    Global  Absolute                CLOSF   104000000022    Global  Absolute
        ERJMP   320700000000    Global  Absolute                ERSTR   104000000011    Global  Absolute
        GCVEC   104000000300    Global  Absolute                GETOV.        402026    Entry   Relocatable
        GTJFN   104000000020    Global  Absolute                HALTF   104000000170    Global  Absolute
        INIOV.        402016    Entry   Relocatable             JFNS    104000000030    Global  Absolute
        LOGOV.        402617    Entry   Relocatable             OPENF   104000000021    Global  Absolute
        PBOUT   104000000074    Global  Absolute                PSOUT   104000000076    Global  Absolute
        REMOV.        402045    Entry   Relocatable             RMAP    104000000061    Global  Absolute
        RPACS   104000000057    Global  Absolute                RUNOV.        402065    Entry   Relocatable
        RUNTM   104000000015    Global  Absolute                SFPTR   104000000027    Global  Absolute
```

```
          LINK symbol map of      TEST    version 12(600)         page 2
OVRLAY
        SIN     104000000052    Global  Absolute                SOUT    104000000053    Global  Absolute
        TIME    104000000014    Global  Absolute                %OVRLA     400000037    Global  Absolute    Suppressed
        .FHSLF        400000    Global  Absolute    Suppressed  .OVRLA          2171    Entry   Relocatable
        .OVRLO          2176    Global  Relocatable             .OVRLU        402346    Entry   Relocatable
        .OVRWA          2175    Global  Relocatable

                *************

LILOWS  from SYS:C74LIB.REL[1,4] created by MACRO on 24-Oct-78 at  8:39:00

        Zero length module

                *************

CON012  from SYS:C74LIB.REL[1,4] created by MACRO on 24-Oct-78 at  8:39:00
        Low  segment starts at    2672 ends at    3036 length      145 (octal),   101. (decimal)

        CN.12           2672    Entry   Relocatable             COBST.          2672    Global  Relocatable
        GJ%OLD  100000000000    Global  Absolute    Suppressed  GJ%SHT       1000000    Global  Absolute    Suppressed
        GT%ADR        200000    Global  Absolute    Suppressed  JS%DIR   70000000000    Global  Absolute    Suppressed
        JS%GEN      70000000    Global  Absolute    Suppressed  JS%NAM    7000000000    Global  Absolute    Suppressed
        JS%PAF             1    Global  Absolute    Suppressed  JS%TYP     700000000    Global  Absolute    Suppressed
        PA%PRV     200000000    Global  Absolute    Suppressed

                *************

TRACED  from SYS:C74LIB.REL[1,4] created by MACRO on 24-Oct-78 at  8:39:00
        Low  segment starts at    3037 ends at    3046 length       10 (octal),     8. (decimal)

        BTRAC.          3042    Entry   Relocatable             C.TRCE          3037    Entry   Relocatable
        CBDDT.          3044    Entry   Relocatable             CNTRC.          3042    Entry   Relocatable
        HSRPT.          3042    Entry   Relocatable             PTFLG.          3045    Global  Relocatable
        SBPSG.          3042    Entry   Relocatable             SFOV.           3042    Entry   Relocatable
        TRPD.           3043    Entry   Relocatable             TRPOP.          3042    Entry   Relocatable

                *************

USRDSL  from SYS:C74LIB.REL[1,4] created by MACRO on 24-Oct-78 at  8:39:00

        Zero length module

                *************
```

```
Index to LINK symbol map of      TEST    version 12(600)         page 3

Name      Page     Name      Page     Name      Page     Name      Page

CBL0       1       CON012     2       OVRLAY     1       USRDSL     2
CBL1       1       LILOWS     2       TRACED     2
```

```
        LINK symbol map of      TEST    version 12(600) #1      page 4

        Overlay no.     1       name    LEFT
        Low  segment starts at    7107 ends at    7642 length      534 =   1P
        High segment starts at    3463 ends at    4466 length     1004 =   2P
        Control Block address is   7577, length     30 (octal), 24. (decimal)
        Path is 0
        93 words free in Low segment, 211 words free in high segment
        23 Global symbols loaded, therefore min. hash size is 26

                *************

CBL2    from DSK:CBL2.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:30:00
        Low  segment starts at    7107 ends at    7356 length      250 (octal),   168. (decimal)
        High segment starts at  403463 ends at  404226 length      544 (octal),   356. (decimal)

        CBL2          403465    Entry   Relocatable             CBL2A         403503    Entry   Relocatable
        CBL2B         403766    Entry   Relocatable

                *************

CBL3    from DSK:CBL3.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:30:00
        Low  segment starts at    7357 ends at    7576 length      220 (octal),   144. (decimal)
        High segment starts at  404227 ends at  404466 length      240 (octal),   160. (decimal)

