Burroughs

B 1700 Systems
COBOL

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A.

March, 1975

1057197

Burroughs

B 1700 Systems
COBOL

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. March, 1975 1057197

Burroughs believes that the information described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material. The information contained herein is subject to change. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this document should be forwarded using the Remarks Form at the back of the manual, or may be addressed directly to Systems Documentation, Technical Information Organization, TIO-Central, Burroughs Corporation, Burroughs Place, Detroit, Michigan 48232

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

ACKNOWLEDGEMENT

The information contained in this document is based on the COBOL language initially developed in 1959 and the updated COBOL68.

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

The authors and copyright holders of the copyrighted material used in this document,

> FLOW-MATIC (trademark of Sperry Rand Corporation), programming for the UNIVAC (R) I and II. Data Automation Systems, copyrighted 1958, 1959 by Sperry Rand Corp.; IBM Commercial Translator, form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27 A5260-2760, copyrighted 1960 by Minneapolis-Honeywell,

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

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

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

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

SECTION 1

INTRODUCTION

This manual provides a complete description of COBOL (COMMON BUSINESS ORIENTED LANGUAGE) as implemented for use on the Burroughs B 1700 system. This concept of COBOL embraces the adoption of the American National Standards Institute (ANSI) 1968.

ADVANTAGES OF COBOL

The long list of COBOL advantages is derived chiefly from its intrinsic quality of permitting the programmer to state the problem solution in English. The programming language reads much like ordinary English prose, and can provide automatic program and system documentation. When users adopt in-house standardization of elements within files, plus well-chosen data-names, before attempting to program a system, they obtain maximum documentational advantages of the language described herein.

To a computer user, the Burroughs COBOL offers the following major advantages:

a. Expeditious means of program implementation.
b. Accelerated programmer training and simplified retraining requirements.
c. Reduced conversion costs when changing from a computer of one manufacturer to that of another.
d. Significant ease of program modification.
e. Standardized documentation.
f. Documentation which facilitates non-technical management participation in data processing activities.
g. Efficient object program code.
h. Segmentation capability which sets the maximum allowable program size well in excess of any practical requirement.
i. Due to the incorporation of debugging language statements, a high degree of sophistication in program design is achieved.
j. A comprehensive source program diagnostic capability.

A program written in COBOL, called a source program, is accepted as input by the COBOL compiler. The compiler verifies that all rules outlined in this manual are satisfied, and translates the source program language into an ob-

ject program language capable of communicating with the computer and directing it to operate on the desired data. Should source corrections become necessary, appropriate changes can be made and the program recompiled. Thus, the source deck always reflects the object program being operationally executed.

PROGRAM ORGANIZATION

Every COBOL program must contain these four divisions in the following order:

```
IDENTIFICATION
ENVIRONMENT
DATA
PROCEDURE
```

The IDENTIFICATION DIVISION identifies the program. In addition, the programmer may include such optional pieces of information as the date compiled, and programmer's name for documentation purposes. This division is completely machine-independent and thus does not produce object code.

The ENVIRONMENT DIVISION specifies the equipment being used. It contains computer descriptions and deals, to some extent, with the files the program will use.

The DATA DIVISION contains file and record descriptions describing the data files that the object program is to manipulate or create, and the individual logical records which comprise these files. The characteristics or properties of the data are described in relation to a standard data format rather than an equipment-oriented format. Therefore, this division is to a large extent computer-independent. While compatibility among computers cannot be absolutely assured, careful planning in the data layout will permit the same data descriptions, with minor modification, to apply to more than one computer.

The PROCEDURE DIVISION specifies the steps that the user wishes the computer to follow. These steps are expressed in terms of meaningful English words, statements, sentences, and paragraphs. This division of a COBOL program is often referred to as the "program" itself. In reality, it is only part of the total program, and is insufficient by itself to describe the entire program. This is true because repeated references must be made (either explicitly or implicitly) to information appearing in the other divisions. This division, more than any other, allows the user to express his/her thoughts in meaningful English. Concepts of verbs to denote actions, and sentences to describe procedures, are basic, as is the use of conditional statements to provide alternative paths of action.

A program written in COBOL is called the source program, and is accepted as input by the B 1700 COBOL compiler. The compiler will verify that the rules presented in this manual have been followed and will generate an object program in machine code, ready to be executed. Due to the speed of compilation, no object deck is supplied. Instead, the object program is placed on the disk, and may be dumped on a magnetic tape for back-up storage. Should changes become necessary, the source deck is corrected and a new compilation run made. Thus, the source deck always reflects the object program being executed.

SECTION 2

LANGUAGE FORMATION

GENERAL

As stated in section 1, COBOL is a language based on English, and is composed of words, statements, sentences, paragraphs, etc. The following paragraphs define the rules to be followed in the creation of this language. The use of the different constructs formed from the created words is covered in subsequent sections of this document.

CHARACTER SET

The COBOL character set for this system consists of the following 53 characters:

0 - 9		.	period or decimal point
A - Z		;	semicolon
	blank or space	"	quotation mark
+	plus sign	(	left parenthesis
-	minus sign or hyphen	)	right parenthesis
*	asterisk	>	greater than symbol
/	slash (virgule)	<	less than symbol
=	equal sign	:	colon
$	currency sign	@	"at" sign
,	comma		

Characters Used for Words

The character set for words consists of the following 37 characters:

0 - 9	
A - Z	
-	(hyphen)

Punctuation Characters

The following characters may be used for program punctuation:

@	"at" sign		space or blank
"	quotation mark	.	period
(	left parenthesis	,	comma (see note below)
)	right parenthesis	;	semicolon

NOTE

Commas may be used between statements, at the programmer's discretion, for enhanced readability of the source program. Use of these characters implies that a following statement is to be included as a portion of an entire statement.

Characters Used in Editing

The COBOL compiler accepts the following characters in editing:

$	currency sign	+	plus
*	asterisk (check protect)	-	minus
,	comma	CR	credit
.	period	DB	debit
B	space or blank insert	Z	zero suppress
0	zero insert		

Characters Used in Formulas

The COBOL compiler accepts the following characters in arithmetic expressions:

+	addition	**	exponentiation
-	subtraction	(	left parenthesis
*	multiplication	)	right parenthesis
/	division		

Characters Used in Relations

The COBOL compiler accepts the following characters in conditional relations:

=	equal sign
<	less than symbol
>	greater than symbol

DEFINITION OF WORDS

A word is created from a combination of not more than 30 characters, selected from the following:

A through Z
0 through 9
- hyphen

A word is ended by a space, or by a period, comma, or semicolon. A word may not begin or end with a hyphen. (A literal constitutes an exception to these rules, as explained later.)

Types of Words

COBOL contains the following word types:

a. Nouns.
b. Verbs.
c. Reserved words.

Nouns

Nouns are divided into ten special categories:

- File-name
- Record-name
- Data-name
- Condition-name
- Procedure-name
- Mnemonic-name
- Index-name
- Literal
- Figurative constant
- Special registers

Since the noun is a word, its length may not exceed 30 characters (exception: literals may not exceed 160 characters). For purposes of readability, a noun may contain one or more hyphens. However, the hyphen may neither begin nor end the noun (this does not apply to literals).

File-Name. A file-name is a name containing at least one alphabetic character assigned to designate a set of data items. The contents of a file are divided into logical records that in turn are made up of any consecutive set of data items.

Record-Name. A record-name is a noun containing at least one alphabetic character assigned to identify a logical record. A record can be subdivided into several data items, each of which is distinguishable by a data-name.

Data-Name. A data-name is a noun assigned to identify elements within a record or work area and is used in COBOL to refer to an element of data, or

to a defined data area containing data elements. Each data-name must contain at least one alphabetical character.

Condition-Name. A condition-name is the name assigned to a specific value, set of values, or range of values, within the complete set of values that a data item may assume. The data item itself is called a "conditional variable." The condition-name must contain at least one alphabetic character and must be unique, or be able to be referenced uniquely through qualification. A conditional variable may be used as a qualifier for any of its condition-names. If references to a conditional variable require indexing, subscripting, or qualification, then references to any of its condition-names also require the same combination of indexing, subscripting, or qualification. A condition-name is used in conditions as an abbreviation for the relation condition; its value is TRUE if the associated condition variable is equal to one of the set values to which that condition-name is assigned.

Procedure-Name. A procedure-name is either a paragraph-name or section-name, and is formulated according to noun rules. The exception is that a procedure-name may be composed entirely of numeric characters. Two procedure-names are identical only if they both consist of the same character strings. For example: procedure-names 007 and 7 are not equivalent.

Mnemonic-Name. The use of mnemonic-names provides a means of relating certain hardware equipment names to problem-oriented names the programmer may wish to use. See the discussion of SPECIAL-NAMES in section 5.

Index-Name. An index-name is a word with at least one alphabetic character that names an index associated with a specific table (refer to indexing in section 6). An index is a register, the contents of which represent the character position of the first character of an element of a table with respect to the beginning of the table.

Literals. A literal is an item of data which contains a value identical to the characters being described. There are three classes of a literal: numeric, non-numeric, and undigit.

Numeric Literal

A numeric literal is defined as an item composed of characters chosen from the digits 0 through 9, the plus sign (+) or minus sign (-), and the decimal point. The rules for the formation of a numeric literal are:

a. Only one sign character and/or one or more one decimal point may be contained in a numeric literal for use with Sterling. The leftmost decimal determines the scale.

NOTES

A comma must be substituted for the decimal point if the DECIMAL-POINT IS COMMA option is used (see SPECIAL-NAMES in the ENVIRONMENT DIVISION).

The implied USAGE of numeric literals is COMPUTATIONAL except when used with the verbs DISPLAY or STOP.

b. There must be at least one digit in a numeric literal.

c. The sign of a numeric literal must appear as the leftmost character. If no sign is present, the literal is defined as a positive value.

d. The decimal point may appear anywhere within the literal except for the rightmost character of a numeric literal. A decimal point within a numeric literal is treated as an implied decimal point. Absence of a decimal point denotes an integer quantity. (An integer is a numeric literal which contains no decimal point.)

e. A numeric literal used for arithmetic manipulations cannot exceed 160 digits. The following are examples of numeric literals.

```
13247
.005
+1.808
-.0968
7894.54
```

Non-Numeric Literal

A non-numeric literal may be composed of any allowable character. The beginning and end of a non-numeric literal are each denoted by a quotation mark. Any character enclosed within quotation marks is part of the non-numeric literal. Subsequently, all spaces enclosed within the quotation marks are considered part of the literal. Two consecutive quotation marks within a non-numeric literal cause a single quote to be inserted into the literal string. Four consecutive quotation marks will result in a single " literal.

A non-numeric literal cannot itself exceed 160 characters. Examples of non-numeric literals are:

Literal on Source Program Level	Literal Stored by Compiler
"ACTUAL SALES FIGURE"	ACTUAL SALES FIGURE
"-1234.567"	-1234.567
"""LIMITATIONS"""	"LIMITATIONS"
"ANNUAL DUES"	ANNUAL DUES
""""""	"
"A""B"	A"B

NOTE

Literals that are used for arithmetic computation must be expressed as numeric literals and must not be enclosed in quotation marks as non-numeric literals. For example, "-7.7" and -7.7 are not equivalent. The compiler stores the non-numeric literal as -7.7, whereas the numeric literal would be stored as 0077 if the PICTURE were S999V9 DISPLAY with the assumed decimal point located between the two sevens.

Undigit Literals

Binary 10 through 15 are represented as A through F and must be bounded by @ signs. For example, binary 11 would be expressed as @B@. An undigit literal cannot exceed 160 digits. Undigit literals are treated like numeric literals by the compiler.

Figurative Constant. A figurative constant is a particular value that has been assigned a fixed data-name and must never be enclosed in quotation marks except when the word, rather than the value, is desired. The figurative constant names and their meanings are:

ZERO ZEROS ZEROES	Represents the value 0, or one or more of the character 0, depending on the context.
SPACE SPACES	Represents one or more spaces (blanks).
HIGH-VALUE HIGH-VALUES	Represents the highest internal coding sequence (i.e., 999) value. When HIGH-VALUES are moved to a signed numeric computational field, the sign will be changed to a plus sign.
LOW-VALUE LOW-VALUES	Represents the lowest internal coding sequence (blanks) value. When LOW VALUES are moved to a signed numeric computational field, zeros will be moved into the field and the sign will be changed to a plus.

QUOTE
QUOTES

Represents one or more of the single character " (quotation mark). The word QUOTE or QUOTES does not have the same meaning in COBOL as the symbol ". For example, if "STANDARDS" appears as part of the COBOL source program, STANDARDS is stored in the object program. If, however, the full "STANDARDS" is desired in a DISPLAY statement, it can be achieved by writing QUOTE "STANDARDS" QUOTE, in which case the object program will print "STANDARDS". The same result can be obtained by writing """STANDARDS""" in the source program. Only the latter method can be used in MOVE statements and conditionals.

ALL

When followed by an integer numeric literal, a non-numeric literal, or a figurative constant, the word ALL represents a series of that literal. For example, if the COBOL statement is MOVE ALL literal TO ERROR-CODE, then the resultant ERROR-CODE would take on the following values:

ALL literal	Size of ERROR-CODE	Resulting value of ERROR-CODE
ALL "ABC"	7 characters	ABCABCA
ALL "3" or ALL 3	5 characters	33333
ALL "HI-LO"	12 characters	HI-LOHI-LOHI
ALL QUOTE	3 characters	"""
ALL SPACES	9 characters	(nine spaces)

NOTE

The use of ALL with figurative constants, as illustrated in the last two instances, is redundant. MOVE ALL SPACES and MOVE SPACES would yield the same result.

Special Registers. The B 1700 COBOL compiler provides the following five special PROCEDURE DIVISION register names:

a. TALLY.
b. TODAYS-DATE (Calendar).
c. TODAYS-NAME.
d. DATE (Julian).
e. TIME.

Tally

The special register TALLY is automatically provided by the COBOL compiler and has a defined length of five COMPUTATIONAL digits. The primary use of TALLY is

in conjunction with the EXAMINE statement; however, TALLY may be used as temporary storage or an accumulative area during the interim when EXAMINE... TALLYING... is not being executed in a program.

Todays-Date (Calendar)

This special register is included in each COBOL program and will contain the current date whenever TODAYS-DATE is requested as the sending field in a MOVE statement. Its format is made of three character pairs, each representing the month, day and year. For example, if the current date is Dec. 13th, 1971, the TODAYS-DATE register contains 121371. The function of TODAYS-DATE is to provide the programmer with a means of referring to the current date during program execution. TODAYS-DATE is maintained in COMPUTATIONAL form.

Todays-Name (Day of Week)

This special register is included in each COBOL program and will contain the current day of the week whenever TODAYS-NAME is requested as the sending field in a MOVE statement. TODAYS-NAME is returned left-justified in a nine-character field.

Date (Julian)

This special register is included in each COBOL program and will contain the current Julian date whenever DATE is requested as the sending field in a MOVE statement. Its format is YYDDD. For example, if the current date were January 1, 1975, the DATE register would contain 75001. The function of DATE is to save programmatic evaluation of TODAYS-DATE when Julian dates are required. DATE is maintained in COMPUTATIONAL form.

Time

Access to an internal clocking register reflecting the time of day is programmatically available whenever TIME is requested as the sending field of a MOVE statement. The contents of the TIME register will be maintained in hours, minutes, seconds and 10th of seconds. Its format is HHMMSST. For example, 10:30:51:8 would be stored as 1030518.

Verbs

Another type of COBOL word is a verb. A verb in COBOL is a single word that denotes action, such as ADD, WRITE, MOVE, etc. All allowable verbs in COBOL, with the exception of the word IF, are truly English verbs. The usage of the COBOL verbs takes place primarily within the PROCEDURE DIVISION.

Reserved Words

The third type of COBOL word is a reserved word. Reserved words have a specif function in the COBOL language and cannot be used out of context, or for any purpose other than the one for which they were intended. Reserved words are for syntactical purposes and can be divided into three categories:

a. Connectives.
b. Optional words.
c. Key words.

A complete list of reserved words in COBOL used by the compiler is included in appendix A.

Connectives. Connectives are used to indicate the presence of a qualifier or to form compound conditional statements. The connectives OF and IN are used for qualification. The connectives AND, AND NOT, OR, or NOT are used as logical connectives in conditional statements. The comma is used as a series connective to separate two or more operands.

Optional Words. Optional words are included in the COBOL language to improve the readability of the statement formats. These optional words may be included or omitted, as the programmer wishes. For example, IF A IS GREATER THAN B... is equivalent to IF A GREATER B..... Therefore, the inclusion or omission of the words IS and THAN does not influence the logic of the statement.

Key Words. The third kind of reserved words is referred to as being a key word. The category of key words includes the verbs and required words needed to complete the meaning of statements and entries. The category also includes words that have a specific functional meaning. In the example shown in the previous paragraph, the words IF and GREATER are key words.

LANGUAGE DESCRIPTION NOTATION

COBOL reference manuals have almost universally adopted a particular form of notation. This manual uses that notation as described in the paragraphs that follow.

Key Words

All underlined upper case words are key words and are required when the functions of which they are a part are utilized. Their omission will cause error conditions at compilation time. An example of key words is as follows:

IF data-name IS [NOT] {NUMERIC / ALPHABETIC}

The key words are IF, NOT, NUMERIC, and ALPHABETIC.

Optional Words

All upper case words not underlined are optional words and are included for readability only and may be included or excluded in the source program. In the example above, the optional word is IS.

Generic Terms

All lower case words represent generic terms which must be supplied in that format position by the programmer. Integer-1 and integer-2 are generic terms in the following example:

FILE-LIMIT IS integer-1 THRU integer-2

Braces

When words or phrases are enclosed in braces { }, a choice of one of the entries must be made. In reference to the key words example above, either NUMERIC or ALPHABETIC must be included in the statement.

Brackets

Words and phrases enclosed in brackets [ ] represent optional portions of a statement. If the programmer wishes to include the optional feature, he may do so by including the entry shown between brackets. Otherwise, it may be omitted. In terms of the example above, the word enclosed in brackets is optional. However, if the programmer wishes to distinguish between NUMERIC and ALPHABETIC, he must choose one of the words enclosed in braces.

Ellipsis

The presence of three consecutive periods (...) within any format indicates that the data immediately preceding the notation may be successively repeated, depending upon the requirements of problem solving.

Period

When a single period is shown in a format, it must appear in the same position whenever the source program calls for the use of that particular statement.

SECTION 3

CODING FORM

GENERAL

The format of the COBOL coding form (figure 3-1) has been defined by CODASYL, by ANSI, and by common usage. The B 1700 COBOL compiler accepts this standard format. Should program interchange be a major consideration, the user is directed to the ASA standard.

The same coding form format is used for all four divisions of a COBOL program. These divisions must appear in proper order: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE.

SEQUENCE FIELD (CARD COLUMNS 1-6)

The sequence field may be used to sequence the source program. Normally, a numeric sequence is used; however, the B 1700 compiler allows any combination of characters. A warning message is given if there is a sequence error. The B 1700 compiler provides for insertion or replacement of card images during compilation, controlled by the sequence field. (See section on "COBOL COMPILER CONTROL," section 11.)

CONTINUATION INDICATOR (COLUMN 7)

Column 7 has several functions as follows:

a. A $ symbol in column 7 is used for cards which specify options for compiler operation. (See section 11.)

b. If column 7 contains an asterisk (*), the rest of the card is considered to be a comment and, hence, is not "compiled" to produce object code.

c. If column 7 contains a slash (/), the listing, if any, is advanced to channel 1 before printing, and the card is considered to be a comment card.

d. The letter L followed by a "library-name" entry causes all succeeding source card data to be placed into the COBOL Library File during compilation. Termination of the action takes place when an L card is encountered followed by spaces.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM	COBOL DIVISION	PAGE OF
PROGRAMMER	DATE	IDENT 73 80

PAGE NO. 1 3	LINE NO. 4 6	7	A 8 11	B 12 22 32 42 52 62 72 Z

Figure 3-1. COBOL Coding Form

e. The presence of a hyphen (-) indicates that the last word or literal on the previous card is not complete, but is continued on this card.

Words and numeric literals may be split at any point by placing a hyphen in column 7 of the following card. Any rightmost blank spaces on a card are ignored as are the leftmost blank spaces on the continuation card.

Non-numeric literals are split in a slightly different fashion. On the initial card, starting from the quotation mark, all information through column 72 is taken as part of the literal, and on the next card a quote mark must be used to indicate the start of the second part of the literal.

MARGIN A (COLUMNS 8 THRU 11)

DIVISION, SECTION, and PARAGRAPH headers must begin in margin A. A division header consists of the division name (IDENTIFICATION, ENVIRONMENT, DATA, or PROCEDURE), followed by a space, then the word DIVISION followed by a period.

A section header consists of the section-name, followed by a space and then the word SECTION, followed by an optional priority number, followed by a period.

A paragraph header consists of the paragraph-name followed by a period. The first sentence of the paragraph may appear on the same line as the paragraph header.

Within the IDENTIFICATION and ENVIRONMENT divisions, the section and paragraph headers are fixed and only the headers shown in this manual are permitted. Within the PROCEDURE DIVISION, the section and paragraph headers are defined by the user.

MARGIN B (COLUMNS 12 THRU 72)

All entries which are not DIVISION, SECTION, or PARAGRAPH headers should start in margin B.

RIGHT MARGIN (COLUMN 72)

The text of the program must appear between columns 8 and 72, inclusive. A word or statement may end in column 72.

IDENTIFICATION (COLUMNS 73 THRU 80)

The identification field may contain any information desired by the user. The field is ignored but is reproduced on the output listing by the compiler. This field normally contains the program name.

PUNCTUATION

The following rules of punctuation apply to the writing of COBOL programs for the B 1700.

a. A sentence is terminated by a period followed by a space. A period may not appear within a sentence unless it is within a non-numeric literal or is a decimal point in a numeric literal or PICTURE string.

b. Two or more names in a series may be separated by a space or by a comma. If used, commas can appear only where allowed.

c. Semicolons (;) are used only for readability and are never required.

d. A space must never be embedded in a name; hyphens should be used instead. (A hyphen may not start or terminate a name.) For example:

NET-PAY

SAMPLE CODING

An extract sample from a source program, showing the continuation of both words and non-numeric literals, is illustrated in figure 3-2.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM CONTINUATION | COBOL DIVISION | PAGE 1 OF 1

PROGRAMMER SAM | DATE | IDENT. 73 80

```
PAGE NO. 1 3  LINE NO. 4 6  7  A 8 11  B 12 22 32 42 52 62 72 Z
      FILE-CONTROL. SELECT PRINTINGFILE ASSIGN TO TAPE RESERVE
     -    2 ALTERNATE AREAS.
          SELECT MASTERINPUT ASSIGN TO DISK RES
     -  ERVE 1 ALTERNATE AREA.

      WORKING-STORAGE SECTION.

      01  A---FEW--LITERALS.  05  NUM-
     -                    -LIT-123  VALUE 123
     -                456.78  PIC 99
     -                        999
     -                      9V99.
                      05 NON-NUM--LIT PIC X(30)  VALUE  "ALL C
     -  "HARACTERS VALID.$9;t-".

      PROCEDURE DIVISION.

      FIRST--PARAGRAPH.  MOVE 3456
     -                12.00 TO NUM--LIT-123.
                      IF THEE = THOU GO TO FIRST--PA
     -  GRAPH.
```

Figure 3-2. Example of Continuation of Words and Literals

SECTION 4

IDENTIFICATION DIVISION

GENERAL

The first part or division of the source program is the IDENTIFICATION DIVISION. Its function is to identify the source program and the resultant output of its compilation. In addition, the date the program was written, the date the compilation was accomplished, plus other pertinent information may be included in the IDENTIFICATION DIVISION.