        CBL3          404231    Entry   Relocatable

                *************
        LINK symbol map of      TEST    version 12(600) #2      page 5

Overlay no.     2       name    LEFT1
Low  segment starts at    7643 ends at   10110 length      246 =   1P
High segment starts at    4467 ends at    4712 length      224 =   1P
Control Block address is  10063, length     16 (octal), 14. (decimal)
Path is 0, 1
439 words free in Low segment, 211 words free in high segment
18 Global symbols loaded, therefore min. hash size is 21

        *************

CBL5    from DSK:CBL5.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:30:00
        Low  segment starts at    7643 ends at   10062 length      220 (octal),   144. (decimal)
        High segment starts at  404467 ends at  404712 length      224 (octal),   148. (decimal)

        CBL5          404471    Entry   Relocatable

                *************
                LINK symbol map of      TEST    version 12(600) #3      page 6

        Overlay no.     3       name    LEFT2
        Low  segment starts at    7643 ends at   10112 length      250 =   1P
        High segment starts at    4467 ends at    5204 length      516 =   1P
        Control Block address is  10065, length     16 (octal), 14. (decimal)
        Path is 0, 1
        437 words free in Low segment, 211 words free in high segment
        19 Global symbols loaded, therefore min. hash size is 22

                *************

CBL6    from DSK:CBL6.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:30:00
        Low  segment starts at    7643 ends at   10064 length      222 (octal),   146. (decimal)
        High segment starts at  404713 ends at  405204 length      272 (octal),   186. (decimal)

        CBL6          404715    Entry   Relocatable

                *************
                LINK symbol map of      TEST    version 12(600) #4      page 7

        Overlay no.     4       name    RIGHT
        Low  segment starts at    7107 ends at    7356 length      250 =   1P
        High segment starts at    3463 ends at    5454 length     1772 =   2P
        Control Block address is   7331, length     16 (octal), 14. (decimal)
        Path is 0
        273 words free in Low segment, 211 words free in high segment
        19 Global symbols loaded, therefore min. hash size is 22

                *************

CBL4    from DSK:CBL4.REL[4,70] created by COBOL-74 on  6-Dec-78 at 13:30:00
        Low  segment starts at    7107 ends at    7330 length      222 (octal),   146. (decimal)
        High segment starts at  405205 ends at  405454 length      250 (octal),   168. (decimal)

        CBL4          405207    Entry   Relocatable

                *************
```

```
Index to overlay numbers of TEST        version 12(600)         page 8

Overlay Page    Overlay Page    Overlay Page    Overlay Page

#0       3      #2       5      #3       6      #4       7
#1       4

Index to overlay names of TEST  version 12(600)

Name     Page   Name     Page   Name     Page   Name     Page

LEFT     4      LEFT2    6      RIGHT    7      TEST     3
LEFT1    5

[End of LINK map of     TEST]
```

CHAPTER 12

CALLING NON-COBOL SUBPROGRAMS

Some programming tasks are more conveniently accomplished 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 one or more identifiers, literals, or procedure-names.

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 Eexcution]
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 or DISPLAY-7 item, 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 7 for DISPLAY-7.

The second word of the descriptor is of the form:

bit 0	numeric flag
bit 1	signed number flag
bit 2	figurative constant flag
bit 3	literal flag
bits 4-11	reserved
bit 12	flag for Ps preceding decimal point in PICTURE
bits 13-17	number of decimal places (if bit 12 is 0), or number of Ps (if bit 12 is 1)
bits 18-35	number of bytes in the item

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
           Z 4,ARG]             ;Set up arg for SQRT
        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 will yield 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 Excution]
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-74 PROGRAMS

Normally, the code generated by the COBOL-74 compiler is adequately efficient. However, since there are certain COBOL-74 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-74 compiler will solve 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 may 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 will reflect, to some extent, practices which 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 will affect 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 economics. 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 will promote 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 will be 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, he/she 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 will 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 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 endure: 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 may 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 will use 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 will enable 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 will aid 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 may 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 will aid 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 will improve 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 criteria 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 relevant.

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 easily predictable.

13.1.5 Documentation

It is a good practice to document everything you have done during program optimization. You will want to improve other programs, and the notes you take for the first attempt will 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 and should 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 will 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 will guide you into understanding how to order your decision lists. For example:

Suppose P-1 was entered 1000 times, P-2 was entered 500 times, and these paragraphs are chosen via 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 up. Generally, if you can cut the CPU time, the elapsed time will also drop and the application will perform 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 and the monitor.

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 will improve 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 relative or ISAM reads and updates. Inspection of the blocking revealed that while the files were blocked to conserve disk space, large amounts of data was 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-74 offers you three different DISPLAY types and several COMPUTATIONALS. Each data type will offer 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 which 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 may 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 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 usage 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 existance of many redefined records with mixtures of EBCDIC and COMP-3 make 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) will have to be converted to ASCII if it is to be sent to one of these devices.

In memory, the usage of ASCII will make 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 existance of a "vertical form" (normally a line feed) character (or several such characters). Thus when reading ASCII files, it is necessary to read them 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 which can be moved via the block transfer instruction.

ASCII is the standard data type for "text files". Files created by editors which 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. Basically, 64 characters allows 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. It 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 usage 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 will be 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 corect data types for certain operations. Using COMPUTATIONAL for counters will work 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 will be 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, it will become double-word computational, and all arithmetic will be 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" which 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 is 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 may 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 which have proven efficient. Any of these can be demonstrated beneficial by writng short programs which execute these sequences a large number of times. It is also possible to look at the MACRO expansion of the program to see why things are different.

13.5.1 Alignment

When the addresses of the data items are known at compile time the compiler may generate efficient in-line code. This code may include the usage 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 may be called.

The simplest way to insure that data will be 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 Usage of Subscripts

Avoid the usage 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 via 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 here and it pays to eliminate it.

13.5.3 Incrementing Counters

COBOL-74 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 will never become negative. If they are not signed, additional instructions will be generated 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.

However, it suffers from the fact that it requires some information to be posted upon entry to a routine and cleared upon exit from that routine. COBOL-74 is fussy about the nesting of PERFORMs so that there is a 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 which 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.
               .
```

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 word being accomplished by each paragraph is small.

13.5.5 Use of the INSPECT Statement

Use of the INSPECT 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 INSPECT statement see Part 2, COBOL Language Reference Material.

13.5.6 Data Movement

This is just an observation on data movement. On a character-oriented machine, there is generally a machine instruction with a name something like MVC (for move characters). On such a system, there is a fixed cost for picking up the instruction, plus a variable cost which is a function of the number of characters. TOPS-10 and TOPS-20 act similarly, but the fixed cost is higher.

Especially if data conversion is implied (ASCII to SIXBIT), then it is more efficient to change all fields in a record with one move statement than to move the data field by field. If the compiler recognizes that data conversion is not necessary, and that the records are aligned, then efficient in-line code can be generated. Because the fixed cost to move any number of characters is higher on TOPS-10 and TOPS-20 than on some systems, programmers should 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 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 will only go 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

DIFFERENCES BETWEEN COBOL-68 AND COBOL-74

The terms COBOL-68 and COBOL-74, which are used in the following text, refer to DIGITAL's implementation of ANS-68 and ANS-74 COBOL, respectively. Any references to ANS COBOL will be made clear by the use of the initials "ANS".

COBOL-74 differs from COBOL-68 in the following ways:

1. A stroke (slash, "/", virgule) in the continuation area (seventh character position) of a line causes page ejection of the compilation listing. (The line is treated as a comment.) <1NUC (1) New feature to COBOL-74>

2. Two contiguous quotation marks may be used to represent a single quotation mark character in a nonnumeric literal. <1NUC (1) New feature.>

3. REMARKS paragraph is deleted. <1NUC (2) Function was replaced by the comment line.>

4. Continuation of Identification Division comment-entries must not have a hyphen in the continuation indicator area. <1NUC (2)>

5. PROGRAM COLLATING SEQUENCE clause specifies that the collating sequence associated with alphabet-name is used in nonnumeric comparisons. <1NUC (1) New feature.>

6. SPECIAL-NAMES paragraph: "L", "/", and "=" may not be specified in the CURRENCY SIGN clause. <2NUC (2) This restriction did not exist in X3.23-1968.>

7. Alphabet-name clause relates a user-defined name to a specified collating sequence or character code set (ANSI, native, or implementor-specified). <1NUC (1) New feature.>

8. Alphabet-name clause: the literal phrase specifies a user-defined collating sequence. <2NUC (1) New feature.>

9. All items which are immediately subordinate to a group item must have the same level-number. <1NUC (2)>

10. Object of a REDEFINES clause can be subordinate to an item described with an OCCURS clause, but must not be referred to in the REDEFINES clause with a subscript or an index. <1NUC (1) New feature.>

11. An asterisk used as a zero suppression symbol in a PICTURE clause and the BLANK WHEN ZERO clause may not appear in the same entry. <1NUC (2)>

12. Alphabetic PICTURE character-string may contain the character B. <1NUC (1) New feature.>

13. Stroke (/) permitted as an editing character. <1NUC (1) New feature.>

14. SIGN clause allows the specification of the sign position. <1NUC (1) New feature.>

15. In the Procedure Division a section may contain zero or more paragraphs and a paragraph may contain zero or more sentences. <1NUC (1) New feature.>