IDENTIFICATION DIVISION STRUCTURE

The structure of this division is as follows:

```
[MONITOR...]
 IDENTIFICATION DIVISION.
[PROGRAM-ID.  Any COBOL word.]
[AUTHOR.  Any entry.]
[INSTALLATION.  Any entry.]
[DATE-WRITTEN.  Any entry.]
[DATE-COMPILED.  Any entry - appended with
                 current date and time as main-
                 tained by the MCP.]
[SECURITY.  Any entry.]
[REMARKS.  Any entry.  Continuation lines must
           be coded in Area B of the coding form.]
```

The following rules must be observed in the formation of the IDENTIFICATION DIVISION:

a. The IDENTIFICATION DIVISION must begin with the reserved words IDENTIFICATION DIVISION followed by a period.

b. All paragraph-names within this division must begin in Area A of the coding form.

c. An entry following a paragraph-name cannot contain periods, with the exception that a period must be present to denote the end of that entry.

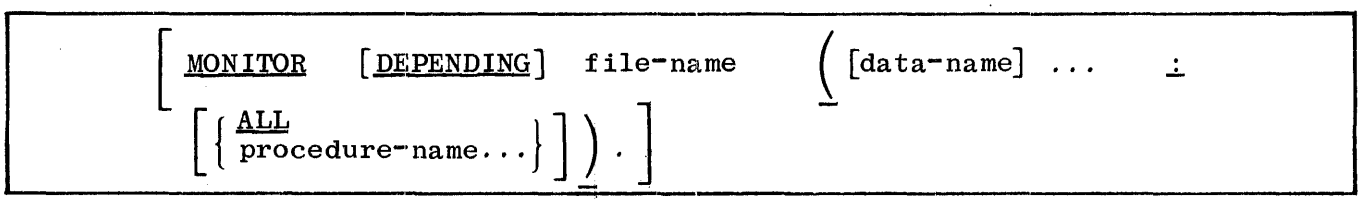
When DATE-COMPILED is included, the compiler automatically inserts the time of compilation in the form of HH:MM and the date of compilation in the form of MM/DD/YY.

With the exception of the DATE-COMPILED paragraph, the entire division is copied from the input source program by the compiler and listed on the output listing for documentation purposes only.

MONITOR

This statement provides a debugging trace of specified data-names and/or procedure-names.

The format of this statement is:



This statement must begin under Area A of the coding form. The parentheses and colon are required as part of the source program statement.

Only one MONITOR statement per program is allowed and must precede the IDENTIFICATION DIVISION header card in the source program.

The file-name must be ASSIGNed to a line printer and is recognized by the compiler as being the output media for the MONITORed data-names. When the ALL option is used, the file-name must be opened in the first paragraph in the program; otherwise, a run-time error will occur.

The data-name(s) may be any name(s) appearing in the DATA DIVISION except for those which require subscripting or indexing.

Whenever a MONITORed elementary data-name is encountered as the receiving field in a MOVE or arithmetic statement, the data-name and its current value are listed.

If a group item appears in the data-name-list, it will be MONITORed only when explicitly used as a receiving field.

If the DEPENDING option is present, SW6 will be tested for an ON-OFF condition. Print of MONITORed items will depend upon the setting as being "ON".

All paragraph-names listed will be printed each time they are encountered, along with a total indicating the number of times that a paragraph-name has been passed.

The use of the ALL option, instead of the procedure-name list, will cause all section-names and paragraph-names to be MONITORed, thus providing a trace of the entire program's control path during operation.

CODING THE IDENTIFICATION DIVISION

Figure 4-1 provides an example of how the IDENTIFICATION DIVISION may be coded in the source program. Note that continued lines must be indented to the B position of the form, or beyond.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM: IDENTIFICATION DIVISION CODING | COBOL DIVISION | PAGE ___ OF ___

PROGRAMMER: DEE | DATE | IDENT 73 80

PAGE NO. 1 3	LINE NO. 4 6	7	A 8 11	B 12 22 32 42 52 62 72
				IDENTIFICATION DIVISION.
				PROGRAM-ID. SALES-PERFORMANCE.
				AUTHOR. LULU SMITH.
				INSTALLATION. BURROUGHS CORPORATION
				GOLETA, CA.
				DATE-WRITTEN. 6 JUNE 1974.
				DATE-COMPILED.
				SECURITY. LEVEL 1.
		*		SEE DOCUMENTATION BOOK
				REMARKS. THE FIRST PART OF THE PROGRAM PRINTS ACTUAL SALES AND
				SALES QUOTA FIGURES IN STATEMENT FORM
		*		THE SECOND PART EXPRESSES THESE IN BAR
				GRAPH FORMAT.

Figure 4-1. IDENTIFICATION DIVISION Coding

SECTION 5

ENVIRONMENT DIVISION

GENERAL

The ENVIRONMENT DIVISION is the second division of a COBOL source program. Its function is to specify the computer being used for the program compilation, to specify the computer to be used for object program execution, to associate files with the computer hardware devices, and to provide the compiler with pertinent information about disk storage files defined within the program. Furthermore, this division is also used to specify input-output areas to be utilized for each file declared in a program.

ENVIRONMENT DIVISION ORGANIZATION

The ENVIRONMENT DIVISION consists of two sections. The CONFIGURATION SECTION contains the overall specifications of the computer. The INPUT-OUTPUT SECTION deals with files to be used in the object program.

ENVIRONMENT DIVISION STRUCTURE

The structure of this division is as follows:

```
ENVIRONMENT DIVISION.
[[CONFIGURATION SECTION.]
[SOURCE-COMPUTER . . .]
[OBJECT-COMPUTER . . .]
[SPECIAL-NAMES . . .]
[INPUT-OUTPUT SECTION.]
[FILE-CONTROL . . .]
[I-O-CONTROL . . .]]
```

The following rules must be observed in the formulation of the ENVIRONMENT DIVISION:

a. The ENVIRONMENT DIVISION must begin with the reserved words ENVIRONMENT DIVISION followed by a period.

b. All entries other than the ENVIRONMENT DIVISION source line are optional but, when used, they must begin in Area A of the coding form.

CONFIGURATION SECTION

The CONFIGURATION SECTION contains information concerning the system to be used for program compilation (SOURCE-COMPUTER), the system to be used for program execution (OBJECT-COMPUTER), and the special-names paragraph, which relates hardware names used by the B 1700 COBOL compiler to the mnemonic-names in the source program.

Source-Computer

The function of this paragraph is to allow documentation of the configuration used to perform the COBOL compilation.

The format of this paragraph has the following two options:

Option 1:

```
SOURCE-COMPUTER. COPY library-name
[, REPLACING word-1 BY word-2
    [, word-3 BY word-4] ... ].
```

Option 2:

```
SOURCE-COMPUTER. { B-1700    }
                 { any entry }
```

This paragraph is for documentation only.

OBJECT-COMPUTER

Object-Computer

The function of this paragraph is to allow a description of the configuration used for the object program.

The format of this paragraph has the following two options:

Option 1:

```
OBJECT-COMPUTER. COPY library-name
[, REPLACING word-1 BY word-2
   [, word-3 BY word-4] ...] .
```

Option 2:

```
OBJECT-COMPUTER. [ {B-1700   } ]
                 {any entry}
[, [SORT] MEMORY SIZE    integer-1 [CHARACTERS]]
 [ , DATA SEGMENT-LIMIT IS integer-2 CHARACTERS]
 [ , SEGMENT-LIMIT IS priority number]  .
```

If section priority numbers are used in the PROCEDURE DIVISION, they must be positive integers with a value from 0 through 99. The SEGMENT-LIMIT clause signifies the limit for non-overlayable program segmentation of sections numbered from 0 through 49. See SEGMENT CLASSIFICATION, PROGRAM SEGMENTS, and PRIORITY NUMBERS.

The MEMORY SIZE clause is used to increase the amount of memory for overlayable data or change the size of memory for the sort to use during a sort operation.

When the SORT MEMORY SIZE clause is used, the integer-1 will reflect the amount of memory the sort will use when the program is executed. If integer-1 is less than 8K bytes, the sort will use 8K bytes by default.

When the MEMORY SIZE clause is used without the SORT option, the compiler will assign the amount of memory between the base and limit register to reflect the size of integer-1, or the memory required for all of the overlayable data segments when more than one segment is referenced in the same operation.

Both SORT MEMORY SIZE and MEMORY SIZE clauses may be used in the same OBJECT COMPUTER paragraph.

The use of the word CHARACTERS after integer-1 specifies the number of bytes to be used; otherwise, the specification is the number of digits to be used.

The DATA SEGMENT-LIMIT clause may be used to specify the size of the data segments in the WORKING-STORAGE section. Integer-2 will reflect the number of characters desired in each data segment. When the value of integer-2 is zero, the WORKING-STORAGE section will not be segmented, and will reside in memory as a contiguous block.

If the DATA SEGMENT-LIMIT clause is omitted, no data segmentation will take place.

When data segmentation is specified each file record is placed in a separate segment.

All 77 level entries are placed in data segment 0 (zero).

A record (01 level) that is greater in length than the DATA SEGMENT-LIMIT will be placed in a segment by itself, and will not be split between segments. If DATA SEGMENT-LIMIT has been declared larger than the defined record size, the record will reside in the declared amount of memory, as well as succeeding records to the limit of the defined segment.

Special-Names

The function of this paragraph is to allow the programmer to assign a significant character for all currency signs, to declare decimal points as being commas and to provide a means of relating implementor hardware-names to user specified mnemonic-names.

The format of this paragraph has the following two options:

Option 1:

```
SPECIAL-NAMES.  COPY  library-name

 [REPLACING  word-1  BY  word-2

 [, word-3  BY  word-4] ... ].
```

Option 2:

```
SPECIAL-NAMES.      [CURRENCY SIGN IS literal]

    [, implementor-names IS mnemonic-name] . . .

    [, DECIMAL-POINT IS COMMA]
```

This paragraph is required if all decimal points are to be interchanged with commas and/or if all currency signs are to be represented by a character other than a dollar sign ($).

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

a. Numeric digits 0 through 9.
b. Alphabetic characters A, B, C, D, J, K, P, R, S, V, X, Z, or blank.
c. Special characters * + - , . ; ( ) ".

The clause DECIMAL-POINT IS COMMA signifies that the functions of comma and period are to be exchanged in the PICTURE character-string and in numeric literals.

The implementor-name clause must be one of the allowable B 1700 COBOL hardware-names which may be specified in FILE-CONTROL paragraph. For example:

PUNCH IS CARD-PUNCH-EBCDIC

The mnemonic named device can be directly referred to in the ASSIGN clause.

The SPECIAL-NAMES paragraph statement ends with a period as a delimiter. Periods between clauses are not allowed.

INPUT-OUTPUT SECTION

The INPUT-OUTPUT section contains information concerning files to be used by the object program, the manner of recording used or to be used, and the presence of any multiple-file tape or disk.

FILE-CONTROL

The function of this paragraph is to name each file, to identify the file medium, and to specify a particular hardware assignment. The paragraph also specifies alternative input-output areas.

The format of this paragraph has the following three options:

Option 1:

```
FILE-CONTROL.    COPY    library-name
[ REPLACING  { word-1 | data-name-1 }  BY  { word-2 | data-name-2 | literal-1 }
  [ , { word-3 | data-name-3 }  BY  { word-4 | data-name-4 | literal-2 } ] ... ]
```

Option 2:

```
FILE-CONTROL.

    SELECT [OPTIONAL]  file-name-1 ASSIGN TO hardware-name-1

[ [OR] BACKUP [ { TAPE | DISK } ] ] [FORM] [FOR MULTIPLE REEL] [SINGLE]
                                   [ALL-AT-OPEN] [WORK]

[ ,RESERVE  { NO | integer-1 }  [ALTERNATE [ { AREA | AREAS } ]] ]

[ ,{ FILE-LIMIT IS | FILE-LIMITS ARE }  { literal-1 | data-name-1 }  { THRU | THROUGH }  { END | literal-2 | data-name-2 }

  [ , { literal-m | data-name-m }  { THRU | THROUGH }  { literal-n | data-name-n } ] : . . ]

[ ,ACCESS MODE IS  { RANDOM | SEQUENTIAL } ]

[ ,ACTUAL KEY IS data-name-3]

[ ,PROCESSING MODE IS SEQUENTIAL ]  . [SELECT] . . .
```

Option 3:

```
FILE-CONTROL.
SELECT sort-file-name ASSIGN TO SORT DISK.
```

Option 1 may be used when the system's library contains the LIBRARY name entry. See COPY verb, section 7.

The files used in a program must be the subject of only one SELECT statement. If it is to be OPENed INPUT-OUTPUT or I-O, it must be present in the MCP Disk Directory.

The OPTIONAL clause is applicable to input files only. Its specification is required for input files that are not necessarily present each time the object program is executed.

The ASSIGN clause must be used in order for the MCP to associate the file with a hardware peripheral component. The allowable hardware-name entries are:

CARD96	QUEUE
DISK (or DISC)	READER
DISK-DFC1	READER-SORTER
DISK-DFC2	REMOTE
DISK-DPC1	SPO
DISK-DPC2	TAPE (7 or 9 channel MCP to assign)
DISK-HPT	TAPE-MTC1
DISKPACK	TAPE-MTC2
MFCU	TAPE-MTC3
PRINTER	TAPE-MTC4
PT-PUNCH	TAPE-MTC5
PT-READER	TAPE-7 (7 channel only)
PUNCH	TAPE-9 (9 channel only)

The BACKUP option will cause printer output files to be placed on a printer backup tape or disk file for subsequent printing. The BACKUP option will cause punch output files to be placed on punch backup disk files for subsequent punching.

When hardware-name-1 is selected, without the backup option, the output file may be manually assigned to printer backup by the operator with an "OU" message.

Use of the FORM option with printer or punch files will cause the program to halt and an MCP message to be printed declaring the need for special forms to be loaded in the Line Printer or Card Punch, as applicable.

It is recommended that a STOP literal be executed just prior to a STOP RUN if the FORM option is used. This will allow the operator sufficient time to remove the special forms before the printer is released back to the MCP. Without a temporary halt, there is a possibility that another job in the mix may start printing on that same printer.

With the exception of the ASSIGN clause which must follow the SELECT clause, the rest of the clauses in this paragraph may appear in any order.

The MULTIPLE REEL clause is for documentation only. This function is performed by the MCP.

When the SINGLE option is used, a file assigned to DISKPACK will not be assigned to a multi-file disk cartridge.

The ALL-AT-OPEN option will cause the MCP to allocate all of the areas requested by this file at the time the file is opened.

When the WORK option is used, the MCP will insert a six digit job number (assigned to the program) into the file name starting in the second position from the left. This will allow a program with temporary work files to be multi-programmed.

The RESERVE clause allows a variation of the number of input or output physical record buffers to be supplied by the MCP at the time the file is opened. Each alternate area reserved requires additional memory to be utilized, and will be the size of a physical record as defined in the FD statement of the DATA DIVISION for that specific file. Up to 63 alternate areas may be specified.

No alternate areas are reserved when the NO option is specified or if the entire option is omitted.

The MCP will keep track of record data being passed to or from the buffer and the record work area.

The programmer can use the READ or WRITE statements without regard to the buffering action taking place.

The FILE-LIMIT clause is invalid if specified for a sort file description (SD) entry. The FILE-LIMIT clause for input and output files associated with the SORT verb will not be effective during execution of the SORT unless an input/output procedure is declared.

The FILE-LIMIT clause specifies the following:

a. For SEQUENTIAL access, logical records are obtained from, or placed sequentially in, the disk storage file by the implicit progression from segment to segment. The AT END imperative statement of a READ statement is executed when the logical end of the last segment of the file is reached and an attempt is made to READ another record. The INVALID KEY clause of a WRITE statement is executed when the end of the last segment is reached and an attempt is made to WRITE another record. The END option specifies that the compiler is to determine the upper limit of an existing file. No ACTUAL KEY entry is necessary for the SEQUENTIAL mode.

b. For RANDOM access, logical records are obtained from, or placed randomly in, the disk storage file within the specified FILE-LIMIT. The contents of ACTUAL KEY not within the specified limit will cause the execution of the INVALID KEY branch in the READ and the WRITE statements. The ACTUAL KEY entry must be specified.

In the FILE-LIMIT clause, each pair of operands associated with the key word THRU represents a logical segment of a file. The logical beginning of a disk storage file is considered to be that address represented by the first operand of the FILE-LIMIT clause; the logical end is considered to be that address as specified by the last operand of the FILE-LIMIT clause.

In a FILE-LIMIT series, SEQUENTIAL records are accessed in the order in which they are specified. For example:

```
FILE-LIMITS 1 THRU 5, 10 THRU 12, 3 THRU 7
```

This example will result in the sequential access of records 1, 2, 3, 4, 5, 10, 11, 12, 3, 4, 5, 6 and 7 in that order.

The data-names used with the FILE-LIMIT clause must be defined with a PICTURE of 9(8) COMPUTATIONAL.

For the ACCESS MODE SEQUENTIAL clause, the disk storage records are obtained or placed sequentially. That is, the next logical record is made available from the file on a READ statement execution, or a specific logical record is placed into the file on a WRITE statement execution. The ACCESS MODE SEQUENTIAL clause is assumed if ACCESS MODE RANDOM is not specified.

Values of the ACTUAL KEY data-name-3 are controlled by the programmer, including any execution of the USE FOR KEY CONVERSION statement. The value may range from 1 to n, where n equals the number of records in the file or as

reflected by the FILE-LIMITS clause. The ACTUAL KEY signifies the relative position of a record within the file and is equated to a data-name at any level which is defined with a PICTURE of 9(8) COMPUTATIONAL. ACTUAL KEY is not used for ACCESS MODE SEQUENTIAL files.

The ACTUAL KEY specified for a queue file signifies the relative sub-queue position within the file and is equated to a data-name at any level which is defined with a PICTURE of 9(8) COMPUTATIONAL. If no KEY is specified, the relative queue number will be set to 1.

The ACTUAL KEY specified for a remote file is defined as follows:

```
01 REMOTE-KEY
   03 STATION-RSN      PC 9(3)   (As defined in NDL network controller)
   03 TEXT-LENGTH      PC 9(4)   (Actual length of current message)
   03 MSG-TYPE         PC X(3)   (0 = write
                                  1 = read)
```

STATION-RSN refers to the relative station number within the file. TEXT-LENGTH defines the length of the message in characters, and should never be larger than the largest 01 record declared for the file. Otherwise, data would be truncated from the low-order position.

MSG-TYPE defines a user-defined value to be treated appropriately by the user program.

The ACTUAL KEY for remote file does not have to be defined at the 01 level; however, the group length must be 10 bytes. If ACTUAL KEY is omitted, message length will be taken from the message length being written.

The PROCESSING MODE IS SEQUENTIAL clause is for documentation only.

All integers must be of positive values.

File-name-1 must be unique in the first ten characters if the use of an MCP Label Equation Card is anticipated.

The sort-file-name in Option 3 is the SD level file-name to be used by the SORT verb.

I-O-Control

The function of this paragraph is to specify memory area, to be shared by different files during object program execution and the point in time that a rerun procedure is to be established.

The construct of this paragraph is:

Option 1:

```
I-O-CONTROL.  COPY  library-name
  [REPLACING  word-1  BY  word-2
  [, word-3  BY  word-4] ...].
```

Option 2:

```
I-O-CONTROL.

[; SAME  [RECORD] AREA FOR file-name-2 [file-name-3] ...]
[; MULTIPLE FILE  {DISKPACK  dispack-id }
                  {TAPE      multi-file-id}
      CONTAINS file-name-5  [POSITION integer-2]
     [, file-name-6 [POSITION integer-3] ] ...  ]
```

The I-O-CONTROL paragraph name may be omitted from the program if the paragraph does not contain any of the clause entries.

The SAME AREA clause in this COBOL compiler is used to assign the same address to the record work areas of all files named in the clause. This area will be in the overlayable data section of the program when data segmentation is used. Due to the Virtual Memory concept employed in the design of the system, a given file's file information block (FIB), buffer, and ALTERNATE AREAS will not exist

in memory until an OPEN statement in the PROCEDURE DIVISION has been executed. At this time, to contain these areas the MCP allocates sufficient memory outside of the limits of the Base and Limit registers. The Record Work area of the file is called into the overlayable data section of the program whenever it is referenced by the program. When the file is programmatically CLOSEd, the memory being used to contain the file's FIB, buffer and ALTERNATE AREAS will be returned to the MCP.

COBOL restricts the OPENing of files defined as residing in the SAME AREA of memory to one file at a time. This system ignores that logic and the result saves memory over the conventional intent by not using memory to contain FIB record area, buffers, or ALTERNATE AREAS until a file is actually OPENed by the program.

When the RECORD option of the SAME AREA clause is used, only the record area is shared and the associated alternate areas for each file remain independent. In this case, any number of the files sharing the same record area may be OPEN at one time, but only one of the records can be processed at a time.

The use of the RECORD option may decrease the physical size of a program as well as increase the speed of the object program. To illustrate this point, consider file maintenance. If the SAME RECORD AREA is assigned to both the old and new files, a MOVE will be eliminated which transfers each record from the input area to the output area. The records do not have to be defined in detail for both files. Definition of a record within one file and the simple inclusion of an 01 level entry for the other file will suffice.

Because these are record areas, in fact, in the same memory location, one set of data-names is sufficient for all processing requirements, without requiring qualification.

The MULTIPLE FILE clause specifies that disk files reside on a removable disk cartridge or disk pack, or two or more tape files are resident on one magnetic tape. All files resident on a multi-file (that are required in a program) must be represented in the source program by a SELECT statement and a FD entry for each file.

For tape, the file-name entries do not have to be defined in the program sequence in which the files appear on the multi-file tape. However, the MCP will read the label of the next file on tape, check the label against the file request, and, if the next file is not the one requested, the MCP will rewind the multi-file tape and will start searching for it from the beginning of tape.

When the MULTIPLE FILE clause is used to identify a file on a removable disk cartridge or disk pack, the MCP will use the specified diskpack-ID to locate that file. File-name list is a series of FD file-names in the program indicated as residing on the specified disk cartridge or disk pack.

The "multi-file-id" is the file-name contained in the physical tape label of a magnetic tape containing multi-files, when file-name-list is a series of FD file-names in the program indicated as residing on the multi-file-tape.

All files named in the MULTIPLE FILE TAPE clause have an implied SAME AREA clause.

Multi-files, or any file contained within the file may be OPTIONAL.

The POSITION clause is for documentation only.

CODING THE ENVIRONMENT DIVISION

An example of ENVIRONMENT coding is provided in figure 5-1.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM	ENVIRONMENT DIVISION CODING	COBOL DIVISION	PAGE 1 OF 1
PROGRAMMER	KEAVIN	DATE	IDENT 73 80

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ENVIRONMENT-DIVISION-CODING.
AUTHOR. KEAVIN.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. B-1700.
OBJECT-COMPUTER. B-1700 SEGMENT-LIMIT IS 119.
SPECIAL-NAMES. DECIMAL-POINT IS COMMA.

INPUT-OUTPUT SECTION.
FILE-CONTROL. SELECT OPTIONAL DAILY-TAPE ASSIGN TO TAPE.
    SELECT MASTER-FILE ASSIGN TO DISK
        FILE-LIMIT IS 1 THRU 1000
        ACCESS MODE RANDOM ACTUAL KEY DISK-CONTROL.
    SELECT MASTER-TAPE ASSIGN TO TAPE.
    SELECT ERROR-TAPE ASSIGN TO TAPE RESERVE NO ALTERNATE AREAS.
    SELECT DETAIL-CHANGES-FILE ASSIGN TO DISK.
    SELECT SUMMARY-FILE ASSIGN TO DISK.