16. In relation and sign conditions, arithmetic expressions must contain at least one reference to a variable. <1NUC (2)>

17. Comparison of nonnumeric operands: If one of the operands is described as numeric, it is treated as though it were moved to an alphanumeric item of the same size and the contents of this alphanumeric item were then compared to the nonnumeric operand. <1NUC (3)>

18. Abbreviated combined relation condition: When any portion is enclosed in parentheses, all subjects and operators required for the expansion of that portion must be included within the same set of parentheses. <2NUC (2) No such restriction appeared in X3.23-1968.>

19. Abbreviated combined relation condition: If NOT is immediately followed by a relational operator, it is interpreted as part of the relational operator. <2NUC (2) In X3.23-1968, NOT was a logical operator in such cases.>

20. Class condition: The numeric test cannot be used with a group item composed of elementary items described as signed. <1NUC (3)>

21. In an arithmetic operation, the composite of operands must not contain more than 18 decimal digits. However, if your COBOL-74 compiler makes use of the Business Instruction Set, the maximum is 36 digits. <1NUC (2) X3.23-1968 specified limits only for ADD and SUBTRACT.>

22. ACCEPT identifier FROM DATE/DAY/TIME allows the programmer to access the date, day, and time. <2NUC (1) New feature.>

23. COMPUTE statement: the identifier series. <2NUC (1) New feature.>

24. DISPLAY statement: If the operand is a numeric literal, it must be an unsigned integer. <1NUC (2)>

25. DIVIDE statement: the INTO identifier series and the GIVING identifier series. <1NUC (2)>

26. DIVIDE statement: the remainder item can be numeric-edited. <2NUC (1) New feature.>

27. GO TO statement: the word TO is not required. <1NUC (1) X3.23-1968 requires the word TO.>

28. EXAMINE statement and the special register TALLY were deleted. <1NUC (2) Function was replaced by the INSPECT statement.>

29. INSPECT statement provides ability to count or replace occurrences of single characters or groups of characters. <1NUC (1) New feature.>

30. MOVE statement: A scaled integer item (i.e., the rightmost character of the PICTURE character-string is a P) may be moved to an alphanumeric or alphanumeric-edited item. <1NUC (1) New feature.>

31. MULTIPLY statement: the BY identifier series and the GIVING identifier series. <2NUC (1) New feature.>

32. PERFORM statement: There is no logical difference to the user between fixed and fixed overlayable segments. <1NUC (1) X3.23-1968 did not permit fixed overlayable segments to be treated the same as a fixed segment.>

33. PERFORM statement: Control is passed only once for each execution of a Format 2 PERFORM statement (i.e., an independent segment referred to by such a PERFORM is made available in its initial state only once for each execution of that PERFORM statement). <1NUC,1SEG (3)>

34. STOP statement: If the operand is numeric literal, it must be an unsigned integer. <1NUC (2)>

35. A data description entry with an OCCURS DEPENDING clause may be followed within that record only by entries subordinate to it. That is, only the last part of the record may have a variable number of occurrences. <2TBL (2) This rule did not appear in X3.23-1968.>

36. When a group item, having subordinate to it an entry that specifies Format 2 of the OCCURS clause, is referenced, only that part of the table area that is defined by the value of the operand of the DEPENDING phrase will be used in the operation. That is, the actual size of a variable length item is used, not the maximum size. <2TBL (2)>

37. The subject of the condition in the WHEN phrase of the SEARCH ALL statement must be a data item named in the KEY phrase of the table; the object of this condition may not be a data item named in the KEY phrase. <2TBL (2) X3.23-1968 specified that either the subject or object could be a data item named in the KEY phrase.>

38. SORT statement: COLLATING SEQUENCE phrase provides the ability to override the program collating sequence. <2SRT (1) New feature.>

39. No more than one file-name from a multiple file reel can appear in a SORT statement. <2SRT (2)>

40. Segment-numbers permitted in DECLARATIVES. <1SEG (1)>

41. ACCESS MODE IS DYNAMIC clause: provides ability to access a file sequentially or randomly in the same program. <2REL,2INX (1) New feature.>

42. ACTUAL KEY clause deleted. <(2)>

43. RELATIVE KEY clause added for relative organization. <1REL (1) New featue.>

44. FILE-LIMITS clause deleted. <(2)>

45. PROCESSING MODE clause deleted. <(2)>

46. ORGANIZATION IS RELATIVE clause. <1REL (2) New feature.>

47. ORGANIZATION IS SEQUENTIAL clause. <1SEQ (2) New feature.>

48. ORGANIZATION IS INDEXED clause. <1INX (2) New feature.>

49. MULTIPLE REEL/UNIT clause deleted. <(2)>

50. RESERVE...ALTERNATE AREAS deleted. <(2)>

51. RESERVE integer AREAS to allow the user to specify the exact number of areas to be used. <1SEQ,1REL,1INX (1) New feature.>