I-O-CONTROL.
    SAME RECORD AREA FOR DAILY-TAPE, ERROR-TAPE
    MULTIPLE FILE DISKPACK "MULTIPAK" CONTAINS
        MASTER-FILE DETAIL-CHANGES-FILE SUMMARY-FILE.
```

Figure 5-1. ENVIRONMENT DIVISION Coding

SECTION 6

DATA DIVISION

GENERAL

The third part of a COBOL source program is the DATA DIVISION which describes all data that the object program is to accept as input, and to manipulate, create, or produce as output. The data to be processed falls into three categories:

a. Data which is contained in files and which enters or leaves the internal memory of the computer from a specified area or areas.

b. Data which is developed internally and placed into intermediate storage, or placed into a specific format for output reporting purposes.

c. Constants which are defined by the programmer.

DATA DIVISION ORGANIZATION

The DATA DIVISION is subdivided into two sections:

a. The FILE SECTION defines the contents of data files which are to be created or used by an external medium. Each file is defined by a file description, followed by a record description or a series of file-related record descriptions.

b. The WORKING-STORAGE SECTION describes records, constants, and noncontiguous data items which are not part of an external data field, but which are developed and processed internally.

DATA DIVISION STRUCTURE

The general structure of the DATA DIVISION is as follows:

```
DATA DIVISION.
[FILE SECTION.
     [{file-description-entry}     [record-description-entry] ...] ...]
      {sort-description-entry}

  [WORKING-STORAGE SECTION.
       [77-level-description-entry ] ...]
       [record-description-entry   ]
```

Each section of the DATA DIVISION is optional and may be omitted from the source program if not needed. However, if a section is included, it must be incorporated in order of appearance shown above. These sections are described on the following pages.

The file description defines information pertaining to the physical aspects of a file. Such items as number of records in a block, identification of records in the file, the presence or absence of labels, etc., are included to describe the entire file.

The record description presents logical characteristics of each record. This includes the layout of items within each record type, size of various items in the record, indication of the range of values for each item, picture of the contents of each item, whether the item is signed or not, and the usage of an item within the program. All of these parameters may be utilized to define logical characteristics of each record.

The WORKING-STORAGE SECTION is comprised of internal record descriptions and individual unrelated items, which are described as record entries, or parts of record entries.

In summary, the DATA DIVISION contains information pertaining to the data to be used by the program: the files used, the records contained in each file, and items comprising each record; in addition, working storage and constants may be specified.

FILE AND RECORD CONCEPTS

The approach taken in defining file information is to distinguish between the physical aspects of the file and the conceptual characteristics of the data contained within the file.

Physical Aspects of a File

The physical aspects of a file describe the data as it appears on the input or output media and include such features as the following:

a. The mode in which the data file is recorded on the external medium.

b. The grouping of logical records within the physical limitations of the file medium.

c. The means by which the file can be identified.

Conceptual Characteristics of a File

The conceptual characteristics of a file explicitly define each logical entity within the file itself. In a COBOL program, the input or output statements refer to one logical record.

It is important to distinguish between a physical record and a logical record. For COBOL a logical record is a group of related information, uniquely identifiable, that is treated as a unit.

A physical record is a physical unit of information whose size and recording mode are convenient to a particular computer for the storage of data on an input or output device. The size of a physical record is hardware-dependent and bears no direct relationship to the size of the file of information contained on a device.

A logical record may be contained within a single physical unit; or several logical records may be contained within a single physical unit; or a logical record may require more than one physical unit to contain it. There are several source-language methods available for describing the relationship of logical records and physical units. Once the relationship has been established, the control of the accessibility of logical records as related to the physical unit is the responsibility of the operating system. In this manual, reference to records means to logical records, unless the term "physical record" is specifically used.

The concept of a logical record is not restricted to files but may be applied to all sections of the DATA DIVISION.

Record Concepts

The record description consists of a set of DATA DESCRIPTION entries which describe the characteristics of a particular record. Each DATA DESCRIPTION entry consists of a level-number followed by a data-name, followed by a series of independent clauses, as required.

Example:

```
01 ITEM-ONE PICTURE IS X(6).
```

The maximum size of a record description (i.e., the sum of the maximum sizes of all the items subordinate to an 01 level item) is restricted to 65,535 bits.

LEVEL NUMBERS CONCEPT

The concept of hierarchy is inherent in the structure of a logical record. This concept arises from the need to specify subdivisions of a record for the purpose of data reference. Once a subdivision has been specified, it may be further subdivided to permit more detailed data referral. In other words, level numbers define the interrelationship of the items comprising the record and allow the programmer to access individual items or groups of items.

The most basic (least generic) subdivisions of a record, that is, those not further subdivided, are called elementary items; consequently, a record is said to consist of a sequence of elementary items, or the record itself may be an elementary item.

In order to refer to a set of elementary items, the elementary items may be combined into groups. Each group consists of a named sequence of one or more elementary items. Groups, in turn, may be combined into groups of two or more groups, etc. Thus, an elementary item may belong to more than one group.

In COBOL, the item relationship is specified by the use of a series of level numbers. These numbers may range from 1 thru 49. (Special level numbers of 66, 77, and 88 are discussed later.)

Each record of a file begins with the level number 1 (which may also be written as 01). This number is reserved for the record name only, as the most generic grouping. Less inclusive groupings are given higher numbers (not necessarily successive) up to a limit of 49. Figure 6-1 illustrates a form of level construction.

The smallest elements of the description are called elementary items. In figure 6-1, EMP-NO, EMP-COST-CENTER, EMP-LAST-NAME, EMP-FIRST-INITIAL, and EMP-M-INITIAL are all elementary items, as well as EMP-H-MONTH, EMP-H-DAY, EMP-H-YEAR, EMP-GROSS, EMP-HOSPITAL, EMP-LIFE, EMP-FICAT, EMP-STATE-TAX, EMP-WITHHOLDING, EMP-LMONTH and EMP-LDAY. None of these items are further subdivided; therefore, they are called elementary items.

Each elementary item belongs to one or more groups. In the example, EMP-HOSPITAL is a part of the EMP-INSURANCE group. EMP-INSURANCE, in turn, is part of the EMP-DEDUCTIONS group, which is part of the EMP-PAY-DATA group. Therefore, a group is defined as being composed of all group and elementary items described under it, until a level number equal to or less than the

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM: LEVEL NUMBER CONSTRUCTION | COBOL DIVISION: | PAGE: 1 OF 1

PROGRAMMER: BARBRA | DATE: | IDENT 73 80

PAGE NO. 1 3	LINE NO. 4 6	7	A 8 11	B 12 22 32 42 52 62 72
			01	EMPLOYEE-INFO.
				03 EMP-NO PIC 9(5).
				03 EMP-COST-CENTER PIC 99.
				03 EMP-NAME.
				05 EMP-LAST-NAME PIC X(13).
				05 EMP-FIRST-INITIAL PIC X.
				05 EMP-M-INITIAL PIC X.
				03 EMP-ANNUAL-SALARY PIC 9(6)V99.
				03 EMP-DT-HIRED.
				05 EMP-H-MONTH PIC 99.
				05 EMP-H-DAY PIC 99.
				05 EMP-H-YEAR PIC 99.
				03 EMP-PAY-DATA.
				05 EMP-GROSS PIC 9(6)V99.
				05 EMP-DEDUCTIONS.
				09 EMP-INSURANCE.
				11 EMP-HOSPITAL PIC 9(4)V99.
				11 EMP-LIFE PIC 9(4)V99.
				09 EMP-TAXES.
				11 EMP-FICA PIC 9(4)V99.
				11 EMP-STATE-TAX PIC 9(4)V99.
				11 EMP-WITHHOLDING PIC 9999V9(2).
				03 EMP-LAST-REVIEW.
				04 EMP-L-MONTH PIC 99.
				04 EMP-L-DAY PIC 99.

Figure 6-1. Level Number Construction

group level number is encountered. In the example, EMP-PAY-DATA group includes all items to, but not including, EMP-LAST-REVIEW (which has an equal level number). Likewise, EMP-DEDUCTIONS group includes all subsequent items up to, but not including, EMP-LAST-REVIEW (which has a level number less than EMP-DEDUCTIONS).

Level numbers used in defining successively smaller groupings, working toward an elementary item, are given in larger values. Although it is not necessary that they be consistent or consecutive, a level number must not exceed 49. A level number immediately following the last elementary item of a group must have a value of less than or equal to the level number for that group and equal to the level number of some previous group. An exception is that level number 1 (or 01) is reserved exclusively for identifying the beginning of a record description.

In the above example, the rule prohibits EMP-ANNUAL-SALARY from having a level number of 2 (or 02). Likewise, the entry name EMP-LAST-REVIEW could not have had a level number of 10 or 06 because, in the example, no previous group appears with either of these levels. As a completely separate group, it could only have a level number the same as that of the major groups previously shown. Figure 6-2 illustrates another way to visualize the concept of level numbers by using the same example.

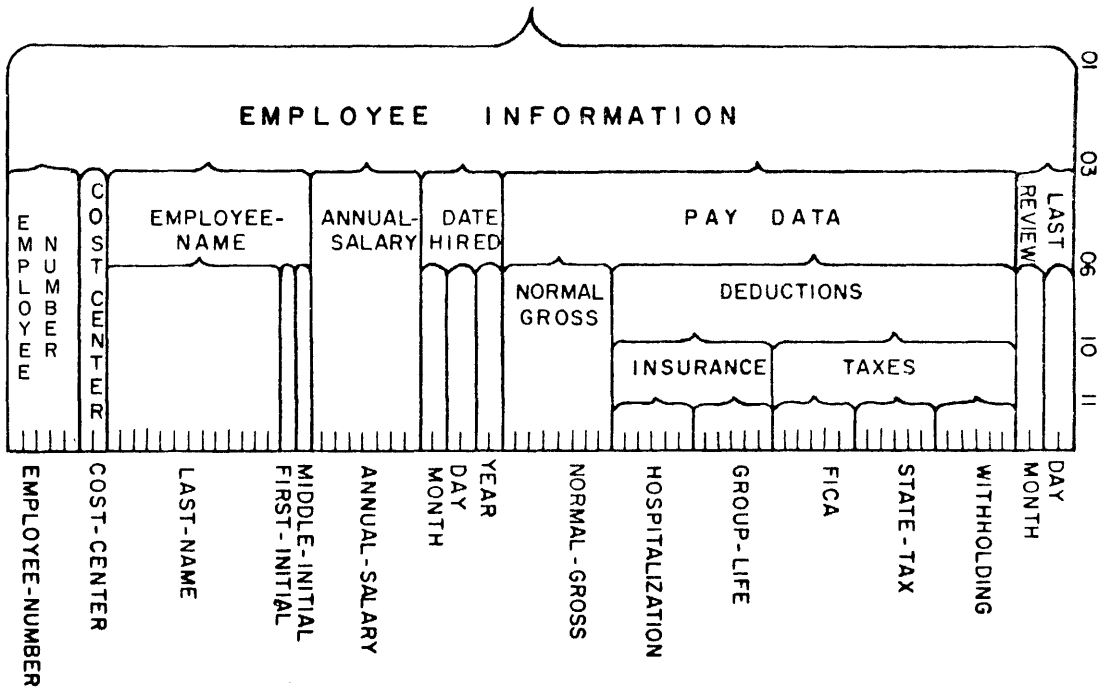


Figure 6-2. Concept of Level Numbers

QUALIFICATION

Every user-defined name explicitly referenced in a COBOL source program must be uniquely referenced either because no other name has the identical spelling and hyphenation or because it is unique within the context of a REDEFINES clause, or because the name exists within a hierarchy of names such that reference to the name can be made unique by mentioning one or more of the higher-level names in the hierarchy. These higher-level names are called qualifiers and this process that specifies uniqueness is called qualification. Identical user-defined names may appear in a source program; however, uniqueness must then be established through qualification for each user-defined name explicitly referenced, except in the case of redefinition. All available qualifiers need not be specified so long as uniqueness is established.

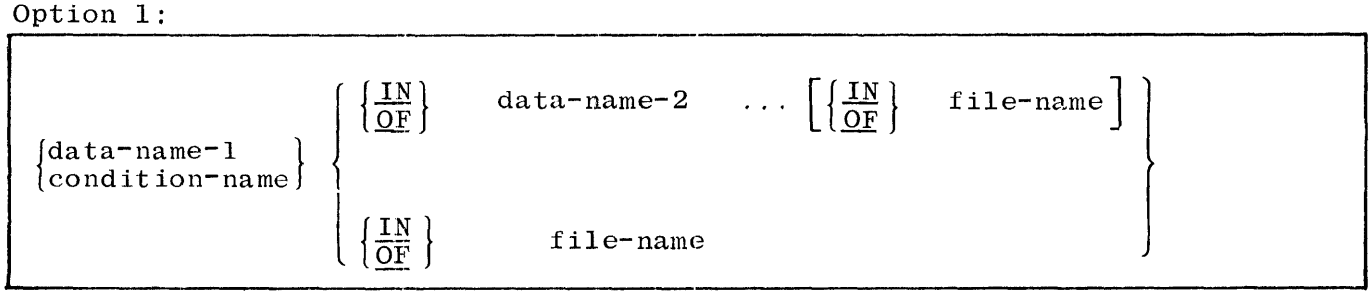
The hierarchy of qualification is as follows: names associated with a level indicator are the most significant; then names associated with level-number 01, then those names associated with level-number 02, ... , 49. A section-name is the highest (and the only) qualifier available for a paragraph-name. Thus, the most significant name in the hierarchy must be unique and cannot be qualified. Subscripted or indexed data-names and conditional variables, as well as paragraph-names and data-names, may be made unique by qualification. The name of a conditional variable can be used as a qualifier for any of its condition-names.

Regardless of the available qualification, no name can be both a data-name and a procedure-name.

Qualification is performed by following a data-name or a paragraph-name by one or more phrases composed of a qualifier preceded by IN or OF. IN and OF are logically equivalent.

The format for qualification consists of two options which are shown below:

Option 1:



Option 2:

paragraph-name	{OF / IN}	section-name

The rules for qualification are as follows:

a. Each qualifier must be of a successively higher level and within the same hierarchy as the name it qualifies.

b. The same name must not appear at two levels in a hierarchy so that the name would appear to qualify itself.

c. If a data-name or a condition-name is assigned to more than one data item in a source program, the data-name or condition-name must be qualified each time it is referred to in the PROCEDURE DIVISION, ENVIRONMENT DIVISION, and DATA DIVISION (except REDEFINES where, by definition, qualification is unnecessary).

d. A paragraph-name must not be duplicated within a section. When a paragraph-name is qualified by a section-name, the word SECTION must not appear. A paragraph-name need not be qualified when referenced within its own section.

e. A data-name cannot be subscripted or indexed when it is being used as a qualifier.

f. A name can be qualified, even though it does not need qualification; if there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used.

In the example below, all item descriptions (except the data-name PREFIX) are unique. In order to refer to either PREFIX item, qualification must be used. Otherwise, if reference is made to PREFIX only, the compiler would not know which of the two is desired. Therefore, in order to move the contents of one PREFIX into the other PREFIX, the PROCEDURE DIVISION must be coded with one of the following sentences:

a. MOVE PREFIX IN ITEM-NO TO PREFIX OF CODE-NO.
b. MOVE PREFIX OF ITEM-NO TO PREFIX IN MASTER-FILE.
c. MOVE PREFIX OF TRANSACTION-TAPE TO PREFIX IN CODE-NO.
d. MOVE PREFIX IN TRANSACTION-TAPE TO PREFIX IN MASTER-FILE.

QUALIFICATION

Example:

```
01  TRANSACTION-TAPE . . .          01 MASTER-FILE . . .
    03 ITEM-NO . . .                    03 CODE-NO . . .
       05 PREFIX . . .                     05 PREFIX . . .
       05 CODE . . .                       05 SUFFIX . . .
    03 QUANTITY . . .                   03 DESCRIPTION . . .
```

TABLES

Frequently, the need arises to describe data that appears in a table (i.e., array, list, etc.). For example, a master record might contain 16 total fields, and these might be described as TOTAL-ONE, TOTAL-TWO, etc. However, this requires 16 data-names, and each total must be individually referenced in the PROCEDURE DIVISION. A more powerful way to describe the field is:

```
TOTAL . . . OCCURS 16 TIMES.
```

Elements of a table are referenced thru the use of subscripting or indexing. An element of a table is represented by an occurrence number.

The elements of a table may contain subordinate fields. For example:

```
02 TOTAL . . .          OCCURS 16 TIMES.
    03 TOTAL-A . . . PICTURE 9(6).
    03 TOTAL-B . . . PICTURE 9(6) OCCURS 3 TIMES.
```

Also, as shown above, OCCURS may be nested to describe tables of more than one dimension by applying an OCCURS clause to a subordinate name. Standard COBOL limits tables to three-dimensions.

In the WORKING-STORAGE SECTION, initial values of elements within tables may be specified as follows. The table may be described as a record by a set of contiguous data description entries, each of which specifies the VALUE of an element, or part of an element, of the table. In defining the record and its elements, any data description clause (USAGE, PICTURE, etc.) may be used to complete the definition, where required. This form is required when the elements of the table require separate handling due to synchronization, USAGE, etc. The hierarchical structure of the table is then shown by use of the REDEFINES entry and its associated subordinate entries. The subordinate entries following the REDEFINES entry, which are repeated due to the OCCURS clause, must not contain VALUE clauses.

Example:

```
01  W-S-TOTS.
    03 FILLER PC X(24) VALUE IS ZEROS.
    03 CARDIMAGE-VALUES PC X(80).
01  R-TOTS REDEFINES W-S-TOTS.
    03 TOT PC 9(4) OCCURS 26 TIMES.
```

SUBSCRIPTING

Subscripts can be used only when reference is made to an individual element within a table of like elements that have not been assigned individual data-names. (Refer to the OCCURS clause.)

The subscript can be represented by a numeric literal that is an integer, or by a data-name. The data-name must be a numeric elementary item that represents an integer. The data-name may be qualified.

The subscript may be signed and if signed must be positive. However, the subscript cannot be computational-3 or J-signed. The lowest permissible subscript value is 1. This value points to the first element of the table. The next sequential elements of the table are pointed to by subscripts whose values are 2, 3, ... The highest permissible subscript value, in any particular case, is the maximum number of occurrences of the item as specified in the OCCURS clause. Violation of this rule will cause the object program to terminate with an INVALID SUBSCRIPT message.

The subscript, or a set of subscripts, identifying the table element is enclosed in parentheses. The table element data-name appended with a subscript is called a subscripted data-name or an identifier. When more than one subscript appears within a pair of parentheses, the subscripts may be separated by commas and are written in the order of successively less inclusive dimensions of the data organization.

The general construct for subscripting is:

$$\left\{\begin{matrix}\text{data-name}\\ \text{condition-name}\end{matrix}\right\} \quad \left(\text{subscript } [\text{,subscript}] \ \ldots\right)$$

For example, in figure 6-3, to reference the first volume, EN-VOLUME (1) is written. If data-name N contains the number of the volume desired, EN-VOLUME (N) is written. If the data item PAGE-NO contains the number of the page desired, then EN-HEADING (N, PAGE-NO) would reference the 12-character page heading.

Where qualification and subscripting are both required, the qualification is shown first, followed by the subscripting. For example, EN-PAGE OF ENCYCLOPEDIA (N, PAGE-NO). EN-PAGE (N, 3) OF ENCYCLOPEDIA is incorrect. For further restrictions, refer to the discussion of identifiers in this section.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM: MULTI-DIMENSIONED TABLE | COBOL DIVISION | PAGE 1 OF 1

PROGRAMMER: BEV | DATE | IDENT 73 80

PAGE NO. 1 3	LINE NO. 4 6	7	A 8 11	B 12 22 32 42 52 62 72
			01	ENCYCLOPEDIA.
			05	EN-VOLUME OCCURS 20 TIMES.
			10	EN-INDEX PIC X(6).
			10	EN-PAGE OCCURS 10 TIMES.
			15	EN-HEADING PIC XXXX.
			15	EN-PARAGRAPH OCCURS 5 TIMES.
			20	EN-TEXT PIC X(320).

Figure 6-3. Coding of Multi-Dimensioned Table

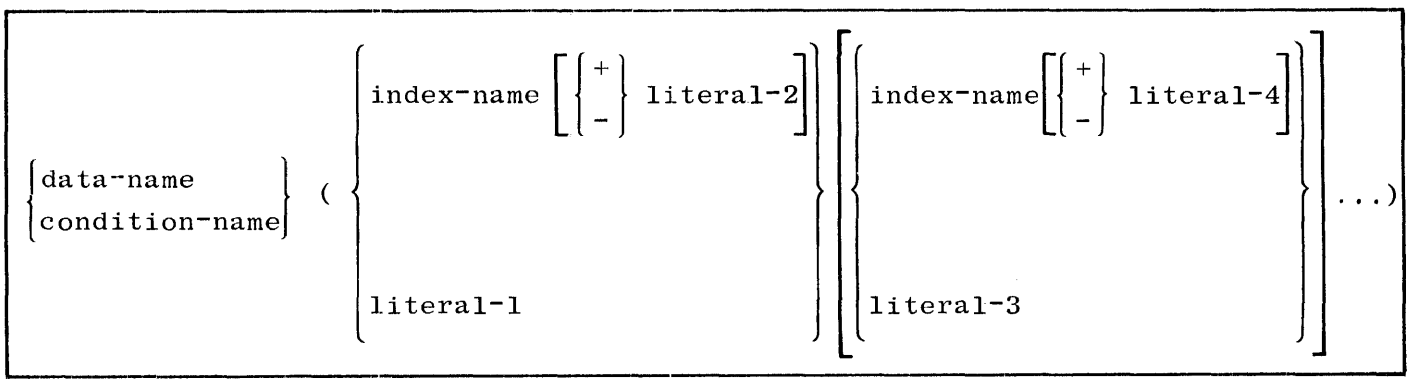
INDEXING

References can be made to individual elements within a table of like elements by specifying indexing for that reference. An index is assigned to that level of the table by using the INDEXED BY clause in the definition of a table. A name given in the INDEXED BY clause is known as an index-name and is used to refer to the assigned index. The value of an index corresponds to the occurrence number of an element in the associated table. An index must be initialized before it is used as a table reference. An index can be given an initial value by either a SET or a PERFORM statement.

Direct indexing is specified by using an index-name in the form of a subscript. Relative indexing is specified when an index-name is followed by the operator + or -, followed by an unsigned integer numeric literal all delimited by the balanced pair of separators left parenthesis and right parenthesis following the table element data-name. The occurrence number resulting from relative indexing is determined by incrementing (where the operator + is used) or decrementing (where the operator - is used), by the value of the literal, the occurrence number represented by the value of the index. When more than one index-name is required, they are written in the order of successively less-inclusive dimensions of the data organization.

At the time of execution of a statement which refers to an indexed table element, the value contained in the index referenced by the index-name associated with the table element must neither correspond to a value less than one (1) nor to a value greater than the highest permissible occurrence number of an element of the associated table. This restriction also applies to the value resultant from relative indexing.

The general construct for indexing is:



IDENTIFIER

An identifier is a term used to reflect that a data-name, if not unique in a program, must be followed by a syntactically correct combination of qualifiers, subscripts, or indices necessary to ensure uniqueness.

The construct for identifiers has two options which are as follows:

Option 1:

```
data-name-1  [{OF|IN}  data-name-2] ...
                    [( subscript-1 [, subscript-n]...)]
```

Option 2:

```
data-name-1  [{OF|IN}  data-name-2] ...
[ ( {index-name-1|literal-1}  [{+|-}  literal-2] ) [, {index-name-2|literal-3}  [{+|-}  literal-4]]... ]
```

Restrictions on qualification, subscripting, and indexing are as follows:

a. The commas as shown in both options are optional.

b. The data-name-2 must not itself be subscripted nor indexed.

c. Indexing is not permitted where subscripting is not permitted.

d. An index may be modified only by the SET, SEARCH, and PERFORM statements. Data items described by the USAGE IS INDEX clause permit storage of the values of index-names as data without conversion. Such data items are called index data items.

e. Where more than one occurrence number is required for a data-name reference, it is illegal to use a data-name for one occurrence number and an index-name for another. However, literals and index-names may be mixed.

FILE SECTION

This section contains descriptions of the files used by the object program.

FILE DESCRIPTION

The function of the FILE SECTION is to furnish information to the compiler concerning the physical structure, identification, and record names pertaining to a given file.

The construct of this section contains four options:

Option 1:

```
FD    file-name    COPY    library-name

[ REPLACING { word-1      } BY { word-2      }
            { data-name-1 }    { data-name-2 }
                               { literal-1   }

  [ { word-3      } BY { word-4      } ]... ]
    { data-name-3 }    { data-name-4 }
                       { literal-2   }
```

Option 2:

```
FD file-name-1  [ ;RECORDING MODE IS { ASCII      } ]
                                     { STANDARD   }
                                     { NON-STANDARD }

  [ ;FILE CONTAINS integer-1 [BY integer-2] { RECORDS  } ]
                                            { STATIONS }
                                            { STATION  }
                                            { QUEUES   }
                                            { QUEUE    }

  [ ;BLOCK CONTAINS [integer-3 TO] integer-4 [ RECORDS    ] ]
                                             [ CHARACTERS ]

  [ ;RECORD CONTAINS [integer-5 TO] integer-6 CHARACTERS ]

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

  [ ;{ VA    } OF ID IS { [literal-1/] [literal-2][/[literal-4]] }
     { VALUE }          { data-name-3                           }

      [ SAVE-FACTOR IS integer-7 ] ]

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

Option 3:

```
SD  sort-file-name COPY library-name

[ REPLACING  { word-1      }  BY  { word-2      }
             { data-name-1 }      { data-name-2 }
                                  { literal-1   }

          [ { word-3      }  BY  { word-4      } ] ... ] .
            { data-name-3 }      { data-name-4 }
                                 { literal-2   }
```

Option 4:

```
SD  sort-file-name

FILE CONTAINS integer-1 [BY integer-2]  RECORDS

[;RECORD CONTAINS integer-4 CHARACTERS ]

[;BLOCK CONTAINS  integer-6 [ RECORDS    ] ]
                            [ CHARACTERS ]

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

A level indicator of FD or SD identifies the beginning of a File Description or a Sort File Description and must precede the file statement. Both entries should commence under Area A of the coding form. Only one period is allowed in the entry and it must follow the last clause specified.

Options 1 and 3 can be used when the Systems library contains the library-name entry; otherwise, Option 2 and/or Option 4 must be used.

In many cases, the clauses within the File Description or Sort File Description sentence are optional. Their order of appearance is immaterial. Each clause is discussed in detail.

Figure 6-4 illustrates the use of the File Description sentence followed by data record entries.

NOTE

The three 01 levels implicitly redefine the record area. The DATA RECORDS clause is treated by the compiler as being for documentation purposes only and does not cause an explicit redefinition of the area.

BURROUGHS COBOL CODING FORM
ADDITIONS, DELETIONS AND CHANGES

PROGRAM: FILE SECTION EXAMPLE | COBOL DIVISION | PAGE 1 OF 1
PROGRAMMER: JULIE | DATE | IDENT 73 80

PAGE NO. 1 3	LINE NO 4 6	7	A 8 11	B 12 22 32 42 52 62 72
			FILE	SECTION.
			FD	MASTER-FILE BLOCK CONTAINS 3 RECORDS
				VALUE OF ID "PERS"/"MASTER" SAVE-FACTOR 10.
			01	EMPL-REC.
				05 EMPL-NUMBER PIC 9(8).
				05 DEPT PIC 999.
				05 FILLER PIC X(9).
				05 JOB-CODE PIC XXXX.
				05 FILLER PIC X(60).
			SD	EXTRACT-FILES FILE CONTAINS 10000 RECORDS.
			01	SORT-REC.
				05 S-JOB-CODE PIC XXXX.
				05 S-EMPL-NUMBER PIC 9(8).
				05 FILLER PIC X(68).
			FD	DEPT-REPORT VALUE OF ID "PERS"/"REPORT"
				DATA RECORD BODY-LINES HEADING-LINES.
			01	HEADING-LINES PIC X(132).
			01	BODY-LINES.
				05 JOB-CODE PIC X(4) B(4).
				05 EMPL-COUNT PIC Z(5)9.
				05 FILLER PIC X(118).

Figure 6-4. Coding of FD and DATA RECORDS

BLOCK

The function of this clause is to specify the size of a physical record (block).

The construct of this clause is:

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

Integer-1 and integer-2 must be positive integer values.

This clause is required if the block contains more than one logical record.

When only integer-2 is used, it will represent logically blocked, fixed-length, records if its value is other than 1. When the integer-1 TO integer-2 option is used, it will represent the minimum to maximum size of the physical record and indicates the presence of blocked variable-length records. Integer-1 is for documentation purposes only.

The maximum value of the integer used in this clause is shown in table 6-1 and refers to the number of characters in a block.

The word CHARACTERS is an optional word in the BLOCK clause. Whenever the key word RECORDS is not present, the integers represent characters.

For object program efficiency, the use of blocked records is recommended. The physical size of the block should be as large as possible depending on memory availability.

Blocks of records are read into the input buffer area by the MCP, and the delivery of each record to the record work-area of the program (required by an explicit READ statement) is completed.

Blocking or deblocking of records is automatically performed by the MCP.

NOTE

If the file is assigned to an input disk file and this clause is omitted, the blocking factor specified in the disk file header will be used by default.

Table 6-1. Maximum Value of Integers

I/O MEDIUM	MAXIMUM BLOCK SIZE - CHARACTERS
READER	80/96
PUNCH	80/96
TAPE	Limited only by the amount of memory available.
DISK	Limited only by the amount of memory available.
PRINTER	One print line.
PT-READER	Limited only by the amount of memory available.
PT-PUNCH	Limited only by the amount of memory available.

Every explicit WRITE statement causes compiler-generated object code to notify the MCP that a write is to be done. The MCP accumulates the number of logical records necessary to create a specified block size and writes the block. When a file is CLOSEd, the records left in the output buffer area, if not a full block, will be written as a short block by the MCP before the file is physically CLOSEd, The transfer of records to the buffer is automatic, and is a function of the MCP.

The user must specify the actual size of variable-length records in the first four bytes of each record. This four-character indicator is counted in the physical size of each record.

The BLOCK clause is not applicable to the READER, PT-PUNCH, or PT-READER peripherals.

This clause may be omitted for unblocked files.

When a file is assigned to disk, the user should be aware that the physical disk segment size is 180 bytes and that all READ and WRITE statements are, in effect, in multiples of this size. The hardware must write (or read) in segments; therefore, it is preferred that the block size used be a multiple of 180 bytes.

DATA RECORDS

The function of this clause is to document the names of the logical record(s) actually contained within the file being described.

The construct of this clause is:

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

This statement is only for documentation purposes. The compiler will obtain this information from 01 level record description entries.

The presence of more than one data-name indicates that the file contains more than one type of data record. These records may be of differing sizes, different formats, etc. The order in which they are listed is not significant

No syntax error will occur when a record declared for the file is not listed in the DATA RECORDS clause.

FILE CONTAINS

The function of this clause is to indicate the number of logical records in a file. This statement is required for disk files, and optional for all other files.

The construct of this clause is:

FILE CONTAINS [integer-1 BY] integer-2 { RECORDS | STATIONS | STATION | QUEUES | QUEUE }

The indicated integers must be positive values.

Integer-1 may not exceed 105 when present.

An entry of FILE CONTAINS 20 by 500 RECORDS will notify the MCP to allot 20 separate areas of disk as each area is programmatically required. The size of each area would be 500 logical records in length.

The above technique allows the MCP to efficiently assign file areas as needed, rather than to assign immediately one huge file area during the first operation of the program.

Programmatic usage of the file can either enhance the area technique or defeat its purpose completely. For example, assume that a RANDOM file at some future date will require a maximum size of 40 x 1584 (126,720) logical records, and that no key conversion formula is used, due to the key being a six-digit number running from 1 through 126,720, which exactly fills the key requirement, as is the case in auto license numbers in some states. It could happen that the first 40 records could open up an entire disk module, if they were in increments of 1584, which would negate the area technique completely and thus cause the MCP Disk Directory to recognize the file as being of maximum size, even though only 40 records were processed.

FILE CONTAINS integer-1 STATIONS must be specified if more than one station exists on this file. Otherwise, only one station will be enabled.

LABEL

The function of this clause is to specify the presence or absence of file label information as the first and last record of an input or output file.

The construct for this clause is:

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

STANDARD specifies that labels exist for the file or device to which the file is assigned. It also specifies that output labels conform to the standards as implemented.

STANDARD, when specified for disk files, indicates that the 20-character contents of the VALUE OF ID clause will be inserted into the disk file header. Should VALUE OF ID be omitted, the first 10 characters of the FD or SD file-name will be inserted into the second 10 characters of the disk file header. When the LABEL clause is not specified, LABEL RECORD STANDARD is assumed.

Data-name-1, data-name-2,..., are names of label records and must not appear in the DATA RECORDS clause, or be the subject of a record description associated with the file.

OMITTED specifies that physical labels do not exist for the specific input file to which the file is ASSIGNed. During object program execution, the operator will be queried by the MCP as to which unit possesses the input data. The operator must reply with "mix-index" UL "unit-mnemonic" control message.

OMITTED specifies that labels are not to be created for the specific output file ASSIGNed.

LABEL

The Burroughs Standard label record serves as both the beginning and ending label record, and is comprised of the following parts:

Position	Field Description
1	Always blank.
2-8	Always contains the literal "LABEL ".
9	Always contains zero.
10-16	Contains zeros, unless the file is a multifile tape (that is, a tape which may contain more than one file), in which case the field will contain the value of the identification of the multi-file, from one to seven characters.
17	Always contains zero.
18-24	Contains the value of the identification of the file, from one to seven characters. In a COBOL program, this value is taken from the VALUE OF ID clause in the File Description, or from the first seven characters of the FILE-NAME in the File Description if the clause has been ommited.
25-27	The value of the reel number is preset at "001" and incremented by 1 each time a subsequent reel is opened for this file.
28-32	The value of this field is taken from the current date as maintained by the MCP.
33-34	The value of cycle is preset to "00". This field may be used to distinguish between multiple runs of the same program, as controlled by a user program.
35-39	The date at which the MCP will assume this tape to be a scratch tape. If this date is reached, and the tape is mounted with a write ring in place, it is a contender for selection by the MCP as an output tape file, and could be over-written. This date, by default, is one day after the file was created (as taken from the MCP current date filed). To assign a save-factor of more than one day, refer to the SAVE-FACTOR option of the File Description.
40	Used only for ending labels, and enables the MCP to distinguish between the physical end of a reel (indicating that a subsequent reel or reels follow) and the actual end of a file. 0 = end-of-file. 1 = end-of-reel.
41-45	Used only in ending labels, and contains the number of blocks (physical records) written on the tape.
46-52	Used only for ending labels, and contains the number of records (logical records) written on the tape.
53	A value of 1 notifies the MCP to format the output into memory dump notation. This feature is not implemented.

Position	Field Description
54-58	Used to maintain a permanent serial number (usually a tape library reel number) for this reel. It may be assigned by the user, then permanently maintained by the MCP, regardless of the tape's status (in use, scratch, multifile reel, etc.).
59-63	Identifies the system which created this tape. When created on the B 1700, this value will always be " B 1700 ".
64-66	File buffer size in binary; for use by MCP if the DEFAULT option is specified.
67-69	Record size in binary; for use by MCP if the DEFAULT option is specified.
70-80	Reserved.

RECORD

The function of this clause is to specify minimum and/or maximum variable record lengths.

The construct of this clause is:

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

Integer-1 and integer-2 must be unsigned non-zero integer values.

If integer-1 and integer-2 are specified, the variable-length record technique is utilized.

If only integer-2 is specified, the compiler will treat the clause as being documented only. The record size will be determined by the structure of the record description.

If integer-1 and integer-2 are specified, they refer to the minimum and maximum size of the variable records to be processed. At least one record description must reflect the maximum size record length as specified in the RECORD CONTAINS clause.

The user must specify the actual size of variable-length records in the first four bytes of each record. The four-character variable-size indicator is counted in the physical size of each record.

This clause is applicable to disk or magnetic tape files sequentially OPENed INPUT or OUTPUT.

RECORDING MODE

The function of this clause is to specify the recording mode for peripheral devices, where a choice can be made.

The construct for this clause is:

$$\underline{\text{RECORDING}}\ \text{MODE IS}\ \left\{\begin{array}{l}\underline{\text{STANDARD}}\\ \underline{\text{NON-STANDARD}}\\ \underline{\text{ASCII}}\end{array}\right\}$$

STANDARD RECORDING MODE is assumed if this clause is absent from the FD sentence. The MCP automatically checks the parity of input magnetic tapes and will read the tape in the intelligent mode. For this reason, this clause is not required for input tapes.

The MCP will automatically assign STANDARD RECORDING MODE on 9-channel magnetic tape drives if a SELECT clause indicates TAPE, even though the programmer has designated the unit as being NON-STANDARD.

Binary files are read or written, with no possibility of translation.

The recording modes for the peripheral devices are provided in table 6-2.

Table 6-2. Recording Modes for Peripheral Devices

DEVICE	STANDARD	NON-STANDARD
TAPE-7	Odd Parity	Even Parity
TAPE-9	Odd Parity	-
DISK	Memory Image	-
READER	EBCDIC	Binary
PUNCH	EBCDIC or BCD	Binary
PT-READER	BCL	Binary
PT-PUNCH	BCL	Binary
PRINTER	BCL	-

VALUE OF ID

The function of this clause is to define the identification value assigned, or to be assigned, to a file of records and to declare the length of time that a file is to be saved.

The construct of this clause is:

$$\left\{\begin{matrix}\underline{\text{VALUE}}\\ \underline{\text{VA}}\end{matrix}\right\} \text{ OF } \underline{\text{ID}} \text{ IS } \left\{\begin{matrix}[\text{literal-1}/]\\ \text{data-name-1}\end{matrix}\right. \text{ [literal-2] } \left[/[\text{literal-3}]\right]\Big\}$$

$$[\underline{\text{SAVE-FACTOR}} \text{ IS integer-1}]$$

This clause may be used when the label records are present in the file being described. If this clause is not present, the compiler will take the VALUE OF ID from the first 10 characters of the file-name (FD or SD) and place that ID in the ID entry of the label where the value of the main directory entry would normally be found. The file-name must be uniquely constructed so that the MCP will be able to recognize the files.

Example:

```
FD  SCHEDULE-DISK1      Would create a VALUE OF ID as
FD  SCHEDULE-DISK2      SCHEDULE-D for both files and
                        cause a dup file action by the MCP.
```

To make them unique:

```
FD  DISKOUTPAY          Would create a VALUE OF ID as
FD  DISKOUTTAX          DISKOUTPAY and one of DISKOUTTAX,
                        thus causing no MCP confusion
                        during object program execution.
```

The first name for a magnetic tape file is a common name of a multi-file tape and the second name will be the name of a file within the multi-file. The first name of a magnetic tape file will be taken from the multi-file clause in the I-O-CONTROL paragraph. The second name will be taken from the value of literal-2. Non-disk files are limited to two names.

The pack-id name of a disk file will be taken either from the multi-file clause in the I-O-CONTROL paragraph, or from the value of literal-1. The main directory (family) name will be taken from literal-1 (in the case of systems disk or if I-O-CONTROL is used to specify user disk), from literal-2 (in the case of user disk without I-O-CONTROL or if literal-2 is followed by a slash (/)). The sub-directory entry (file-name) will be taken from the value of literal-3. Literal-3 cannot be used when literal-1 and literal-2 are both

blank. When using the literal option, if three literals are used, they represent pack-id, main directory (family), and sub-directory (file-name), respectively. If two literals are used they represent main directory and sub-directory. If only one literal is used it represents the main directory entry.

PACK-ID	MAIN DIRECTORY	SUB-DIRECTORY
[literal-1 /]	[literal-2]	[/ [literal-3]]
can be specified in I-O-CONTROL and forces literal-2 to be specified	can come from FD or SD name	forces literal-1 / and literal-2 to be specified

Examples:

```
VALUE OF ID IS "USER1"/"PAYROLL"/"DEDUCTS".
VALUE OF ID IS "WORKPACK1"/"TRANS"/.
VALUE OF ID IS "PAYROLL"/"MASTER".
VALUE OF ID IS "ITEMS".
VALUE OF ID IS "MSTTAPE" SAVE-FACTOR IS 031.
```

The data-name-1 option should only be used if file names are to be built under program control, as this option overrides file equates and I-O-CONTROL name assignments for that file. When data-name-1 is used it must be defined as being 30 characters in length and alphabetic or alphanumeric.

When the data-name-1 option is used for disk files, the disk-pack-id must be included in the description. The compiler will use the first 10 characters of the data-name as the disk-pack-id each time the file is opened. If the file is on or is to be created on systems disk, the first 10 characters must be blank.

```
01 DATA-NAME-1,                          Overrides I-O-CONTROL or use of
                                         FD or SD name for that file.
    03 PACK-ID  PC X(10).                Pack-id name for user disk must be
                                         blank for system disk or non-disk
                                         files.
    03 MAIN-DIRECTORY  PC X(10).         Cannot be blank at open time.
    03 SUB-DIRECTORY  PC X(10).          A non-blank entry here requires a
                                         non-blank entry for MAIN-DIRECTORY.
```

VALUE OF ID

Examples:

```
01 FILE-IDENTIFICATION.
     03  PACK-ID  PC X(10) VA "USER1 ØØØØØ".
     03  MAIN-DIRECTORY  PC X(10) VA "PAYROLL ØØØ".
     03  SUB-DIRECTORY  PC X(10) VA "DEDUCTS ØØØ."

01 DATA-NAME-1.
     03  PACK-ID                    PC  X(10) VA "WORKPACK1Ø".
     03  MAIN-DIRECTORY             PC  X(10) VA "TRANSØØØØØ".
     03  SUB-DIRECTORY              PC  X(10) VA SPACES.

01 FILE-ID.
     03  PACK-ID                    PC  X(10) VA SPACES.
     03  MAIN-DIRECTORY             PC  X(10) VA "PAYROLLØØØ".
     03  SUB-DIRECTORY              PC  X(10) VA "MASTERØØØØ".

01 VA-NAME.
     03  PACK-ID                    PC  X(10) VA SPACES.
     03  MAIN-DIRECTORY             PC  X(10) VA "ITEMSØØØØØ".
     03  SUB-DIRECTORY              PC  X(10) VA SPACES.

01 SOME-DATA-NAME.
     03  BACKUP-PACK-NAME           PC  X(10) VA SPACES.
     03  WHICH-SYSTEM               PC  X(10) VA SPACES.
     03  FOR-WHAT-DAY               PC  X(10) VA SPACES.
```

NOTE

Names must be moved in prior to OPEN.

A file with one name (main directory name) will be placed in the main directory by means of a scramble technique. The address following the name in the directory will point to the disk file header. A file with two names adds another level to the directory. The first name is the family or main directory name. The main directory name will be scrambled to a directory with the file-type set to "2". The "2" designates that the address following the name is the address of a sub-directory. The second name or sub-directory name is then placed in this additional directory. The address in the sub-directory now points to the disk file header of the file. The sub-directory entry will not be scrambled into the directory, as is the main directory entry which has the location of the sub-directory. When the MCP finds the sub-directory, it must search for the sub-directory file-name.

The VALUE OF ID declared for OUTPUT disk files will cause up to 20 characters to be inserted into the disk file header. Inversely, up to 20 characters will be checked against the MCP Disk File Directory to obtain the physical disk location of the file when declared as being INPUT or INPUT-OUTPUT disk files. The (PACK-ID) is carried in the file parameter block FPB or in the INPUT-OUTPUT disk files.

When SAVE-FACTOR is specified for output magnetic tape files integer-1 represents the number of days the file is to be saved before it can be purged and used for other purposes by the system; integer-1 is limited to an unsigned integer not to exceed three digits in length with values from 001 to 999.

SAVE-FACTOR, when declared for a disk file, is for documentational purposes, due to the fact that files residing on disk should only be purged by mutual consent within an EDP organization and can only be performed as a physical action by the systems operator on the automatic RMOV option of MCP.

If SAVE-FACTOR is not specified, tapes are automatically assigned a SAVE-FACTOR of one day to preclude expiration action when the system is being operated during the period just prior to midnight or thereafter.

NOTE

For magnetic tape file names, the names must be unique in the first seven characters of each name.

RECORD DESCRIPTION

This portion of a COBOL source program follows the file description entries and serves to completely identify each data element within a record of a given file.

The construct of these entries contain the following four options:

Option 1:

```
01 data-name-1; COPY library-name

[ REPLACING { word-1      } BY { word-2      }
            { data-name-2 }    { data-name-3 }
                               { literal-1   }

  [ ,{ word-3      } BY { word-4      } ]  ... ] .
     { data-name-4 }   { data-name-5 }
                       { literal-2   }
```

Option 2:

```
level-number { FILLER      } [;REDEFINES data-name-2 ]
             { data-name-1 }

[;{ PC      } IS (allowable PICTURE characters) ]
  { PIC     }
  { PICTURE }

[                  { DISPLAY                }  ]
[                  { CMP                    }  ]
[                  { CMP-1                  }  ]
[                  { CMP-3                  }  ]
[                  { COMP                   }  ]
[;[USAGE IS ]      { COMP-1                 }  ]
[                  { COMP-3                 }  ]
[                  { COMPUTATIONAL          }  ]
[                  { COMPUTATIONAL-1        }  ]
[                  { COMPUTATIONAL-3        }  ]
[                  { INDEX                  }  ]
[                  { ASCII                  }  ]

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

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

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

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

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

[; { BZ                  }]
   { BLANK WHEN ZERO     }

[;{ VA    } [IS ]literal-1 ] .
  { VALUE }  [ARE]
```

Option 3:

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

Option 4:

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

[,literal-3  [{ THRU    } literal-4] ] ... .
              { THROUGH }
```

The optional clauses shown may occur in any order, with the exception that if REDEFINES is used it must follow data-name-1.

The record description must be terminated by a period.

Level-numbers in Option 2 may be any number from 1-49 or 77. The optional clauses may be written in any order, with two exceptions: the data-name-1 or FILLER clause must immediately follow the level-number; the REDEFINES clause, when used, must immediately follow the data-name-1 clause.

The clauses PICTURE, BLANK WHEN ZERO, JUSTIFIED, and SYNCHRONIZED must occur on elementary item level only.

The PICTURE clause must be specified for every elementary item except an index data item, in which case use of the clause is prohibited.

Option 1 can be used when the COBOL library contains the record description entry. Otherwise, one of the other options must be used.