52. The data-name option of the LABEL RECORDS clause was deleted. <1SEQ,1REL,1INX (2) X3.23-1968 provided for user-defined label records.>

53. LINAGE clause permits programmer definition of logical page size. <2SEQ (1) New feature.>

54. CLOSE...FOR REMOVAL statement. <2SEQ (1) New feature.>

55. DELETE statement. <1REL (1) New feature.>

56. OPEN REVERSED positions the file at its end. <2SEQ (2)>

57. OPEN EXTEND statement adds records to an existing file. <2SEQ (1) New feature.>

58. The OPEN REVERSED statement applies to all devices that claim support for this function. <2SEQ (1) X3.23-1968 restricted the application of this phrase.>

59. READ statement: AT END phrase required only if no applicable USE AFTER ERROR/EXCEPTION procedure specified. <1SEQ,1REL,1INX (1) New feature.>

60. READ statement: INVALID KEY phrase required only if no applicable USE AFTER ERROR/EXCEPTION procedure specified. <1REL,1INX (1) New feature.>

61. READ...NEXT statement: used to retrieve the next logical record from a file when the access mode is dynamic. <2REL, 2INX (1) New feature.>

62. REWRITE statement. <1SEQ,1REL (1) New feature.>

63. SEEK statement was deleted. <(2)>

64. START statement provides for logical positioning within a relative or indexed file for sequential retrieval of records. <2REL, 2INX (1) New feature.>

65. USE statement: the label processing options were deleted. <1SEQ,1REL,1INX (2) X3.23-1968 provided for the processing of user-defined labels.>

66. USE...ERROR/EXCEPTION statement. <1SEQ,1REL,1INX (1) New feature.>

67. Recursive invocation of USE procedures prohibited. <1SEQ,1REL,1INX (2)>

68. WRITE statement: INVALID KEY phrase required only if no applicable USE AFTER ERROR/EXCEPTION procedure specified. <1REL,1INX (1)>

69. WRITE statement: BEFORE/AFTER PAGE phrase provides ability to skip to top of a page. <1SEQ (1)>

70. WRITE statement: END-OF-PAGE phrase. <2SEQ (1) New feature.>

71. CALL identifier statement. <1IPC (1) New feature.>

Note A. (RELATIVE files)

The RANDOM file access method of COBOL-68 has been replaced by the RELATIVE file organization in COBOL-74. This means a number of syntactic changes, but in addition it means some important semantic changes as well.

In the Environment Division, the syntactic changes include the substitution of an ORGANIZATION IS RELATIVE clause for the old ACCESS IS RANDOM clause, and the substitution of the ACCESS IS SEQUENTIAL / RANDOM / DYNAMIC for the old PROCESSING IS SEQUENTIAL clause. The FILE LIMITS clause goes away. The ACTUAL KEY clause is replaced by the RELATIVE KEY clause, although the meaning of the key value is identical to that in COBOL-68.

The Data Division is unchanged.

The Procedure Division verbs are changed considerably. OPEN, CLOSE and the USE ON ERROR procedures are unchanged. The WRITE statement is unchanged in syntax, but its meaning is restricted to writing a record into an "empty" position in the file. If the record position in the file into which the record is being written is already "occupied", the WRITE must not alter the existing contents of the record position, but must instead take the INVALID KEY path (or execute a USE procedure). In order to change the contents of an "occupied" record position one either has to REWRITE it or DELETE and WRITE it. Attempting to DELETE or REWRITE a record position which is already "empty" causes the INVALID KEY path to be taken. In other words, each record position of the relative file must have an "occupied" state, which can be recognized by the object time I/O routines. There is also a START verb which can be used to position at or beyond a given record position. Then sequential READs or WRITEs may be done. The sequential READ is done with a READ NEXT statement, whereas the random READ is just a READ statement. The sequential READ uses the AT END phrase (which is optional) and the random READ uses the INVALID KEY phrase (also optional). Thus, there are not only many syntactic changes in existing verbs, but new verbs, and a markedly different approach to the file's contents.

Note B. (INDEXED files)

The INDEXED I/O module of COBOL-74 is fairly similar to that of COBOL-68. There are syntactic differences in the Environment Division and in the Procedure Division.

COBOL-68 had a SYMBOLIC KEY clause to designate the key used in READ, WRITE, REWRITE and DELETE statements. COBOL-74 does not have a SYMBOLIC KEY clause. The random READ statement has a "KEY IS identifier" phrase which supplies a key value. The WRITE statement uses key values from the record being written, and the DELETE and REWRITE statements must follow a successfully executed READ and use the "remembered" key from that operation.