In Option 4, there is no practical limit to the number of literals in the condition-name series.

The SYNCHRONIZED clause is for documentation only.

BLANK WHEN ZERO

The function of this clause permits the blanking of an item when its value is zero.

The construct of this clause is:

$$\left\{ \begin{array}{l} \underline{\text{BZ}} \\ \underline{\text{BLANK}} \text{ WHEN } \underline{\text{ZERO}} \end{array} \right\}$$

BLANK WHEN ZERO may be abbreviated BZ.

This clause overrides the zero-suppress float-sign functions in a PICTURE. If the value of a field is all zeros, the BZ clause will cause the field to be edited with spaces. However, it does not override the check protect function (zero suppression with asterisks) in a PICTURE.

The BZ clause can only be used in conjunction with an item on an elementary level.

BLANK WHEN ZERO may be associated only with PICTUREs describing numeric or numeric edited fields.

The category of the item is considered to be numeric edited.

CONDITION-NAME

Condition-name is a special name which the user may assign to a value or values within a data element. This value may then be referred to by the specified condition-name.

The construct of this clause is:

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

[,literal-3  [ { THRU    }  literal-4 ] ] ...
               { THROUGH }
```

Since the testing of data is a common data processing practice, the use of conditional variables and condition-names supplies a shorthand method which enables the writer to assign meaningful names (condition-names) to particular code values that may appear in a data-field (conditional variable).

A condition-name can be associated with any item containing a level-number, except the following:

a. Another condition-name.

b. A level 66 item.

c. A group containing items with descriptions including JUSTIFIED, or USAGE (other than USAGE IS DISPLAY).

d. An index data-item.

When defining condition-names, the following rules must be observed:

a. If reference to a conditional variable requires subscripting, then references to its condition-names also require subscripting.

b. A conditional variable may be used as a qualifier for any of its condition-names.

c. Condition-names can only appear in conditional statements.

d. Whenever the THRU phrase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.

e. The characteristics of a condition-name are implicitly those of its conditional variable.

The following example illustrates a conditon-name. If THIS-YEAR identifies the 12 months of a year, whereas its subordinate data items are defined as JANUARY, FEBRUARY, etc., and the values assigned to each month range from 01 to 12, then it follows that JUNE would have the assigned value of 06. Using the condition-name JUNE, the programmer can utilize it in conditional statements as follows:

```
IF JUNE GO TO . . . .
```

which is logically equivalent to the statement:

```
IF THIS-YEAR IS EQUAL TO 06 GO TO . . . .
```

BURROUGHS COBOL CODING FORM
ADDITIONS, DELETIONS AND CHANGES

PROGRAM: CONDITION NAMES | COBOL DIVISION | PAGE 1 OF 1
PROGRAMMER: DAVID | DATE | IDENT 73 80

PAGE NO. 1 3 | LINE NO. 4 6 | A 8 11 | B 12 22 32 42 52 62 72 | Z

```
05  GRADE                PIC 99.
    88  FIRST-GRADE        VA 1.
    88  SECOND-GRADE       VA 2.
    88  THIRD-GRADE        VA 3.
    88  FOURTH-GRADE       VA 4.
    88  FIFTH-GRADE        VA 5.
    88  6TH-GRADE          VA 6.
    88  7TH-GRADE          VA 7.
    88  8TH-GRADE          VA 8.
    88  9TH-GRADE          VA 9.
    88  10TH-GRADE         VA 10.
    88  11TH-GRADE         VA 11.
    88  12TH-GRADE         VA 12.
    88  GRADE-SCHOOL       VA 1 THRU 6.
    88  JR-HIGH            VALUE 7, 8, 9.
    88  HIGH-SCHOOL        VALUE 10 THRU 12.
    88  GRADE-ERROR        VALUE 13 THRU 99, 0.
```

Figure 6-5. Coding of Condition-Name

DATA-NAME

The purpose of this mandatory clause is to specify the name of each data element to be used in a program. If a data element requires a definite label, a data-name is assigned. Otherwise, the word FILLER can be used in its place.

The construct of this clause is:

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

The word FILLER can be used to name a contiguous description area that does not require programmatic reference.

This entry must immediately follow a level-number other than an 88 level. FILLER is only applicable to elementary levels.

A data-name need not be unique if it can be made unique through qualification by use of data-names on higher levels than itself.

JUSTIFIED

The JUSTIFIED clause specifies non-standard positioning of data within a receiving data item.

The format for the JUSTIFIED clause is as follows:

$$\left\{\begin{matrix}\underline{\text{JUSTIFIED}}\\ \underline{\text{JUST}}\end{matrix}\right\} \text{ RIGHT}$$

The JUSTIFIED clause cannot be specified for a numeric-edited data item or for an item described as numeric. The JUSTIFIED clause cannot be specified for an item whose size is variable, for group items or for an index-data-name.

The following are the standard rules for positioning within an area:

a. Numeric data is aligned by decimal point (either implicit or explicit), with zeros filling any unused positions on either end, as required. In the absence of an explicit decimal point indication, the decimal point is assumed to be in the next position to the right of the units digit. Edited numeric data items are aligned by decimal point, with zero fill or truncation at either end as required within the receiving character positions of the data item, except where editing requirements cause replacement of the leading zeros.

b. Alphabetic or alphanumeric receiving data items are aligned at the leftmost character position in the data item, with space fill or truncation to the right.

When the receiving data items are described with the JUSTIFIED clause and it is larger than the sending item, the data is aligned at the rightmost character position in the data item, with leading space fill.

Example:

SENDING						RECEIVING							
PIC X(5)	A	1	2	3	C	PIC X(7)			A	1	2	3	C

When the receiving item is described with the JUSTIFIED clause and it is smaller than the sending item, the left-most characters are truncated.

Example:

SENDING								RECEIVING					
PIC X(7)	A	1	2	3	C	D	E	PIC X(5)	2	3	C	D	E

If JUSTIFIED RIGHT is specified for an alphabetic or alphanumeric item, data is placed into the area, with space fill to the left.

If JUSTIFIED RIGHT is specified for an alphabetic or alphanumeric item and the receiving field is smaller than the sending field, truncation will occur from the left.

When standard justification is desired, the JUSTIFIED clause is not required. Justification is considered only when data is moved into an area.

LEVEL-NUMBER

The function of this clause is to show the hierarchy of data within a logical record. Its further function is to identify entries for condition-names, non-contiguous constants, working-storage items, and for re-grouping.

The construct of this clause is:

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

A level-number is the first required element of each record and data-name description entry.

Level-numbers may be as follows:

a. 01 to 49 - record description and WORKING-STORAGE entries.

b. 66 - RENAMES clause used as a record description or WORKING-STORAGE entry.

c. 77 - applicable to WORKING-STORAGE only as non-contiguous items and must precede all other level-numbers.

d. 88 - condition names clause used as a record description or WORKING-STORAGE entry.

Level-numbers 01 through 49 are used for record or WORKING-STORAGE descriptions. Level number 01 is reserved for the first entry within a record description. Level-number 66 is reserved for RENAMES entries. Level-number 77 is used for miscellaneous elementary items in the WORKING-STORAGE SECTION when these items are unrelated to any record. They are called non-contiguous items since it makes no difference as to the order in which they actually appear. Level-number 88 is used to define the entries relating to condition-names in record descriptions or WORKING-STORAGE entries.

For additional information on level-numbers, see LEVEL NUMBER CONCEPT.

OCCURS

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

The construct for this clause has the following two options:

Option 1:

```
{OC      }
{OCCURS  }  integer-2 TIMES

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

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

Option 2:

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

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

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

Integer-1 and integer-2 must be positive integers. If both are used, the value of integer-1 must be less than integer-2. The value of integer-1 may be zero, but integer-2 cannot be zero.

The data description of data-name-1 must describe a positive integer.

Data-name-2 must either be 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 which is the subject of this entry.

Data-name-1, data-name-2, and data-name-3 may be qualified.

The OCCURS clause cannot be specified in a data description that:

a. Has an 01, 66, 77, or 88 level-number.
b. Describes an item whose size is variable. The size of an item is variable if its data description, or any item subordinate to it, contains option 2 of the OCCURS clause.

OCCURS

The OCCURS clause is used in defining tables and other homogeneous sets of repeated data. Whenever the OCCURS clause is used, the data-name which is the subject of this entry must be either subscripted or indexed whenever it is referred to in a statement other than SEARCH. Further, if the data-name associated with the OCCURS clause is the name of a group item, then all data-names belonging to the group must be subscripted or indexed whenever they are used as operands.

Except for the OCCURS clause itself, all data description clauses associated with an item whose description includes an OCCURS clause applies to each occurrence of the item described.

In option 1, the value of integer-2 represents the exact number of occurrences of items within the table.

In option 2, the value of integer-1 represents the minimum number of occurrences, and integer-2 represents the maximum number of occurrences. This does not imply that the length of the table is variable but that the number of occurrences is variable. When option 2 is specified in a data description entry, only items subordinate to the data item described with the option 2 OCCURS may follow in the Record Description. Thus, the following is illegal:

```
01 DATA-1.
    05  TAB-1 OCCURS 1 TO 50 DEPENDING ON CNT.
        10 TAB-2 PIC 9(5).
    05  TAB-3 PIC  9(5).
```

Any unused character positions resulting from the DEPENDING option will appear in the external media.

The DEPENDING option is for documentation and serves only to document the end of the occurrences of data items. The value of data-name-1 is the count of the number of occurrences of items, and its value should not exceed integer-2. The user must employ his own tests to determine how many occurrences of the item are actually valid and present in the record.

If data-name-1 in the DEPENDING option is an entry in the same record as the current data description entry, data-name-1 should not be the subject of, or be subordinate to, an entry whose description includes option 2 of an OCCURS clause.

An entry which contains option 2, or has a subordinate entry which contains option 2, cannot be the object of the REDEFINES clause. For example, the following is illegal:

```
01  W-S-TABLE.
        02 TAB-SIZE 5 OCCURS 1 TO 5 TIMES DEPENDING ON DEP-NAME.
        02 RED-TAB REDEFINES TAB.
```

When integer-2 and integer-3 are both specified, variable length records are implied and the user must specify the actual size of variable-length records in the first four bytes of each record. The four-character variable size indicator is counted in the physical size of each record.

The KEY IS option is used to indicate that the repeated data is arranged in ascending or descending order according to the values contained in data-name-2, data-name-3, and so on. The data-names are listed in descending order of their significance.

If data-name-2 is not the subject of this entry, then the following applies:

a. All of the items identified by the data-names in the KEY IS phrase must be within the group item which is the subject of the OCCURS entry.
b. None of the items identified by data-names in the KEY IS phrase can be described by an entry which either contains an OCCURS clause or is subordinate to an intervening entry which contains an OCCURS clause.

The following example illustrates a use of the OCCURS clause to provide nested descriptions. A reference to ITEM-4 requires the use of three levels of subscripting; e.g., ITEM-4 (2, 5, 4). A reference to ITEM-3 requires two subscripts; e.g., ITEM-3 (I,J).

```
.          .          .
.          .          .
.          .          .
2 ITEM; OCCURS 2 TIMES;...
    3 ITEM-1;...
    3 ITEM-2; OCCURS 5 TIMES;...
        4 ITEM-3;...
        4 ITEM-4; OCCURS 5 TIMES;...
            5 ITEM-5;...
            5 ITEM-6;
            .          .          .
            .          .          .
            .          .          .
```

In the example above, there are 50 ITEM-4 quantities.

The following example shows another use of the OCCURS clause. Assume that the user wishes to define a record consisting of five AMOUNT items, followed by five TAX items. Instead of the record being described as containing 10 individual data items, it could be described in the following manner:

```
1 TABLE;...
    2 AMOUNT; OCCURS 5 TIMES;...
    2 TAX; OCCURS 5 TIMES;...
    .          .          .
    .          .          .
    .          .          .
```

The above definition would result in memory allocated for five AMOUNT fields and five TAX fields. Any reference to these fields is made by addressing the field by name AMOUNT or TAX followed by a subscript denoting the particular occurrence desired. (See the discussion on subscripts, page 6-12.)

An INDEXED BY clause is required if the subject of this entry, or an item within it, is to be referred to by indexing. If indexing is to be used, each table dimension must contain an INDEXED BY clause. The index-names identified by the clause must not be defined elsewhere in the program and must be unique.

The ASCENDING/DESCENDING KEY option is for documentation only.

The operands in the INDEXED BY option are index-names or indices. The operands of an INDEXED BY option must appear in association with an OCCURS clause and are usable only when referencing that level of the table. In the use of three-level indexing, each level must have an INDEXED BY option and in a given indexing operation, only one operand from each option may be used.

Other than its use as an index into an array, an index-name may be referred to only in a SET, SEARCH, PERFORM, or in a relation condition. All index-names must be unique. Index-names have an assumed construction of PC S9(6) COMPUTATIONAL.

Using an index-name associated with one row of a table for indexing into another row of a table will not cause a syntax error, but will, in most cases, cause incorrect object-time results, since it is the index-name that contains the information pertinent to the element sizes.

When using an index-name series (e.g., INDEXED BY A, B, C):

a. The indexes should be used only when referencing the associated row.

b. All "assumed" references are to the first index-name in a series. Others in the series are affected only during an explicit reference.

Indexing into a table follows much the same logic as subscripting. There is a limit of three indexes per operand (e.g., A (INDEX-1, INDEX-2, INDEX-3)). The use of a relative index allows modification of the index-name without actually changing the value of the index-name.

Example:

```
A (INDEX-1 + 3, INDEX-2 - 4, INDEX-3)
```

An index-name followed by a + or - integer indicates relative indexing, which causes the affected index to be incremented or decremented by that number of elements within the table.

A data-name whose USAGE is defined to be INDEX is an index-data-name.

Condition-names, PICTURE, VALUE, SYNCHRONIZED, or JUSTIFIED cannot be associated with an index-data-name.

The COBOL compiler will assign the construction of a PCS9(6) COMPUTATIONAL area for each index-data-name specified.

It is not permissible to relationally compare an index-data-name against a literal or against a regular data-name.

PICTURE

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

The general construct for the PICTURE clause is as follows:

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

The following are rules for the PICTURE clause:

a. A PICTURE clause can only be used at the elementary item level.

b. A character-string consists of certain allowable combinations of characters in the COBOL character set used as symbols. The allowable combinations determine the category of the elementary item.

c. The maximum number of symbols allowed in the character-string is 30. When an unsigned integer enclosed in parentheses immediately follows a symbol, the integer specifies the number of consecutive occurrences of that symbol. This may not be used for those symbols limited to one occurrence per picture.

d. A PICTURE clause must appear in every elementary item except those items whose USAGE is declared as INDEX.

Record descriptions do not have to conform to the physical characteristics of an ASSIGNed hardware-name. The flow of input-output data will terminate at the end of the prescribed PICTURE size. For example:

READER (can read 80 columns) description can be PICTUREd from 1 through 80.

PUNCH (can punch 80 columns) description can be PICTUREd from 1 through 80.

CARD96 (can read or punch 96 columns) description can be PICTUREd from 1 through 96.

PRINTER (120/132 character lines) description can be PICTUREd from 1 through maximum.

Categories of Data

There are five categories of data that can be described with a PICTURE clause: alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited. These categories are described as follows:

ALPHABETIC

To define an item as alphabetic, its PICTURE character-string can only contain the symbol A, and its contents, when represented externally, must be any combination of the 26 letters of the alphabet and the space from the COBOL character set.

NUMERIC

To define an item as numeric, its PICTURE character-string can only contain the symbols 9, P, S, J, K, and V. Its contents, when represented externally, must be a combination of the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The item may include one operational sign.

ALPHANUMERIC

To define an item as alphanumeric, its PICTURE character-string is restricted to certain combinations of the symbols A, X, 9, and the item is treated as if the character-string contained all X's. Its contents, when represented externally, are any of the allowable characters in the COBOL character set. A PICTURE character-string which contains all 9's or all A's does not define an alphanumeric item.

ALPHANUMERIC EDITED

To define an item as alphanumeric edited, its PICTURE character-string is restricted to certain combinations of the symbols A, X, 9, B, and 0 (zero) given by the following rules:

a. The character-string must contain at least one B and one X, or at least one 0 (zero) and one X, or

b. The character-string must contain at least one 0 (zero) and one A.

NUMERIC EDITED

To define an item as numeric-edited, its PICTURE character-string is restricted to certain combinations of the symbols B, P, V, Z, 0, 9, , (comma), . (period), *, +, -, CR, CB, and the currency sign ($). The PICTURE character string must contain at least one symbol other than V and 9. The allowable combinations are determined from the order of precedence of symbols and the editing rules.

Classes of Data

The five categories of data items are grouped into three classes: Alphabetic, Numeric, and Alphanumeric. For Alphabetic and Numeric, the classes and categories are synonymous. The Alphanumeric class includes the categories of Alphanumeric Edited, Numeric Edited and Alphanumeric (without editing). Every

elementary item belongs to one of the classes and further to one of the categories. The class of a group item is treated at object time as Alphanumeric regardless of the class of elementary items subordinate to that group item. Figure 6-6 depicts the relationship of the class and categories of data items.

LEVEL OF ITEM	CLASS	CATEGORY
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric-editied Alphanumeric-edited Alphanumeric
Non-elementary (Group)	Alphanumeric	Alphabetic Numeric Numeric-edited Alphanumeric-edited Alphanumeric

Figure 6-6. Relationship of Class and Category

Function of the Editing Symbols

An unsigned non-zero integer which is enclosed in parentheses following the symbols A, X, 9, P, Z, *, B, 0, +, -, the comma, or the currency sign ($) indicates the number of consecutive occurrences of the symbol. Note that the following symbols may appear only once in a given PICTURE clause: S, J, V, K, (period), CR, and DB.

The functions of the symbols used to describe an elementary item are explained as follows:

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

B Each symbol B in the character-string represents a character position into which the space character will be inserted.

J The symbol J indicates an operational sign appearing as an overpunch in the least-significant position for DISPLAY or as a trailing digit in CMP. J is not allowed for CMP-3. J is not counted in the size for DISPLAY but is counted in CMP. Only one operational sign may be present in each PICTURE. J and S are mutually exclusive. See the S sign discussion for the exact bit configuration of signs.

NOTE

If J appears as other than the leftmost character in a PICTURE character string, it no longer performs as an operational sign but serves to reinitiate zero suppression. J represents a character position and is counted in the length of the elementary item.

P The letter P indicates an assumed decimal scaling position and is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character P is not counted in the length of the data item. Scaling position characters are counted in determining the maximum number of digit positions (160) in numeric edited items or NUMERIC items which appear as operands in arithmetic statements. The scaling position character P can appear only to the left or right as a continuous string of P's within a PICTURE description. Since the scaling position character P implies an assumed decimal point (to the left of P if P's are leftmost PICTURE characters, and to the right of P if P's are rightmost PICTURE characters), the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description. The character P and the insertion character "." (decimal point) cannot both occur in the same PICTURE character string.

S The letter S is used in a character-string to indicate the presence of an operational sign and must be written as the leftmost character in the PICTURE. The S is not counted in determining the length of the elementary item unless USAGE is CMP. If USAGE is DISPLAY, S indicates the sign is carried as an overpunch in the most-significant position. J and S are mutually exclusive. For CMP, S indicates the sign is carried in the leading digit of the field. The four zone bits in EBCDIC and CMP are set to a "D", for negative, and to a "C" for positive. Wherever possible, PICTURE S should be used rather than J or K.

NOTE

Any value other than D will be assumed positive.

K The letter K in the character string indicates the presence of an 8-bit (byte) sign appearing in the leftmost character position of an item when USAGE is implicitly or explicitly DISPLAY and is counted in the length of the item. If USAGE IS COMPUTATIONAL, the letter K

becomes the same as an S. Data elements requiring a K PICTURE clause may not be described by a VALUE clause with a signed literal.

V The letter V is used in a character-string to indicate the location of the assumed decimal point and may only appear once in a character-string. The V does not represent a character position and, therefore, is not counted in the length of the elementary item. When the assumed decimal point is to the right of the rightmost symbol in the string, the V is redundant.

X Each letter X in the character-string is used to represent a character position which contains any allowable character from the computer's character set.

Z Each letter Z in a character-string may only be used to represent the leftmost leading numeric character positions which will be replaced by a space character when the contents of the character position is zero. Each Z is counted in the length of the item. Zero suppression is terminated with the first non-zero numeric character in the data. Insertion characters are also replaced by spaces while suppression is in effect. Z can also appear to the right of J, when the J symbol is used to reinitiate zero suppression. For additional information on zero suppression, see the BLANK WHEN ZERO clause.

9 Each 9 in the character-string represents a character position which contains a numeral and is counted in the length of the item. If USAGE is explicitly or implicitly DISPLAY, the data will be operated on as 8-bit (BYTE) characters. If USAGE is CMP, it will be operated on as 4-bit digits.

0 Each 0 (zero) in the character-string represents a character position into which the numeral zero will be inserted. When that item is receiving field, the 0 is counted in the length of the item.

, Each comma in the character-string represents a character position into which the comma character will be inserted. This character position is counted in the length of the item. See DECIMAL-POINT IS COMMA.)

. When the character period appears in the character-string, it is an editing symbol which represents the decimal point for alignment purposes; in addition, it represents a character position into which the period character will be inserted. The period character is counted in the length of the item. For a given program, the functions

of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. In this exchange, the rules for the period apply to the comma and the rules for the comma apply to the period whenever they appear in a PICTURE clause. V and (.) are mutually exclusive.

+, -, CR, CB — The symbols +, -, CR, and CB are used as editing sign control symbols. When used, they represent the character position(s) into which the editing sign control symbol will be placed. The symbols are mutually exclusive in any one character-string, and each character used in the symbol is counted in determining the length of the data-item. (Note that the symbols CR and DB are two character symbols, and any other use of C or D constitutes an error.)

* Each * symbol in the character-string represents a leading numeric character position into which an asterisk will be placed when the contents of that position is zero. Each * is counted in the length of the item. Asterisk replacement is disabled when the first non-zero character is encountered, or when the decimal point (implicit or explicit) is reached. When the PICTURE character string specifies only asterisks (*), and the value of the item is zero, the entire output item will consist of asterisks and the decimal point, if present. BLANK WHEN ZERO does not override the insertion of asterisks.

$ The currency symbol ($) in the character-string represents a character position into which a currency symbol is to be placed. The currency symbol in a character-string is represented by either the dollar sign ($) symbol or by the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the length of the item.

\- The symbol - when not the leftmost or rightmost character, is treated as a fixed insertion hyphen. This feature is valid only to the left of the decimal if the preceding character is not the symbol Z.

NOTE

Any other character which is not a defined picture character appearing in the PICTURE is assumed to be an insert character.

Example

99/99/99 could be a date mask and
999-99-999 could represent a social security number mask.

Editing Rules

There are two general methods of performing editing in the PICTURE clause: by insertion or by suppression and replacement.

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

The type of editing which may be performed upon an item is dependent upon the category to which the item belongs. Figure 6-7 specifies which type of editing may be performed upon a given category.

CATEGORY	TYPE OF EDITING
Alphabetic	None
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple Insertion, 0 and B
Numeric Edited	All, Subject to Note Above

Figure 6-7. Permissible Editing Types

Insertion Editing. The following are the four types of insertion editing available:

a. Simple Insertion.
b. Special Insertion.
c. Fixed Insertion.
d. Floating Insertion.

Simple Insertion Editing. The comma (,), B (space), and 0 (zero) are used as the insertion characters. The insertion characters are counted in the length of the item and represent the position in the item into which the character will be inserted.

Special Insertion Editing. The period (.) is used as the insertion character. In addition to being an insertion character, it also represents the decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the length of the item. The use of the assumed decimal point (represented by the symbol V) and the actual decimal point represented by the insertion character) in the same PICTURE character-string is prohibited. If the

insertion character is the last symbol in the character-entry, the character-string must be immediately followed by the semicolon punctuation character, and then followed by a space. If the PICTURE clause is the last clause of that DATA DIVISION entry, and the insertion character is the last symbol in the character-string, the insertion character must be immediately followed by a period punctuation character followed by a space. This results in two consecutive periods (or ",." if DECIMAL POINT IS COMMA has been specified) appearing in the data description entry. The result of special insertion editing is the appearance of the insertion character in the item in the same position as shown in the character-string.

Fixed Insertion Editing. The currency sign ($) and the editing sign control symbols "+", "-", CR, and DB are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE character-string. When the symbols CR or DB are used, they represent two character positions in determining the length of the item, and they must represent the rightmost character positions that are counted in the size of the item. The character "-" may be used as a fixed or floating sign insertion character. When this character appears to the left of the decimal point, its use as either a sign or a hyphen is determined as follows: if the character cannot be legally used as a sign according to the usual rules, then it is interpreted as a hyphen. To the right of the decimal point, it is only interpreted as a sign. The symbol "+", when used, must be the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of the item except that it can be preceded by either a + or a - symbol. Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character-string. Depending upon the value of the data item, editing sign control symbols produce the results indicated in table 6-3.

Table 6-3. Editing Symbols and Results

EDITING SYMBOL IN PICTURE CHARACTER-STRING	DATA ITEM POSITIVE	DATA ITEM NEGATIVE
+	+	-
-	SPACE	-
CR	2 SPACES	CR
DB	2 SPACES	DB

Floating Insertion Editing. The currency symbol and editing sign control symbols + or - are the insertion characters, and they are mutually exclusive as floating insertion characters in a given PICTURE character-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; however, they represent themselves rather than numeric character positions.

In the PICTURE character-string, there are only two ways of representing floating insertion editing. One way is to represent any or all of the leading numeric character positions to the left of the decimal point by the insertion character. The other way is to represent all of the numeric character positions in the PICTURE character-string by the insertion character.

The result of floating insertion editing depends upon the representation in the PICTURE character-string. If the insertion characters are only to the left of the decimal point, the result is a single insertion character that will be placed into the character position immediately preceding the decimal point, or the first non-zero digit in the data represented by the insertion symbol string, whichever is further to the left in the PICTURE character-string.

If all numeric character positions in the PICTURE character-string are represented by the insertion character, the result depends upon the value of the data. If the value is zero, the entire data item will contain spaces. If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point.

To avoid truncation, the minimum size of the PICTURE character-string for the receiving data item must be the number of characters in the sending data item, plus the number of fixed insertion characters being edited into the receiving data item, plus one for the floating insertion character.

Suppression Editing. The suppression of leading zeros in numeric character positions is indicated by the use of the character Z or the character * (asterisk) as suppression symbols in a PICTURE character-string. These symbols are mutually exclusive in a given PICTURE character-string. Each suppression symbol is counted in determining the length of the item. If Z is used, the replacement will be the space, and if the asterisk is used, the replacement character will be the *.

Zero suppression and replacement are indicated in a PICTURE character-string by using a string of one or more of the allowable symbols to represent leading numeric character positions which are to be replaced when the associated character position in the data contains a zero. Any of the simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string.

In a PICTURE character-string, there are only two ways of representing zero suppression. One way is to represent by suppression symbols, any or all of the leading numeric character positions to the left of the decimal point. The other way is to represent all of the numeric character positions in the PICTURE character-string by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which corresponds to a 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 string or at the decimal point, whichever is encountered first.

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

When the asterisk is used as the zero suppression symbol and the clause BLANK WHEN ZERO also appears in the same entry, the zero suppression editing overrides the function of BLANK WHEN ZERO.

Replacement Editing. Symbols +, -, *, Z, and the currency symbol, when used as floating replacement characters, are mutually exclusive within a given character string. At least two floating replacement characters must appear as the leftmost characters in the PICTURE.

Precedence of Symbols

Table 6-4 shows the order of precedence when characters are used as symbols in a character string. An X at an intersection indicates that the symbol(s) at the top of the column may precede, in a given character string, the symbol(s) at the left of the row. Arguments appearing in braces indicate that the symbols are mutually exclusive. The currency symbol is indicated by the symbol "cs".

At least one of the symbols "A", "X", "Z", "9", or "*", or at least two of the symbols "+", "-" or "cs" must be present in a PICTURE string.

When "+" or +-+ is to be the rightmost printable character in a PICTURE character(s) P, if any, must follow, instead of preceded, the "+" or "-". Therefore, PICTURE 99+PPP is valid, and PICTURE 99PPP+ is invalid.

Non-floating insertion symbols "+" and "-", floating insertion symbols "Z", "*", "+", "-", and "cs", and other symbol "P" appear twice in the PICTURE character precedence chart. The leftmost column and uppermost row for each symbol represent its use to the left of the decimal point position. The second appearance of the symbol in the chart represents its use to the right of the decimal point position.

Table 6-4. Order of Precedence

Second Symbol \ First Symbol		Non-Floating Insertion Symbols								Floating Insertion Symbols						Other Symbols					
		B	O	,	.	{+ −}	{+ −}	{CR DB}	CS	{Z *}	{Z *}	{+ −}	{+ −}	CS	CS	9	A X	S	V	P	P
Non-Floating Insertion Symbols	B	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	O	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	,	x	x	x	x	x			x	x	x	x	x	x	x	x			x		x
	.	x	x	x		x			x	x		x		x		x					
	(+ −)																				
	(+ −)	x	x	x	x				x	x	x			x	x	x			x	x	x
	(CR DB)	x	x	x	x				x	x	x			x	x	x			x	x	x
	CS					x															
Floating Insertion Symbols	(Z *)	x	x	x		x			x	x											
	(Z *)	x	x	x	x	x			x	x	x								x		x
	(+ −)	x	x	x					x			x									
	(+ −)	x	x	x	x				x			x	x						x		x
	CS	x	x	x		x								x							
	CS	x	x	x	x	x								x	x				x		x
Other Symbols	9	x	x	x	x	x			x	x		x		x		x	x	x	x		x
	A X	x	x													x	x				
	S																				
	V	x	x	x		x			x	x		x		x		x		x		x	
	P	x	x	x		x			x	x		x		x		x		x		x	
	P					x			x									x	x		x

The following examples illustrate some of the ways a PICTURE clause may be coded:

```
ALPHABETIC ITEMS;
    AA
    A(25)
ALPHANUMERIC ITEMS:
    XX
    X(15)
    A(5)9(4)
    99A99XX
NUMERIC ITEMS:
    9
    99999
    9V99
    S99V99
    999PPP
    J99
EDITED NUMERIC ITEMS (CLASS IS ALPHANUMERIC):
    9.99
    ZZZZZ
    $$.99CR
    B(4)9
    $**,***.99
    -----9          ("-" IS A MINUS SIGN)
    ++,++9.999
    $**,***.99DB
    999,999
    99-99-99        ("-" IS A HYPHEN)
```

Table 6-5 demonstrates the editing function of the PICTURE clause.

Table 6-5. Editing Application of the PICTURE Clause

SOURCE AREA		RECEIVING AREA	
PICTURE	DATA	EDITING PICTURE	EDITED DATA
9(5)	12345	$ZZ,ZZ9.99	$12,345.00
V9(5)	12345	$$$,$$9.99	$0.12
V9(5)	12345	$ZZ,ZZ9.99	$ 0.12
9(5)	00000	$$$,$$9.99	$0.00
9(3)V99	12345	$ZZ,ZZ9.99	$ 123.45
9(5)	00000	$$$,$$$.$$	
9(5)	01234	$**,**9.99	$*1,234.00
9(5)	00000	$**,***.**	*******.**
9(5)	00123	$**,**9.99	$***123.00
9(3)V99	00012	$ZZ,ZZ9.99	$ 0.12
9(3)V99	12345	$$$,$$9.99	$123.45
9(3)V99	00001	$ZZ,ZZZ.99	$ .01
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00000	$ZZ,ZZZ.ZZ	
9(3)V99	00001	$$$,$$$.$$	$.01
S9(5)	(+) 12345	ZZZZ9.99+	12345.00+
S9(5)	(-) 00123	--99999.99	-00123.00
9(3)V99	12345	999.00	123.00
S9(5)	(-) 12345	ZZZZ9.99-	12345.00-
S9(5)	(+) 12345	ZZZZ9.99-	12345.00
9(5)	12345	BBB99.99	45.00
S9(5)V	(-) 12345	-ZZZZ9.99	-12345.00
S9(5)	(-) 12345	$$$$$$.99CR	$12345.00CR
S99V9(3)	(-) 12345	------.99	-12.34
S9(5)	(+) 12345	$$$$$$.99CR	$12345.00
9(3)V99	12345	999.BB	123.
9(5)	12345	00999.00	00345.00
9(7)	0012003	ZZ99JZ9	12 3

REDEFINES

The function of this clause is to allow an area of memory to be referred to by more than one data-name with different formats and sizes.

The construct of the REDEFINES clause is:

```
level-number data-name-1 REDEFINES data-name-2
```

The REDEFINES clause, when specified, must immediately follow data-name-1. The level-numbers of data-name-1 and data-name-2 must be identical and must not be 66 or 88.

This clause must not be used in 01 level entries of the FILE SECTION, as an implicit REDEFINES is assumed when multiple 01 level entries within a file description are present. The size of the record(s) causing implicit redefinition does not have to be equal to that of the record being redefined. The various sizes of implicitly redefined record descriptions create no restriction as to which description is to be coded first, second, third, etc., in the source program. The size of the largest 01 level entry determines the size of the storage area.

Redefinition starts at data-name-2 and ends when a level-number less than or equal to that of data-name-2 is encountered in the source program.

When the level-number of data-name-1 is other than 01 (in the WORKING-STORAGE SECTION), it must specify a storage area of the same size as specified by data-name-2. It is important to observe that the REDEFINES clause specifies the redefinition of a storage area, not simply of the data items occupying that area. Redefined 01 levels do not have to be the same size.

Multiple redefinitions of the same storage area are permitted. The entries giving the new descriptions of the storage area must follow the entries defining the area being redefined, without intervening entries that define new storage areas. Multiple redefinitions of the same storage area may all use the data-name of the originally defined area or the data-name of the area defined just prior to the new area description.

The data description entry being redefined cannot contain an OCCURS clause, nor can it be subordinate to an entry which contains an OCCURS clause.

The entries giving the new description of the storage area must not contain VALUE clauses, except in condition-name entries.

Data-name-2 need not be qualified.

An example of REDEFINES entries follows:

```
01 WORK1.
   03 PART-ONE PC X(60).
   03 PART-TWO REDEFINES PART-ONE.
      05  X   PC X(40).
      05  Y   PC X(20).
   03 PART-THREE REDEFINES PART-TWO PC 9(60).
```

RENAMES

The RENAMES clause permits alternative, and possibly overlapping grouping of elementary items.

The construct of this clause is:

```
66 data-name-1  RENAMES  data-name-2 [{THROUGH} data-name-3]
                                      {THRU   }
```

One or more RENAMES entries can be written for a logical record. All RENAMES entries associated with a given logical record must immediately follow its last record description entry. It is not possible to "chain" RENAMES; i.e., it is illegal to rename data item "A" to "B" and then rename "B" to "C". However, multiple RENAMES of a data-name are permitted. (See figure 6-8.)

Data-name-2 and data-name-3 must be names of elementary items or groups of elementary items of the same logical record and cannot be the same data-name. A 66 level entry cannot rename another 66 level entry nor can it rename a 77, 88, or 01 level entry.

When data-name-3 is specified, data-name-1 is a group item which 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 (if data-name-3 is an elementary item) or the last elementary item in data-name-3 (if data-name-3 is a group item).

When data-name-3 is not specified, data-name-2 can be either a group or an elementary item; when data-name-2 is a group item, data-name-1 is treated as a group item; and when data-name-2 is an elementary item, data-name-1 is treated as an elementary item.

The beginning of the area described by data-name-3 must not be to the left of the beginning of the area described by data-name-2. The end of the area described by data-name-3 must not be to the left of the end of the area described by data-name-2. Data-name-3 cannot be contained within data-name-2. Data-name-2 and data-name-3 may be qualified.

Data-name-1 cannot be used as a qualifier, and can only be qualified by the names of the level 01, SD, or FD entries. Neither data-name-2 nor data-name-3 may have an OCCURS clause in its record description entry or be subordinate to an item that has an OCCURS clause in its record description entry.

When data-name-3 is specified, none of the elementary items within the range, including data-name-2 and data-name-3, can be variable-occurrence items.

Data-name-1 will assume the USAGE of the item being renamed. If the THRU option is used, all items within the RENAMES range must have the same USAGE.

```
01 TAB.
    03 A.
        05 A1 PIC X.
        05 A2 PIC XXX.
        05 A3 PIC XX.
        05 A4 PIC XX.
    03 X.
        05 X1 PIC XX.
        05 X2 PIC X(6).
        05 X3 PIC X(8).
66 B RENAMES A.                  (i.e.,  A1 THRU A4)
66 C RENAMES A.                  (i.e.,  A1 THRU A4)
66 D RENAMES A1 THRU A3.
66 E RENAMES A4 THRU X2
66 F RENAMES A2 THRU X.          (i.e.,  A2 THRU X3)
66 G RENAMES A THROUGH X.        (i.e.,  A1 THRU X3)
```

Figure 6-8. Examples of RENAMES

USAGE

The function of this clause is to specify the format of a data item in computer storage.

The construct of this clause is:

```
[USAGE IS]  { DISPLAY
              CMP
              CMP-1
              CMP-3
              COMP
              COMP-1
              COMP-3
              COMPUTATIONAL
              COMPUTATIONAL-1
              COMPUTATIONAL-3
              INDEX
              ASCII }
```

The USAGE clause can be written at any level. If USAGE is written on group level, it applies to each elementary item in that group.

The USAGE of an elementary item cannot contradict the USAGE of a group to which the item belongs.

COMPUTATIONAL-1 and CMP-1 are acceptable substitutes for, and are equivalent to, COMPUTATIONAL, COMP, or CMP entries.

A warning message of POSSIBLE CMP GROUP USAGE ERROR will appear whenever the receiving field is a group CMP item. This message indicates that the resultant contents during object-program execution of the group CMP item may not contain expected results.

Group moves are performed whenever the sending or receiving field is a group item, and both will be treated as alphanumeric (byte) data. The appropriate conversion takes place when a translation occurs from ASCII to EBCDIC or EBCDIC to ASCII.

USAGE is a declaration for the EBCDIC internal representation of the system and is defined as follows:

a. When USAGE IS DISPLAY, the data item consists of 8-bit (byte) characters.

b. When USAGE IS COMPUTATIONAL, the data item consists of 4-bit coded digits and must be numeric. If a group item is described as computational, the elementary items in the group are computational. The group item itself is not computational (cannot be used in computations).

c. When USAGE IS INDEX, a PICTURE may not be specified. For example, "77 ABC USAGE IS INDEX." An elementary item described with the USAGE IS INDEX clause is called an index data item. An index data item can be referred to directly only in a PERFORM, SEARCH, or SET statement or in a relational condition. A PICTURE may not be specified.

d. When USAGE IS COMPUTATIONAL-3 or CMP-3 it specifies the data item consists of 4-bit coded digits with the low-order digit (LSD) containing the sign. If the data item is unsigned, the LSD position will contain a filler. A COMPUTATIONAL-3 or CMP-3 data item will always end on a byte boundary and its length will be a multiple of a byte adding filler to the left, if necessary.

For example:
```
PC S9999 CMP-3 VALUE +1234 IN MEMORY = ƀ1234C
PC S9999 CMP-3 VALUE -1234 IN MEMORY = ƀ1234D
PC  9999 CMP-3 VALUE +1234 IN MEMORY = ƀ1234ƀ
PC  9999 CMP-3 VALUE -1234 IN MEMORY = ƀ1234ƀ
```
ƀ indicates that one digit of filler has been added.

e. The USAGE IS ASCII clause can only be used for 77 level or 01 level data-names in the WORKING-STORAGE SECTION. A file with recording mode of ASCII will be ASCII USAGE by default.

The PICTURE of a COMPUTATIONAL item can contain only 9's, the operational sign character S, J, the decimal point character V, and one or more P's.

COMPUTATIONAL items may be declared for 9-channel magnetic tape files (TAPE-9), disk file (DISK), Supervisory Printer, paper tape files (PT-READER or PT-PUNCH), or for WORKING-STORAGE SECTION items.

A DISPLAY item is automatically converted to its 4-bit equivalent whenever the receiving area is defined as COMPUTATIONAL, except when the receiving area is a group item. A CMP item is automatically converted to its 8-bit equivalent whenever the receiving area is declared DISPLAY, except when the sending CMP item is a group item.

USAGE

If the USAGE clause is not specified for an elementary item, or for any group to which the item belongs, the usage is assumed to be DISPLAY.

For the most efficient use of hardware storage and internal record storage areas, records should be devised so as to avoid intermixing of odd-length COMPUTATIONAL items with DISPLAY items. This rule is due to the compiler automatically placing the machine addresses of DISPLAY areas to a character boundary.

When the USAGE IS ASCII is used, it specifies that the data item consists of ASCII coded data. A DISPLAY or COMPUTATIONAL item will be automatically converted to its ASCII equivalent whenever the receiving area is defined as ASCII. An ASCII item will be automatically converted to its numeric or EBCDIC equivalent when the receiving field is COMPUTATIONAL or DISPLAY.

VALUE

The function of this clause is to declare an initial value to WORKING-STORAGE items, or the value associated with a condition-name.

The construct of this clause is:

```
{VA    }     IS literal-1      [{THRU    }     literal-2 ]
{VALUE }                       {THROUGH }
 [ literal-3     [{THRU    }     literal-4] ]  ...
                 {THROUGH }
```

The VALUE clause cannot be stated for any item whose size, explicitly or implicitly, is variable.

Abbreviation VA can be used in lieu of VALUE.

Literals may consist of Figurative Constants; e.g., ZEROS, QUOTES, etc.

Literals may be replaced by the reserved word DATE-COMPILED. If DATE-COMPILED is used in the VALUE clause, the date that the program was compiled will be placed in the data-name in the JULIAN form of YYDDD.

In the FILE SECTION, the VALUE clause is allowed only in condition-name (88 level) entries. VALUE entries in other data descriptions in the FILE SECTION are considered as being for documentation purposes only.

The entire VALUE clause may be used with condition-name entries. All levels other than 88 are restricted to the use of literal-1 only.

The VALUE clause must not be stated in a Record Description entry with an OCCURS clause, or in an entry which is subordinate to an entry containing an OCCURS clause. This rule does not apply to condition-name entries.

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

a. If a category of an item is numeric, all literals in the VALUE clause must be numeric literals; e.g., VA 1, 3 THRU 9, 12, 16 THRU 20, 25 THRU 50, 51, 56.

b. If the category of the item is alphabetic, all literals in the VALUE clause must be specifically stated as non-numeric literals; e.g., VA IS "A", "B", "C", "F", "M", "N", "O", "P", "Q", "Z".

c. All literals in a VALUE clause of an item must have a value which requires no editing to place that value in the item as indicated by the PICTURE clause.

d. The function of any editing clause or editing characters in a PICTURE clause is ignored in determining the initial appearance of the item described. However, editing characters are included in determining the length of the item.

In a condition-name entry, the VALUE clause is required and is the only clause permitted in the entry. The characteristics of a condition-name are explicitly those of its conditional variable.

Whenever the THRU phase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.

If this clause is used in an entry at the group level, the literal must be a figurative constant or a non-numeric literal (byte characters). The group area is initialized without consideration for the USAGE of the individual elementary items. Subordinate levels within the group cannot contain VALUE clauses.

The VALUE clause must not be specified for a group containing items that require separate handling due to the USAGE clause.

In a VALUE clause, there is no practical limit to the number of literals in a series. VALUE cannot be associated with an index-data-name.

All numeric literals in a VALUE clause of an item must have a value which is within the range of values indicated by the PICTURE clause, and must not have a value which would require truncation of non-zero digits. Non-numeric items in a VALUE clause of an item must not exceed the size indicated by the PICTURE clause.

WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION is optional and is that part of the DATA DIVISION set aside for intermediate processing of data. The difference between WORKING-STORAGE and the FILE SECTION is that the former deals with data that is not associated with an input or output file. All clauses which are used in normal input or output record descriptions can be used in a WORKING-STORAGE record description.

Organization

Whereas the FILE SECTION is composed of file description (FD or SD) entries and their associated record description entries, the WORKING-STORAGE SECTION is composed only of record description entries and non-contiguous items. The WORKING-STORAGE SECTION begins with a section-header and a period, followed by item description entries for non-contiguous WORKING-STORAGE items, and then by record description entries for WORKING-STORAGE records, in that order. The format for WORKING-STORAGE SECTION is as follows:

```
WORKING-STORAGE SECTION.
77 data-name-1
    88 condition-name-1
          .
          .
77 data-name-n
01 data-name-2
    02 data-name-3
          .
          .
          .
    66 data-name-m RENAMES data-name-3
01 data-name-4
    02 data-name-5
        03 data-name-n
           88 condition-name-2
```

Non-Contiguous WORKING-STORAGE

Items in WORKING-STORAGE which bear no relationship to one another need not be grouped into records, provided they do not need to be further subdivided. Instead, they are classified and defined as non-contiguous items. Each of these items is defined in a separate record description entry which begins with the special level-number 77. The following record description clauses are required in each entry:

a. Level-number.
b. Data-name.
c. PICTURE clause.

The OCCURS clause is not meaningful on a 77 level item and will cause an error at compilation time if used. Other record description clauses are optional and can be used to complete the description of the item if necessary.

All level 77 items must appear before any 01 levels in WORKING-STORAGE.

WORKING-STORAGE Records

Data elements in WORKING-STORAGE which bear a definite relationship to one another must be grouped into records according to the rules for the formation of record descriptions. All clauses which are used in normal input or output record descriptions can be used in a WORKING-STORAGE record description, including REDEFINES, OCCURS, and COPY. Each WORKING-STORAGE record-name (01 level) must be unique since it cannot be qualified by a file-name. Subordinate data-names need not be unique if they can be made unique by qualification.

Initial Values

The initial value of any item in the WORKING-STORAGE SECTION is specified by using the VALUE clause of the record description. If VALUE is not specified, the initial values are set to 4-bit zeros (COMPUTATIONAL). The initial value of any index data item is unpredictable.

Condition-Names

Any WORKING-STORAGE item may be a conditional variable with which one or more condition-names are associated. Entries defining condition-names must immediately follow the conditional variable entry. Both the conditional variable entry and the associated condition-name entries may contain VALUE clauses.