COBOL-74 includes a START statement in the Procedure Division which positions the record pointer in the file specified. Also, the READ NEXT statement is used to do sequential reading through the existing records of the file.

Note C. (Segmentation and PERFORM rules)

In COBOL-74, sections in the Procedure Division can have segment numbers (called "priority numbers" in COBOL-68) that range from 00 to 99. Segments with numbers 50 and above are called "independent" segments. Also, the programmer can specify a SEGMENT LIMIT IS clause with a value between 00 and 49. This divides the segments with numbers below 50 into two groups. Thus all segments fall into one of three groups:

1. Below the segment limit, called "fixed permanent", that is, always resident.

2. From the segment limit to 49, called "fixed overlayable", that is, each segment number defines an overlay and the code in such a segment is brought into memory only as needed. Any GO TOs which have been ALTERED will retain their most recently set values when they are brought into memory.

3. From 50 up, called "independent", that is, each segment number defines an overlay and the code is brought into memory only as needed. Any GO TOs which have been altered will be reset each time the segment is brought into memory.

The restrictions on the ALTER and PERFORM verbs have not really changed from ANS-68 COBOL to ANS-74 COBOL but they have become more explicit. COBOL-74 implements the restrictions on the ALTER statement correctly (by either standard) but implements the restrictions on PERFORM in a manner different from either standard. COBOL-74 uses the segment-limit value as the dividing line for the PERFORM restrictions, whereas the standards use the segment number 50 as the dividing line. When you do not specify a segment limit value the compiler supplies 50 as the default, making the restrictions the same for COBOL-74 and the standards. However, when you do supply the segment limit value, COBOL-74 applies the rules in such a way as to make all overlayable segments behave the same.

(In the Journal of Development, the ALTER statement has been abolished and segment numbers are restricted to 00 to 49, hence all these rules go away, but the JOD change occurred after the ANS-74 COBOL standard was frozen for publication.)

Note D. (CALL and CANCEL rules)

There are many differences between COBOL-74's implementation of CALL and CANCEL and the ANS-74 COBOL standard.

1. The syntax is different for both statements in that COBOL-74 interprets a user-word as a program-name with or without quotes around it, whereas ANS-74 COBOL interprets a user-word as a data-name in which is stored the program-name.

2. In ANS-74 COBOL there is an ON OVERFLOW ... clause for handling instances in which there is insufficient memory space available to load the called subprogram. This does not exist in COBOL-74.

3. COBOL-74 allows alternate entry points to subprograms, not allowed in ANS-74 COBOL, and COBOL-74 uses the ENTER (MACRO/FORTRAN) statement to allow the user to call subprograms written in those languages.

4. The semantics are very different. COBOL-74 uses LINK to construct a tree-structured overlay scheme from user-supplied commands to LINK. When a subprogram is CALLed, the branch of the tree up to that subprogram is loaded along with the subprogram. Likewise, when a subprogram is CANCELled, the entire tree beyond that subprogram is cancelled. ANS-74

COBOL recognizes no such tree structure, and allows loading and cancelling to occur strictly on a subprogram basis. In addition, LINK allows more than one subprogram to be linked into a single overlay, with the effect that a cancel of one of the subprograms in the overlay results in a cancel of all subprograms in that overlay.

Note E. (COPY statement)

The double equal sign (==) is a syntactic element new to ANS-74 COBOL (but not new to users of COBOL-68 version 12) used to set off pseudo-text. This notation is used to delimit pieces of text which you wish to replace or insert with the COPY verb. It is not necessary to use the double equal sign if your text-string is a literal or a data-name, but if you wish to replace complex pieces of text the double equal sign will serve as a clear delimiter, and you may include the notation any time you wish without risk of confusing the compiler.

APPENDIX B

COBOL RESERVED WORDS

In the listing below, words not preceded by symbols are reserved in both ANSI-74 Standard COBOL and in DECsystem-10 and DECSYSTEM-20 COBOL. Words preceded by '*' are reserved in ANSI-74 Standard COBOL but not reserved in DECsystem-10 and DECSYSTEM-20 COBOL. Words preceded by '**' are reserved in DECsystem-10 and DECSYSTEM-20 COBOL but not reserved in ANSI-74 Standard COBOL. Reserved words may not be used as user-created names.