Coding the WORKING-STORAGE SECTION

Figure 6-9 illustrates the coding of the WORKING-STORAGE SECTION.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM: WORKING-STORAGE SECTION CODING | COBOL DIVISION: | PAGE: 1 OF 1

PROGRAMMER: KIMBERLY | DATE: | IDENT 73 80

```
WORKING-STORAGE SECTION.
77  DISK-CONTROL PICTURE 9(8) COMPUTATIONAL.
77  TOTAL-SALES PIC 9(11) VALUE ZERO.
77  SALES-QUOTA PIC 9(10).

01  STATE-TABLE.
    05  STATES.
        10  CALIF  PIC 9999.
        10  NEVADA PIC 9(4).
        10  ORE    PIC 9(4).
    05 STATE-KEY REDEFINES STATES OCCURS 3.
        25  STATE-CODE PIC 99.
        25  COUNTY     PIC 9.
        25  CITY       PIC 9.
01  HDG-LINE.
    03  FILLER PIC A(52) VALUE SPACES.
    03  FILLER PIC A(17) VA "SALES PERFORMANCE".
    03  FILLER PIC X(51) VA SPACES.
```

Figure 6-9. WORKING-STORAGE SECTION Coding

SECTION 7

PROCEDURE DIVISION

GENERAL

The fourth part of the COBOL source program is the PROCEDURE DIVISION. This division contains the procedures needed to solve a given problem. These procedures are written as sentences which may be combined to form paragraphs, which in turn may be combined to form sections. The purpose of the following discussion is to explain this division and its elements.

RULES OF PROCEDURE FORMATION

A procedure is composed of a paragraph, a group of successive paragraphs, a section, or a group of successive sections within the PROCEDURE DIVISION. If declaratives are specified, then sections must be used in the remainder of the PROCEDURE DIVISION. A procedure-name is either a paragraph-name or a section-name.

The end of the PROCEDURE DIVISION (the physical end of the program) is that physical position in a COBOL source program after which no further procedures appear.

A section consists of a section header followed by one or more successive paragraphs. A section ends immediately before the next section-name, at the end of the PROCEDURE DIVISION, or in the Declaratives portion of the PROCEDURE DIVISION at the key words END DECLARATIVES.

A paragraph consists of a paragraph-name followed by one or more successive sentences. A paragraph ends immediately before the next paragraph-name or section-name or at the end of the PROCEDURE DIVISION.

A sentence consists of one or more statements and is terminated by a period followed by a space.

A statement is a syntactically valid combination of words and symbols beginning with a COBOL verb.

The term "identifier" is defined as the word or words necessary to make unique reference to a data item.

EXECUTION OF PROCEDURE DIVISION

Execution begins with the first statement of the PROCEDURE DIVISION, excluding declaratives. Statements are then executed in the order in which they are presented for compilation, except where the rules in this section indicate some other order.

The body of the PROCEDURE DIVISION must conform to the following format:

```
PROCEDURE DIVISION.
[DECLARATIVES.
 section-name SECTION.   declarative-statement.
 paragraph-name.  [statement.] ...
[paragraph-name.  [statement.] ...] ...
 section-name SECTION.  declarative-statement.
 paragraph-name.  [statement.]  ...
[paragraph-name.  [statement.] ...]  ... ]  ...
 END DECLARATIVES.]
[[ section-name SECTION  [priority-number]  . ]
   paragraph-name.  [statement.] ...
 [[paragraph-name.]  ...  [statement.]  ... ] ... ] ...
 [END-OF-JOB.]
```

STATEMENTS

There are three types of statements: imperative statements, conditional statements, and compiler-directing statements.

Imperative Statements

An imperative statement is any statement that is neither a conditional statement nor a compiler-directing statement. An imperative statement may consist of a sequence of imperative statements, each possibly separated from the next by a separator. A single imperative statement is made up of a verb followed by its operand. A sequence of imperative statements may contain either a GO TO statement or a STOP RUN statement which, if present, must appear as the last imperative statement of the sequence. Some of the imperative statements are:

ACCEPT	DISPLAY	MOVE	SEEK
ADD(1)	DIVIDE(1)	MULTIPLY(1)	SET
ALTER	EXAMINE	OPEN	SORT
CLOSE	EXIT	PERFORM	STOP
COMPUTE(1)	GO	READ(3)	SUBTRACT(1)
			WRITE(2) (4)

Conditional Statements

A conditional statement specifies that a truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value. A conditional statement is (1) an IF or SEARCH statement, (2) a READ or RETURN statement that specifies the AT END phrase, (3) a READ or WRITE statement that specifies the INVALID KEY phrase, (4) a WRITE statement that specifies the END-OF-PAGE phrase or (5) the arithmetic statements ADD, SUBTRACT, COMPUTE, DIVIDE, or MULTIPLY that specify the optional phrase ON SIZE ERROR. For example, the IF statement syntax is as follows:

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

Statement-1 or statement-2 can be either imperative or conditional statements. If conditional, the statement can, in turn, contain conditional statements to a depth of 15. Also, if statement-1 or statement-2 is conditional, then the conditions within the conditional statement are considered to be "nested".

Compiler—Directing Statements

A compiler-directing statement is one that consists of a compiler-directing verb (COPY and NOTE) and its operand(s).

1 Without the SIZE ERROR Option.
2 Without the INVALID KEY Option.
3 Without the AT END Option.
4 Without the EOP Option.

SENTENCES

There are three types of sentences: imperative sentences, conditional sentences, and compiler-directing sentences. A sentence consists of a sequence of one or more statements, the last of which is terminated by a period.

Imperative Sentences

An imperative sentence is one or more imperative statements terminated by a period. An imperative sentence can contain either a GO TO statement or a STOP RUN statement which, if present, must be the last statement in the sentence. The following are examples of an imperative sentence.

```
ADD MONTHLY-SALES TO TOTAL-SALES, THEN GO TO PRINT-TOTAL.
```

or

```
DISPLAY "PGM-END" THEN STOP RUN.
```

Conditional Sentences

A conditional sentence is a conditional statement which may optionally contain an imperative statement and must always be terminated by a period.

Examples:

```
IF HEIGHT IS GREATER THAN SIX-FEET-NINE GO TO
TALL-MEN, ELSE ADD 1 TO SOME-OTHER, GO GET-ANOTHER-
RECORD.

IF SALES IS EQUAL TO BOSSES-QUOTA THEN MOVE SALESMAN
TO HONOR-ROLL OTHERWISE MOVE SALESMAN TO QUOTA-
LIST.
```

Compiler-Directing Sentences

A compiler-directing sentence is a single compiler-directing statement terminated by a period.

Example:

```
SCAN. COPY SCANER.
```

SENTENCE PUNCTUATION

The following rules apply to the punctuation of sentences:

a. A sentence is terminated by a period followed by a space.

b. A separator is a word or character used for the purpose of enhancing readability. The use of a separator (other than a space) is optional.

c. The allowable separators are spaces, the semicolon (;), the comma (,), and the reserved word THEN.

d. Separators may be used in the following places:

   1. Between statements.

   2. In a conditional statement.

      (a) Between the condition and statement-1.

      (b) Between statement-1 and ELSE.

e. A separator(other than a space) should be followed by at least one space but is not required.

EXECUTION OF IMPERATIVE SENTENCES

An imperative sentence is executed in its entirety and control is passed to the next applicable procedural sentence.

EXECUTION OF CONDITIONAL SENTENCES

In the conditional sentence:

IF condition statement-1 { OTHERWISE / ELSE } statement-2.

the condition is an expression which is TRUE or FALSE. If the condition is TRUE, then statement-1 is executed and control is then implicitly transferred to the next sentence unless statement-1 causes some other transfer of control. If the condition is FALSE, statement-2 is executed and control passes to the next sentence unless statement-2 causes some other transfer of control.

If statement-1 is conditional, then the conditional statement must be the last (or only) statement comprising statement-1. For example, the conditional sentence would then have the form:

IF condition-1 imperative-statement-1 IF condition-2

statement-3 { OTHERWISE / ELSE } statement-4 { OTHERWISE / ELSE } statement-2.

If condition-1 is TRUE, imperative-statement-1 is executed. If condition-2 is TRUE, statement-3 is executed and control is transferred to the next sentence. If condition-2 is FALSE, statement-4 is executed and control is transferred to the next sentence. If condition-1 is FALSE, statement-2 is executed and control is transferred to the next sentence. Statement-3 can in turn be either imperative or conditional and, if conditional, can in turn contain conditional statements to an arbitrary depth. In an identical manner, statement-4 can either be imperative or conditional, as can statement-2. The execution of the phrase NEXT SENTENCE causes a transfer of control to the next sentence written in order, except when it appears in the last sentence of a procedure being PERFORMed, in which case control is passed to the return control.

EXECUTION OF COMPILER-DIRECTING SENTENCES

The compiler-directing sentences direct activities during compilation time. On the other hand, procedural sentences denote action to be taken by the object program. Compiler-directing sentences may result in the inclusion of routines into the source program. They do not directly result in either the transfer or passing of control. The routines themselves, which the compiler-directing sentences may have included in the source program, are subject to the same rules for transfer or passing of control as if those routines had been created from procedural sentences only.

CONTROL RELATIONSHIP BETWEEN PROCEDURES

In COBOL, imperative and conditional sentences describe the procedure that is to be accomplished. The sentences are written successively, according to the rules of the coding form (section 3), to establish the sequence in which the object program is to execute the procedure. In the PROCEDURE DIVISION, names are used so that one procedure can reference another by naming the procedure to be referenced. In this way, the sequence in which the object program is to be executed may be varied simply by transferring control to a named procedure.

In procedure execution, control is transferred only to the beginning of a paragraph or section. Control is passed to a sentence within a paragraph only from the sentence written immediately preceding it. If a procedure is named, control can be passed to it from any sentence which contains a GO TO or PERFORM, followed by the name of the procedure to which control is to be transferred.

PARAGRAPHS

So that the source programmer may group several sentences to convey one idea (procedure), paragraphs have been included in COBOL. In writing procedures in accordance with the rules of the PROCEDURE DIVISION and the requirements of the coding form (section 3), the source programmer begins a paragraph with a name. The name consists of a word followed by a period, and the name precedes the paragraph it names. A paragraph is terminated by the next paragraph-name. The smallest grouping of the PROCEDURE DIVISION which is named is a paragraph. The last paragraph in the PROCEDURE DIVISION is the optional special paragraph-name END-OF-JOB, which will be the last card in the source program the compiler will use to generate code for the object program.

Programs may contain identical paragraph-names, provided they are resident in different sections. If such paragraph-names are not qualified when used, the current section is assumed. Paragraph-names may be used in GO, PERFORM, and ALTER statements.

SECTIONS

A section consists of one or more successive paragraphs and must be named when designated. The section-name is followed by the word SECTION, a priority number which is optional, and a period. If the section is a DECLARATIVE section, then the DECLARATIVE sentence (i.e., USE or COPY) follows the section header and begins on the same line. Under all other circumstances, a sentence may not begin on the same line as a section-name. The section-name applies to all

paragraphs following it until another section-name is found. It is not required that a program be broken into sections, but this technique is exceptionally useful in trimming down the physical size of object programs by stating a priority number to declare overlayable program storage (see SEGMENT CLASSIFICATION).

Since paragraph-names and section-names both have the same designated position on the reference format (i.e., position A), section-names, when specified, are written on one line followed by a paragraph name on a subsequent line. When PERFORM is used in a non-DECLARATIVE procedural section to call another section, the same rules apply as when PERFORM is used in a DECLARATIVE section.

SEGMENTATION

COBOL segmentation is a facility that provides a means to specify object program overlay requirements. COBOL segmentation deals only with segmentation of procedures. As such, only the PROCEDURE DIVISION and the ENVIRONMENT DIVISION are considered in determining segmentation requirements for an object program.

PROGRAM SEGMENTS

Although it is not mandatory, the PROCEDURE DIVISION for a source program may be written as a consecutive group of sections, each of which are operations that are designed to collectively perform a particular function. Each section must be classified as belonging either to the fixed portion or to one of the independent segments of the object program. Segmentation in no way affects the need for qualification of procedure-names to ensure uniqueness.

The object program is composed of two types of segments: a fixed segment and overlayable segments.

a. The fixed segment is the main program segment and may be overlaid in the same manner as if it were an overlayable segment.

b. An overlayable segment is a segment which, although logically treated as if it were always in memory, can be overlaid, if necessary, to optimize memory utilization. However, such a segment, if called for by the program, is always made available in its "initial" state when the segment priority-number is 50 or greater. When the segment priority-number is 49 or less, the segment will be made available in its last-used state.

In addition, depending on availability of memory, the number of permanent segments in the fixed and overlayable portions can be varied by changing the SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph.

Segment Classification

Sections which are to be segmented are classified by means of a system of priority numbers and the following criteria:

a. Logic requirements: sections with priority numbers from 00 through 49 in a program may reside in the fixed segment, depending on the value specified in SEGMENT-LIMIT. Sections containing a priority number lower than that specified in SEGMENT-LIMIT, regardless of their physical location in the program, will be assigned to the fixed

segment; all other sections will be assigned as overlayable segments. "Fall-through" control from one SECTION to another SECTION is accomplished in their order of appearance in the source program.

b. Relationship to other sections: sections coded within the SEGMENT-LIMIT range will become the fixed segment and can communicate freely with each other. Those coded outside the stated SEGMENT-LIMIT range fall into the overlayable category and can also communicate from one to the other.

The compiler will create one program segment which will include all sections with priority numbers below the value specified in SEGMENT-LIMIT. The overlayable sections will be called into memory as needed by the program. When memory is available, more than one overlayable section will be in memory at the same time. This will reduce the number of disk accesses, which in turn will cause the program to have a shorter run time.

Priority Numbers

Section overlay classifications are accomplished by means of a system of priority numbers. The priority number is included in the section header. The general construct of a section header is as follows:

section-name SECTION priority-number.

The priority number must be an integer ranging in value from 00 through 99 (also 0, 1, 2, etc., are permissible priority numbers). If the priority number is omitted from the section header, the priority number is assumed to be 0. Segments with priority numbers ranging from 0 up to, but not including, the value specified in the SEGMENT-LIMIT clause (or 50 if no SEGMENT-LIMIT clause has been specified) are considered as being located in the fixed portion of the object program. Segments with priority number equal to or higher than the value specified in SEGMENT-LIMIT (but not exceeding 99) are independent segments and fully ALTERable; however, segments with priority numbers greater than 49 will be made available in their "initial" state each time they are referenced. A GO TO paragraph 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. Sections in DECLARATIVES are assumed to be 00 and must not contain priority numbers in their section headers. Priority numbers may be stated in any sequence and need not be in direct sequence. The fixed segment does not end when the first priority number equal to or greater than SEGMENT-LIMIT is encountered.

All segments, regardless of their physical location in the source program, whose priority number is less than that which is specified in SEGMENT-LIMIT will be "gathered" into a single segment. All other segments equal to or greater than that which is specified in SEGMENT-LIMIT will be "gathered" into overlayable segments according to equal priority number, regardless of their physical location in the source program.

The use of the "gathering" technique will allow programmers to create tailored segments which will reduce disk access times. For example:

Program A: SEGMENT-LIMIT equals 17.

Non-Gathered

Segment	Description	Size in Digits
00-16	Main body of the program	20,000
17	Used frequently	1,000
18	Used frequently	5,000
19	Used infrequently	4,000
20	Used at EOJ only	500
21	Used frequently	2,000
22	Used at BOJ only	1,000
23	Used frequently	500
24	Used for infrequent test	1,500
25	Used infrequently	3,000

Gathered

Segment	Description	Size in Digits
00-16	Main body of the program	20,000
17	Used frequently	1,000
18	Used infrequently	5,000
19	Used infrequently	4,000
20	Used at EOJ	500
17	Used frequently (was segment 21)	2,000
19	Used at BOJ (was segment 22)	1,000
17	Used frequently (was segment 23)	500
20	Used for infrequent test (was segment 24)	1,500
20	Used infrequently (was segment 25)	3,000

Results of Gathering

Segment	Description	Size in Digits
00-16	Main body of the program	20,000
17	Used frequently	3,500
18	Used infrequently	5,000
19	Used infrequently	5,000
20	Used infrequently	5,000

"Fall through" will be performed in the sequence as outlined in the above "Non-Gathered" example, and not as they appear in the "Results of Gathering" example above, therefore preserving the logical integrity of the original program.

The COBOL interpreter will automatically check to see if an overlay being called for by an object program is already present in memory. If it is present, no disk access is required and the program is not interrupted. If it is not present, the COBOL interpreter interrupts the program and will access the disk for the desired overlayable portion of the program. The COBOL interpreter uses overlay segments directly from the program library where the object program was compiled to and is called in as an overlay in its initial generated code each and every time it is required by the operating program. Although the initial code is retrieved each time, the latest addresses of ALTERed exits are still applicable and are in force by the use of an automatic ALTER table for segments with a priority number of 49 or less.

DECLARATIVES

Declaratives are procedures which operate under the control of the input-output system. Declaratives consist of compiler-directing sentences and their associated procedures. Declaratives, if used, must be grouped together at the beginning of the PROCEDURE DIVISION. The group of declaratives must be preceded by the key word DECLARATIVES, and must be followed by the words END DECLARATIVES. Each DECLARATIVE consists of a single section and must conform to the rules for procedure formation. There are two statements that are called declarative statements in the COBOL compiler. These are the USE and the COPY statements. The next source statement following the END DECLARATIVES statement must be a section-name or paragraph-name.

Use Declarative

A USE declarative is used to supplement the standard procedures provided by the input-output system. The USE sentence immediately following the section-name, identifies the condition calling for the execution of the USE procedures. Only the PERFORM statements may reference all or part of a USE section. The USE sentence itself is never executed. Within a USE procedure, there must be no reference to the main body of the PROCEDURE DIVISION. The construct for the USE declarative is as follows:

```
section-name SECTION.  USE.....
paragraph-name.  First procedure-statement...
```

Complete rules for writing the formats for USE are stated under the USE verb.

COPY Statement as a Declarative

A COPY declarative is used to incorporate a DECLARATIVE library routine in the source program, that is, a routine which is a USE declarative. The construct of the COPY declarative is:

```
section-name SECTION.  COPY  library-name
```

Complete rules for writing the format for COPY are stated under the COPY verb.

ARITHMETIC EXPRESSIONS

An arithmetic expression is an algebraic expression which is defined as:

a. An identifier of a numeric elementary item.

b. A numeric literal.

c. Such identifiers and literals separated by arithmetic operators.

d. Two arithmetic expressions separated by an arithmetic operator.

e. An arithmetic expression enclosed in parentheses.

Any arithmetic expression may be preceded by a unary + or -. The permissible combinations of identifiers, literals, and arithmetic operators are given in table 7-1. Those identifiers and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic operation may be performed.

Table 7-1. Combination of Symbols in Arithmetic Expressions

First Symbol	Second Symbol					
	Variable	*/**	+-	(	)	
Variable	-	P	P	-	P	
*/**	P	-	P	P	-	
+-	P	-	-	P	-	
(	P	-	P	P	-	
)	-	P	P	-	P	

NOTE

In the above table, the letter "P" represents a permissible pair of symbols. The character "-" represents an invalid character pair. Variable represents an identifier or literal.

Arithmetic Operators

There are five arithmetic operators that may be used in arithmetic expressions. These operators, listed below, are represented by specific characters which must be preceded by a space and followed by a space.

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Formation and Evaluation Rules

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be used. Expressions within parentheses are evaluated first and, within a nest of parentheses, evaluation proceeds from the least inclusive set to the most inclusive set. When parentheses are not used or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of operations is implied:

Unary + or -
**
* and /
+ and -

The symbols + and -, if used without parenthesizing, may only follow one of the arithmetic operators **, *, /, or appear as the first symbol in a formula. Parentheses have a precedence higher than any of the operators and are used to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution in formulas where it is necessary to have some deviation from the normal precedence. When the sequence of execution is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right. Thus, expressions ordinarily considered to be ambiguous, e.g., A / B * C, A / B / C, and A**B**C are permitted in COBOL. They are interpreted as if they were written (A / B) * C, (A / B) / C, and (A**B) **C, respectively. Without parenthesizing, the following example:

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

would be interpreted as:

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

with the sequence of operations working from the innermost parentheses toward the outside, i.e., first exponentiation, then multiplication and division, and finally addition and subtraction.

ARITHMETIC EXPRESSIONS

The way in which operators, variables, and parentheses may be combined in an arithmetic expression is summarized in table 7-1.

An arithmetic expression may only begin with the symbols (, +, -, or a variable and may only end with a) or a variable. There must be a one-to-one correspondence between left and right parentheses of an arithmetic expression such that each left parenthesis is to the left of its corresponding right parenthesis.

CONDITIONS

A condition causes the object program to select between alternate paths of control, depending upon the truth value of a test. Conditions are used in IF and PERFORM statements. A condition is one of the following:

a. Relation condition.

b. Class condition.

c. Condition-name condition.

d. Sign condition.

e. NOT condition.

f. Condition $\left\{\begin{matrix}\text{AND}\\\text{OR}\end{matrix}\right\}$ condition.

The construction NOT condition is not permitted if the condition itself contains NOT.

Logical Operators

Conditions may be combined by logical operators. The logical operators must be preceded by a space and followed by a space. The meaning of the logical operators is as follows:

Logical Operator	Meaning
OR	Logical Inclusive OR
AND	Logical Conjunction
NOT	Logical Negation

Table 7-2 indicates the relationships between the logical operators and conditions A and B. Table 7-3 indicates the way in which conditions and logical operators may be combined.

Relation Condition

A relation condition causes comparison of two operands, each of which may be a data-name, a literal, or an arithmetic expression (formula). Comparison of two elementary numeric items is permitted, regardless of the individual USAGE clauses. However, for all other comparisons, the operands must have the same USAGE. Group numeric items are defined to be alphanumeric. It is not permissible to compare an index-data-name to a literal or a data-name.

Table 7-2. Relationship of Conditions, Logical Operators, and Truth Values

CONDITION		CONDITION AND VALUES		
A	B	A AND B	A OR B	NOT A
TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE

Table 7-3. Combinations of Conditions and Logical Operators

FIRST SYMBOL \ SECOND SYMBOL	CONDITION	OR	AND	NOT	(	)
CONDITION	-	P	P	-	-	P
OR	P	-	-	P	P	-
AND	P	-	-	P	P	-
NOT	*P	-	-	-	P	-
(	P	-	-	P	P	-
)	-	P	P	-	-	P

NOTE

The letter "P" represents a permitted pair of symbols, and the character "-" represents an invalid character pair.

The general format for a relation condition is as follows:

```
{data-name-1                  }                          {data-name-2                  }
{literal-1                    }  relational-operator     {literal-2                    }
{arithmetic-expression-1      }                          {arithmetic-expression-2      }
```

The first operand, data-name-1, literal-1, or arithmetic expression-1 is called the subject of the condition. The second operand, data-name-2, literal-2, or arithmetic expression-2 is called the object of the condition. The object and the subject may not both be literals.

* Permissible only if the condition itself is not a "NOT condition".

Relational Operators

The relational operators specify the type of comparison to be made in a relation condition. The relational operators must be preceded by a space and followed by a space. Relational operators are:

IS [NOT] GREATER THAN.

IS [NOT] LESS THAN.

IS [NOT] EQUAL TO.

IS [NOT] >.

IS [NOT] <.

IS [NOT] =.

EQUALS.

Comparison of Operands

Non-Numeric. For non-numeric (byte) operands, a comparison will result when determination is made that one operand is less than, equal to, or greater than the other with respect to a specified internal collating sequence of characters. The size of an operand is the total number of characters in the operand. Non-numeric operands may be compared only when their USAGE is the same, implicitly or explicitly. There are two cases to consider:

a. If the operands are of equal size, characters in corresponding character positions of the two operands are compared starting from the high-order end through the low-order end. If all pairs of characters compare equally through the last pair, the operands are considered equal when the low-order end is reached. The first pair of unequal characters to be encountered is compared to determine their respective relationship. The operand that contains the character that is positioned higher in the internal collating sequence is considered to be the greater operand.

b. If the operands are of unequal size, the comparison of characters proceeds from high-order to low-order positions until a pair of unequal characters is encountered, or until one of the operands has no more characters to compare. If the end of the shorter operand is reached and the remaining characters in the longer operand are spaces, the two operands are considered to be equal.