A

ACCEPT	ACCESS	ACTUAL
ADD	*ADDRESS	ADVANCING
AFTER	ALL	**ALLOWING
ALPHABETIC	ALSO	ALTER
ALTERNATE	AND	**ANY
ARE	AREA	AREAS
ASCENDING	**ASCII	ASSIGN
AT	AUTHOR	

B

BEFORE	BEGINNING	**BINARY
BLANK	BLOCK	BOTTOM
BY	BYTE	

C

**CALL	**CANCEL	CD
CF	CH	**CHANNEL
CHARACTER	CHARACTERS	**CLASS
*CLOCK-UNITS	CLOSE	COBOL
CODE	CODE-SET	COLLATING

COLUMN	COMMA	**COMMUNICATION
COMP	**COMP-1	**COMP-3
**COMPILE	COMPUTATIONAL	**COMPUTATIONAL-1
**COMPUTATIONAL-3	COMPUTE	CONFIGURATION
**CONSOLE	CONTAINS	CONTROL
CONTROLS	COPY	CORR
CORRESPONDING	**COUNT	**CURRENCY
CURRENT		

D

DATA	**DATABASE-KEY	**DATE
**DATE-COMPILED	DATE-WRITTEN	**DBKEY
DE	*DEBUG-CONTENTS	*DEBUG-ITEM
*DEBUG-LINE	*DEBUG-NAME	*DEBUG-SUB-1
*DEBUG-SUB-2	*DEBUG-SUB-3	DEBUGGING
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	DUPLICATES	DYNAMIC

E

**EBCDIC	EGI	ELSE
EMI	**EMPTY	ENABLE
END	END-OF-PAGE	ENDING
ENTER	**ENTRY	ENVIRONMENT
EOP	**EPI	EQUAL
**EQUALS	ERROR	ESI
**EVEN	EVERY	EXCEPTION

COBOL RESERVED WORDS

**EXCL	**EXCLUSIVE	EXIT
EXTEND		
F		
FD	FILE	FILE-CONTROL
FILE-STATUS	FILLER	FINAL
**FIND	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	INSPECT
INSTALLATION	INTO	INVALID
**INVOKE	IS	
J		
**JOURNAL	JUST	JUSTIFIED
K		
KEY	KEYS	
L		
LABEL	LAST	LEADING
LEFT	LENGTH	LESS

LIMIT	LIMITS	LINAGE
LINAGE-COUNTER	LINE	LINE-COUNTER
LINES	**LINKAGE	LOCK
LOW-VALUE	LOW-VALUES	
M		
**MACRO	**MEMBER	**MEMBERS
MEMORY	MERGE	MESSAGE
MODE	**MODIFY	MODULES
MOVE	MULTIPLE	MULTIPLY
N		
NATIVE	NEGATIVE	NEXT
NO	**NOMINAL	**NONE
NOT	NUMBER	NUMERIC
O		
OBJECT-COMPUTER	OCCURS	**ODD
OF	OFF	OMITTED
ON	**ONLY	OPEN
**OPT	OPTIONAL	OR
ORGANIZATION	**OTHERS	OUTPUT
OVERFLOW	**OWNER	
P		
PAGE	PAGE-COUNTER	**PARITY
**PDP-10	PERFORM	PF
PH	PIC	PICTURE
PLUS	**POINTER	POSITION
**POSITIONING	POSITIVE	PRINTING
**PRIOR	**PRIVACY	PROCEDURE
PROCEED	PROCEDURES	PROCESSING
**PROGRAM	PROGRAM-ID	**PROT
**PROTECTED		

Q

QUEUE	QUOTE	QUOTES

R

RANDOM	RD	READ
*READ-REWRITE	*READ-WRITE	RECEIVE
RECORD	**RECORDING	RECORDS
REDEFINES	REEL	REFERENCES
RELATIVE	RELEASE	REMAINDER
REMARKS	REMOVAL	**REMOVE
RENAMES	REPLACING	REPORT
REPORTING	REPORTS	RERUN
RESERVE	RESET	**RETAIN
**RETAINED	**RETR	**RETRIEVAL
RETURN	REVERSED	REWIND
REWRITE	RF	RH
RIGHT	ROUNDED	RUN
**RUN-UNIT		

S

SAME	**SCHEMA	SD
SEARCH	SECTION	SECURITY
SEGMENT	SEGMENT-LIMIT	SELECT
**SELECTIVE	SEND	SENTENCE
SEPARATE	**SEQUENCE	SEQUENTIAL
SET	**SETS	SIGN
**SIXBIT	SIZE	SORT
SORT-MERGE	SOURCE	SOURCE-COMPUTER
SPACE	SPACES	SPECIAL-NAMES
STANDARD	STANDARD-1	**STANDARD-ASCII
START	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
TAPE	TERMINAL	TERMINATE
TEXT	THAN	THROUGH
THRU	TIME	TIMES
TO	TOP	**TRACE
TRAILING	**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 C

ASCII, SIXBIT, AND EBCDIC COLLATING SEQUENCES AND CONVERSIONS

Table C-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 C-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 C-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 C-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 C-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 C-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 C-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 C-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	¢	134	\
113	.	056	.
114	<	074	<
115	(	050	(
116	+	053	+
117	|	174	|

Table C-3 (Cont.)
EBCDIC Collating Sequence and Conversion to ASCII

EBCDIC code	EBCDIC character	ASCII code	ASCII character
120	&	046	&
121		134	\
122		134	\
123		134	\
124		134	\
125		134	\
126		134	\
127		134	\
130		134	\
131		134	\
132	!	041	!
133	$	044	$
134	*	052	*
135	)	051	)
136	^	073	^
137		137	\
140	-	055	-
141	/	057	/
142		134	\
143		134	\
144		134	\
145		134	\
146		134	\
147		134	\
150		134	\
151		134	\
152		134	\
153	,	054	,
154	%	045	%
155		137	
156	>	076	>
157	?	077	?
160		134	\
161		134	\
162		134	\
163		134	\
164		134	\
165		134	\
166		134	\
167		134	\
170		134	\
171		140	`
172	:	072	:
173	#	043	#
174	@	100	@
175	'	47	'
176	=	075	=
177	"	042	"
200		134	\
201	a	141	a
202	b	142	b
203	c	143	c
204	d	144	d
205	e	145	e
206	f	146	f
207	g	147	g
210	h	150	h
211	i	151	i
212		134	\
213		134	\
214		134	\
215		134	\
216		134	\
217		134	\
220		134	\
221	j	152	j
222	k	153	k
223	l	154	l
224	m	155	m
225	n	156	n
226	o	157	o
227	p	160	p
230	q	161	q
231	r	162	r
232		134	\
233		134	\
234		134	\
235		134	\
236		134	\
237		134	\

Table C-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 D

ALTERNATE NUMERIC TEST

LIBOL as normally assembled will include the ANSI standard NUMERIC test. However, a 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 may have only one of the three following representations: a leading graphic sign ("+" or "-"), a trailing graphic sign, or a trailing embedded sign. Leading and trailing spaces and leading and trailing tabs are ignored. All other characters must be digits.

APPENDIX E

DEFINING LOGICAL NAMES UNDER TOPS-20

Most of the file specifications for the COBOL compiler and the utilities associated with COBOL-74 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 will 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 DECSYSTEM-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 may 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 may 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 DECSYSTEM-20 User's Guide for a description of how to determine what logical names are already defined.

Example:

```
        DEFINE PR:  <PAYROLL>
```

will allow you to type the following command to the COBOL-74 compiler:

```
        PR:FEDTAX=TESTFT.CBL
```

This command string will take 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
```

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 that follow. 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.

Alphabet-Name
A user-defined word in the SPECIAL-NAMES paragraph of the Environment Division that assigns a name to a specific character set and/or collating sequence.

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.

Alternate Record Key
A key, other than the prime record key, whose contents identify a record within an indexed file.

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 may 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 may 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 may 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 may 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 may 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 may 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.

Debugging Line
A debugging line is any line with 'D' in the indicator area of the line.

Debugging Section
A debugging section is a section that contains a USE FOR DEBUGGING statement.

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 may 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] ...  ] .
```

Dynamic Access

An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner (see Random Access), and obtained from a file in a sequential manner (see Sequential Access), during the scope of the same OPEN statement.

Editing Character
A single character or a fixed 2-character combination beloriging 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)
/	stroke (virgule, slash)

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 may 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 of Reference
The key, either prime or alternate, currently being used to access records within an indexed file.

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 may 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 may 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 may 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 may be terminated by an ESI, an EMI, or an EGI. A message may 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.)

Native Character Set
The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

Native Collating Sequence
The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

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 will be transferred after execution of the current statement is complete.

Next Executable Statement
The next statement to which control will be 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 may be formed from more restricted character sets.

Nonnumeric Literal
A character-string bounded by quotation marks. The string of characters may 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 may also contain a '+', '-', or other representation of an operational sign.

Numeric Literal
A literal composed of one or more numeric characters that also may 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 may 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 may 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 the user 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.
```

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 may 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 may 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.

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, either the prime record key or an alternate record 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

Relative File
A file with relative organization.

Relative Key
A key whose contents identify a logical record in a relative file.

Relative Organization
The permanent logical file structure in which each record is uniquely identified by an integer value grater than zero, which specifies the record's logical ordinal position in the file.

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 may 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.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
COMMUNICATION 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 may contain only the characters '0', '1', ..., '9'. A segment-number may 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 may 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 may 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 the user to satisfy the format of a clause or statement.

Variable
A data item whose value may 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

COBOL-74
Language Manual
AA-5059A-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________

Please cut along this line.

-- Fold Here --

-- Do Not Tear - Fold Here and Staple --

FIRST CLASS
PERMIT NO. 152
MARLBOROUGH, MA
01752

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
200 Forest Street MR1-2/E37
Marlborough, Massachusetts 01752