Numeric. For operands that are numeric, a comparison results in the determination that one of them is less than, equal to, or greater than the other with respect to the algebraic value of the operands. The length of the oper-

ands, in terms of number of digits, is not significant. Zero is considered a unique value regardless of the sign. Comparison of these operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparisons.

The signs of signed numeric operands will be compared as to their algebraic value of being plus (highest) or minus (lowest).

Sign Condition

The sign condition determines whether or not the algebraic value of a numeric operand is less than, greater than, or equal to 0. The general construct for a sign condition is as follows:

```
arithmetic-expression  IS  [NOT]  {POSITIVE}
                                  {NEGATIVE}
                                  {ZERO    }
```

An operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero.

Class Condition

The class condition determines whether the operand is numeric; that is, consists entirely of the characters 0, 1, 2, 3, ..., 9, with or without an operational sign, or alphabetic, that is, consists entirely of the characters A, B, C, ..., Z, and space. The general construct for the class condition is as follows:

```
identifier IS [NOT] {NUMERIC   }
                    {ALPHABETIC}
```

The usage of the operand being tested must be described, implicitly or explicitly, as DISPLAY or DISPLAY-1.

The NUMERIC test cannot be used with an item whose record description describes the item as alphabetic. If the record description of the item being tested does not contain an operational sign, the item being tested is determined to be numeric only if the contents are numeric and an operational sign is not present.

The ALPHABETIC test cannot be used with an item whose record description describes the item as numeric. The item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters A thru Z and the space.

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. The general construct for the condition-name condition is as follows:

[NOT] condition-name

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in 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 corresponding to the condition-name equals the value of its associated conditional variable.

Evaluation Rules

The evaluation rules for conditions are analogous to those given for arithmetic expressions, except that the following hierarchy applies:

a. Arithmetic expressions (formulas).

b. All relational operators.

c. NOT.

d. AND.

e. OR.

CONDITIONS

Simple Conditions

Simple conditions, as distinguished from compound conditions, are subdivided into four general families of conditional tests: Relation Tests, Relative Value Tests, Class Tests, and the Conditional Variable Tests. A detailed explanation of each of these can be found under the IF verb discussion.

Compound Conditions

The most common construct of a compound condition is:

```
simple-condition-1  {AND}  simple-condition-2
                    {OR }

[ {AND} ...  {AND}  simple-condition-n ]
  {OR }      {OR }
```

Simple conditions can be combined with logical operators, according to specified rules, to form compound conditions. The logical operators AND, OR, and NOT are shown in table 7-2, where A and B represent simple conditions. Thus, if A is TRUE and B is FALSE, then the expression A AND B is FALSE, while the expression A OR B is TRUE.

The following are illustrations of compound conditions:

a. AGE IS LESS THAN MAX-AGE AND AGE IS GREATER THAN 20.

b. AGE IS GREATER THAN 24 OR MARRIED.

c. STOCK-ON-HAND IS LESS THAN DEMAND OR STK-SUPPLY IS GREATER THAN DEMAND + INVENTORY.

d. A IS EQUAL TO B, AND C IS NOT EQUAL TO D, OR E IS NOT EQUAL TO F, AND G IS POSITIVE, OR H IS LESS THAN I * J.

e. STK-ACCT IS GREATER THAN 72 AND (STK-NUMBER IS LESS THAN 100 OR STK-NUMBER EQUAL TO 76920).

Note that it is not necessary to use the same logical connective throughout. The rules for determining the logical (i.e., truth) value of a compound condition are as follows:

a. If AND is the only logical connective used, then the compound condition is TRUE if, and only if, each of the simple conditions is TRUE.

b. If OR is the only logical connective used, then the compound condition is TRUE if, and only if, one or more of the simple conditions is TRUE.

c. If both logical connectives are used, then the conditions are grouped first according to AND, proceeding from left to right, and then by OR, proceeding from left to right.

Parentheses may be used to indicate grouping as specified in the examples below. Parentheses must always be paired the same as in algebra, i.e., the expressions within the parentheses will be evaluated first. In the event that nested parenthetical expressions are employed, the innermost expressions within parentheses are handled first. Examples of using parentheses to indicate grouping are:

a. To evaluate C1 AND (C2 OR NOT (C3 OR C4)), use the first part of rule c above and successively reduce this by substituting as follows:

Let C5 equal "C3 OR C4", resulting in
C1 AND (C2 OR NOT C5)

Let C6 equal "C2 OR NOT C5", resulting
in C1 AND C6

This can be evaluated by referencing table 7-2.

b. To evaluate C1 OR C2 AND C3, use the second part of rule c and reduce this to C1 OR (C2 AND C3), which can now be reduced as in example a.

c. To evaluate C1 AND C2 OR NOT C3 AND C4, group first by AND from left to right, resulting in:

(C1 AND C2) OR (NOT C3 AND C4)

which can now be evaluated as in example a.

d. To evaluate C1 AND C2 AND C3 OR C4 OR C5 AND C6 AND C7 OR C8, group from the left by AND to produce:

((C1 AND C2) AND C3) OR C4 OR ((C5 AND C6)
AND C7) OR C8

which can now be evaluated as in example a.

e. The following uses a condition-name as part of the statement.

IF CURRENT-MONTH AND DAY = 15 OR 30... would
be treated as:

IF (CURRENT-MONTH AND DAY = 15) OR 30... the
actual test desired is:

IF CURRENT-MONTH AND (DAY = 15 OR 30)...

The required result is that CURRENT-MONTH must be true and DAY must contain either 15 or 30.

Without the parentheses as shown, the conditions are:

1. DAY = 30 or
2. CURRENT-MONTH is true AND DAY = 15.

Abbreviated Compound Conditions

Any relation condition other than the first that appears in a compound conditional statement may be abbreviated as follows:

a. The subject, or the subject and relational operator, may be omitted. In these cases, the effect of the abbreviated relation condition is the same as if the omitted parts had been taken from the nearest preceding complete relation condition within the same condition; that is, the first relation is a condition and must be complete.

b. If, in a consecutive sequence of relation conditions (separated by logical operators) the subjects are identical, the relational operators are identical and the logical connectors are identical, the sequence may be abbreviated as follows:

   1. Abbreviation 1: when identical subjects are omitted in a consecutive sequence of relation conditions. An example of abbreviation 1 would be:

      IF A = B AND = C.

      This is equivalent to IF A = B AND A = C.

   2. Abbreviation 2: when identical subjects and relational operators are omitted in a consecutive sequence of relation conditions. An example of abbreviation 2 is:

      IF A = B AND C.

      This is equivalent to IF A = B AND A = C.

c. As indicated in the previous paragraphs, compound conditions can be abbreviated by having implied subjects, or implied subjects and relational operators, providing the first simple condition is a full relation. The missing term is obtained from the last stated relation

in the sentence. The following examples further illustrate the abbreviated compound conditions:

1. IF A = B OR C is equivalent to IF A = B OR A = C.

2. IF A < B OR = C OR D is equivalent to IF A < B OR A = C OR A = D.

INTERNAL PROGRAM SWITCHES

Every compiled object program contains eight automatically provided programmatic switches. Switches SW1 through SW8 are composed of one unsigned digit in length and are located in memory locations 0 through 7 of data segment 0.

These switches can be referred to in the PROCEDURE DIVISION by the use of the reserved words SW1, SW2...SW8. During execution, each individual switch setting can be changed by a MOVE, ADD, SUBTRACT, etc.. For example:

```
MOVE 0 TO SW1.
ADD 1 TO SW2.
SUBTRACT 1 FROM SW3.
```

Note that SW6 has an effect on the MONITOR DEPENDING....requirement if the statement is present.

The switch memory locations are reserved and operate identically to those of the reserved TALLY locations.

VERBS

The verbs available for use with the COBOL Compiler are categorized below. Although the word IF is not a verb in the English language, it is utilized as such in the COBOL language. Its occurrence is a vital feature in the PROCEDURE DIVISION.

a. Arithmetic:
   - ADD
   - COMPUTE
   - DIVIDE
   - MULTIPLY
   - SUBTRACT

b. Compiler Directing:
   - COPY
   - MONITOR
   - NOTE
   - USE

c. Data Manipulations:
   - EXAMINE
   - FORMAT
   - MICR-EDIT
   - MOVE

d. Ending:
   - STOP

e. Input-Output:
   - ACCEPT
   - CLOSE
   - CONTROL
   - DISPLAY
   - OPEN
   - READ
   - SEEK
   - WRITE

f. Logical Control:
   - IF

g. Procedure Branching:
   - ALTER
   - EXIT
   - GO
   - PERFORM
   - ZIP

h. Sort:
   - RELEASE
   - RETURN
   - SORT

i. Table Manipulation:
   - SEARCH
   - SET

j. Debugging:
   - DUMP
   - TRACE

Specific Verb Formats

The specific verb formats, together with a detailed discussion of the restrictions and limitations associated with each, appear on the following pages in alphabetic sequence.

ACCEPT

The function of this verb is to permit the entry of low-volume data from the console typewriter.

The construct of this verb is:

$$\underline{\text{ACCEPT}} \quad \text{identifier} \quad \left[\underline{\text{FROM}} \quad \left\{ \begin{matrix} \underline{\text{SPO}} \\ \text{mnemonic-name} \end{matrix} \right\} \right]$$

This statement causes the operating object program to halt and wait for appropriate data to be entered on the console printer (SPO). The SPO entry will replace the contents of memory specified by the identifier. The systems operator answers an ACCEPT halt by keying in the following message:

mix-index AXdata-required

If a blank appears between the AX and data-required, the blank character will be included in the data-stream.

The MCP will space fill to the right if the number of characters entered is less, or truncate to right if the number of characters entered is more.

If mnemonic-name is used, it must appear in the SPECIAL-NAMES paragraph and be equated to the hardware-name SPO.

The receiving identifier may be a group level entry and cannot be subscripted.

The maximum number of characters per ACCEPT statement is unlimited.

ACCEPT responses of greater than 60 characters must be entered through the SPO in exact groups of 60 characters, except for the last group, which can be of any size up to 60.

Because of the inefficiency of entering data through the keyborad, this technique of data transmission should be restricted solely to low-volume input data.

NOTE

The "<" is a backspace character and is not passed by the MCP.

ADD

The function of this verb is to add two or more numeric data items and adjust the value of the receiving field(s) accordingly.

The construct of this verb has three options.

Option 1:

```
ADD   {literal-1   }    [ {literal-2   }   ... ]
      {identifier-1}      {identifier-2}
TO  identifier-m  [ROUNDED]  [ identifier-n [ROUNDED]... ]
[ ;ON SIZE ERROR statement-1[;ELSE statement-2] ]
```

Option 2:

```
ADD   {literal-1   }  {literal-2   }  [ {literal-3   }   ... ]
      {identifier-1}  {identifier-2}  {identifier-3}
GIVING  identifier-m  [ROUNDED]  [,identifier-n [ROUNDED]] ...
[;ON SIZE ERROR statement-1    [;ELSE statement-2]]
```

Option 3:

```
ADD   {CORR          }  identifier-1  TO  identifier-2
      {CORRESPONDING }
[ROUNDED]  [;ON SIZE ERROR statement-1  [;ELSE statement-2] ]
```

With Option 1, the value(s) of the operand(s) preceding the word TO will be added together and the sum will be added to the existing value(s) of operand(s) following the word TO. A resummation does not occur if the value of one of the identifiers changes in the process.

For example, the result of the statement

```
ADD A, B, C TO C, D(C), E
```

is equivalent to

```
ADD A, B, C GIVING TEMP
ADD TEMP TO C
ADD TEMP TO D(C)
ADD TEMP TO E
```

where TEMP is an intermediate result item provided by the compiler.

In Option 2, the sum of the operands preceding the word GIVING will be inserted as a replacement value of identifier(s) following the word GIVING.

In Options 1 and 2, the identifiers must refer to elementary numeric items only, except that identifiers appearing only to the right of the word GIVING may refer to elementary numeric-edited items.

An ADD statement must have at least two operands.

The composite of operands, which is that data item resulting from the superimposition of all operands, excluding the data item that follows the word GIVING, aligned on their decimal points, must not contain more than 125 digits or characters.

The internal format of operands referred to in an ADD statement may differ among each other. Any necessary format transformation and decimal point alignment are automatically supplied throughout the calculation.

Each literal must be a numeric literal.

If, after point alignment with the receiving data item, the calculated result extends to the right of the receiving data item (i.e., an identifier whose value is to be set equal to the sum), truncation will occur. Truncation is always in accordance with the size associated with the resultant identifier. When the ROUNDED option is specified, it causes the resultant identifier to have its absolute value increased by 1 whenever the most-significant digit of the truncated portion is greater than or equal to 5.

Whenever the magnitude of the calculated result exceeds the largest magnitude that can be contained in a resultant data-name, a size error condition arises. In the event of a size error condition, one of two possibilities will occur, depending on whether or not the ON SIZE ERROR option has been specified. The testing for the size error condition occurs only when the ON SIZE ERROR option has been specified.

a. In the event that ON SIZE ERROR is not specified and size error conditions arise, the value of the resultant identifier is unpredictable.

b. If the ON SIZE ERROR option has been specified and size error conditions arise, then the value of the resultant identifier will not be altered. After it has been determined that there is a size error condition, the "any imperative-statement" associated with the ON SIZE ERROR option will be executed.

If Option 3 is used multiple operations are performed. The operations are executed by the pairing of identical data-names of numeric elementary items subor-

dinate in hierarchy to identifier-1 and identifier-2. Data-names match if they, and all their possible qualifiers up to, but not including identifier-1 and identifier-2, are the same. All general rules pertaining to the ADD verb apply to each individual ADD operation. For instance, if the size of matched data-names does not correspond, in that the decimal point is out of alignment or the sizes differ, the decimal point alignment or truncation takes place according to the rules previously discussed.

In the process of pairing identical data-names, any data-name with the REDEFINES clause is ignored. Similarly, data-names which are subordinate to the subordinate data-names with the REDEFINES clause are ignored.

NOTE

This restriction does not preclude identifier-1 or identifier-2 from having REDEFINES clauses or from being subordinate to data-names with REDEFINES clauses.

If the CORR or CORRESPONDING option is used, any item in the group referred to which contains an OCCURS clause will be ignored. Any items subordinate to such an item will also be ignored.

In Option 3, if either identifier-1 or identifier-2 is a group item which contains RENAMES entries, the entries are not considered in the matching of names.

In Option 3, identifier-1 and identifier-2 must not have a level number of 66, 77, or 88.

If corresponding data-names are not elementary numeric items, the ADD operation will be ignored.

In Option 3, CORR is an acceptable substitute for CORRESPONDING.

ALTER

The function of this verb is to modify a predetermined sequence of operations by changing the operand of a labeled GO TO paragraph.

The construct of this verb is:

```
ALTER  procedure-name-1  TO  [PROCEED TO] procedure-name-2
 [procedure-name-3  TO[PROCEED TO] procedure-name-4 ...]
```

Procedure-name-1, procedure-name-3, ... are names of paragraphs, each of which contains a single sentence consisting of only a GO TO statement as defined under Option 1 of the GO TO verb. Procedure-name-2, procedure-name-4. ... are not subject to the same restrictions and they may be either paragraph-names or section-names.

When control passes to procedure-name-1, control is immediately passed to procedure-name-2 rather than to the procedure-name referred to by the GO TO statement in procedure-name-1, Procedure-name-1 is therefore a "gate" which remains set until again referenced by another ALTER statement.

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.

All other uses of the ALTER statement are valid and are performed even if the GO TO which the ALTER refers to is in an overlayable section, as long as the section priority number is less than 50.

CLOSE

The function of this verb is to communicate to the MCP that the designated file-name being operated on or created is programmatically completed, and also to fulfill the stated action requirements.

The construct of this verb is:

```
CLOSE  file-name-1  [REEL]   [ WITH  { LOCK          } ]
                               {       PURGE         }
                               {       RELEASE       }
                               {       NO REWIND     }
                               {       REMOVE        }

  [file-name-2...]
```

File-names must not be those defined as being SORT files. A file must have been OPENed previously before a CLOSE statement can be executed for the file. File space in memory will not be allocated until the file has been OPENed. When a file is programmatically CLOSEd and the assigned unit is released, the memory allocated for that file will be returned to the MCP. The MCP I/O assignment table reflects any unit which remains assigned to the program after the file on that unit has been CLOSEd.

The above statement applies to the following categories of input and output files.

a. Files whose input and outpput media involve print files, card files, etc.

b. Files which are contained entirely on one reel of magnetic tape and are the only files on that reel.

c. Files which may be contained on more than one physical reel of magnetic tape. Furthermore, the number of reels might possibly be higher than the number of physical tape units provided on the system.

d. Disk files.

To show the effects of the CLOSE options, each type of file will be discussed separately.

a. Card Input.

   1. CLOSE - does not release the input memory areas or the reader.

   2. CLOSE WITH NO REWIND - same as CLOSE.

3. CLOSE WITH RELEASE - releases the input memory areas and returns the reader to the MCP.

4. CLOSE WITH LOCK - same as CLOSE WITH RELEASE.

5. CLOSE WITH PURGE - same as CLOSE WITH RELEASE.

6. CLOSE WITH REMOVE - same as CLOSE.

b. Card Output.

1. CLOSE - punches the trailer label (if any) and does not release the output memory areas or the punch.

2. CLOSE WITH NO REWIND - same as CLOSE.

3. CLOSE WITH RELEASE - releases the output memory areas and returns the punch to the MCP.

4. CLOSE WITH LOCK - same as CLOSE WITH RELEASE.

5. CLOSE WITH PURGE - same as CLOSE WITH RELEASE.

6. CLOSE WITH REMOVE - same as CLOSE.

c. Magnetic Tape Input.

1. CLOSE - rewinds the tape and does not release the input memory areas. The unit remains assigned to the program.

2. CLOSE WITH NO REWIND - same as CLOSE except the tape is not rewound.

3. CLOSE WITH LOCK - releases the input memory areas, rewinds the tape, and the MCP marks the unit not ready.

4. CLOSE WITH RELEASE - releases the memory input areas, rewinds the tape, and returns the unit to the MCP.

5. CLOSE WITH PURGE - releases the input memory areas, rewinds the tape, and if a write ring is in the reel, over-writes the label, making the tape a scratch tape which becomes a candidate for use by the MCP. The unit is returned to the MCP.

6. CLOSE WITH REMOVE - same as CLOSE.

d. Magnetic Tape Output.

1. CLOSE - does not release the output memory areas, writes the trailer label (if any), and rewinds the tape. The unit remains assigned to the program.
2. CLOSE WITH NO REWIND - does not release the output memory areas, writes the trailer label (if any). The tape remains positioned beyond the trailer label (or tape mark if there is no trailer label). The unit remains assigned to the program.
3. CLOSE WITH LOCK - releases the output memory areas, writes the trailer label (if any), rewinds the tape, and the MCP marks the unit not ready.
4. CLOSE WITH RELEASE - releases the output memory areas, writes the trailer label (if any), rewinds the tape, and returns the unit to the MCP.
5. CLOSE WITH PURGE - releases the output memory areas, writes the trailer label (if any), rewinds the tape, returns the unit to the MCP, and the MCP overwrites the label, making it a scratch tape and a candidate for use by the MCP.
6. CLOSE WITH REMOVE - same as CLOSE.

e. Printer Output.

1. CLOSE - prints the trailer label (if any) and does not release the output memory areas or the printer.
2. CLOSE WITH NO REWIND - same as CLOSE.
3. CLOSE WITH RELEASE - releases the output memory areas and returns the printer to the MCP.
4. CLOSE WITH LOCK - same as CLOSE WITH RELEASE.
5. CLOSE WITH PURGE - same as CLOSE WITH RELEASE.
6. CLOSE WITH REMOVE - same as CLOSE.

f. Disk Files. The actions taken on files ASSIGNED to DISK will be discussed in terms of old files and new files. An old file is one that already exists on disk and appears in the MCP Disk Directory. A new file is one created by the program and does not appear in the Directory. A new file may only be referenced by the program which creates it.

1. CLOSE - does not release the input/output memory areas.

(a) For an old file, the file is left in the Directory and is available to other programs.

(b) For a new file, the file is not entered in the Directory; however, it remains on the disk and may be OPENed again by this program.

2. CLOSE WITH NO REWIND - not permitted on disk files.

3. CLOSE WITH RELEASE - releases the input/output memory areas.

   (a) For an old file, the file is left in the directory and is available to other programs.

   (b) For a new file, the file is entered in the directory and is available to other programs.

4. CLOSE WITH LOCK - releases the input/output memory areas.

   (a) For an old file, the file remains in the Directory and is made available.

   (b) For a new file, the file is entered in the Directory and is available to other programs.

5. CLOSE WITH PURGE - releases the input/output memory areas.

   (a) An old file is immediately removed from the disk and deleted from the Directory.

   (b) A new file will be immediately removed from the disk.

6. CLOSE WITH REMOVE - releases the input/output memory areas. This option will cause the MCP to REMOVE a file from the disk directory that has the same file-id as the file being closed. This action will take place prior to entering the closing files file-id in the disk directory. Use of this option will eliminate the DUPLICATE FILE condition and reduce operator intervention. If the REMOVE option is not used, the "RM" SPO input message will accomplish the same results.

If a file has been specified as being OPTIONAL, the standard end-of-file processing is permitted whenever the file is not present.

If a CLOSE statement without the REEL option has been executed for a file, a READ, WRITE, or SEEK statement for that file must not be executed unless an intervening OPEN statement for that file is executed.

CLOSE

The CLOSE REEL option signifies that the file-name being CLOSEd is a multi-reel magnetic tape input or output file. The reel will be CLOSEd when the CLOSE REEL statement is encountered and an automatic OPEN of the next sequential reel of the multi-reel file is then performed by the MCP.

COMPUTE

The function of this verb is to assign to a data item the value of a numeric data item, literal, or arithmetic expression.

The construct of this verb is:

```
COMPUTE  identifier-1 [ROUNDED] =  { identifier-2          }
                                   { literal               }
                                   { arithmetic expression }

  [;ON SIZE ERROR statement-1  [;ELSE statement-2] ]
```

The literal must be a numeric literal.

Identifier-2 must refer to an elementary numeric item. Identifier-1 may describe an elementary edited item.

The arithmetic expression option permits the use of any meaningful combination of identifiers, numeric literals, arithmetic operators, and parenthesization, as required.

All rules regarding ON SIZE ERROR, ROUNDED options, truncation and editing are the same as for ADD.

If numeric-literal exponents are used, the results are accurate up to 18 digits in length or to as many decimal places.

COPY

The function of this verb is to allow library routines contained on a source language library file to be incorporated into the program.

The construct of this verb contains the following two options:

Option 1:

```
COPY  library-name
```

Option 2:

```
COPY  library-name
[ REPLACING  {word-1      }  BY  {word-2      }
             {data-name-1 }      {identifier-2}
                                 {literal-1   }
    [ {word-3      }  BY  {word-4      }  ... ]  . ]
      {data-name-3 }      {identifier-4}
                          {literal-2   }
```

The COPY statement may refer only to one library entry in the library. Library-name is the value placed in a library entry bounded by quotes or a procedure-name type word. The library entry can contain up to three 10-character non-numeric literals each separated by a slash (/), following normal naming conventions for disk files.

The library file is inserted in the source program immediately after the COPY statement at compilation time. The result is the same as if the library data were actually a part of the source program.

Library data can encompass an entire procedure, which may be any number of statements, paragraphs, or entire source program divisions (or parts thereof).

Library files may not contain COPY statements.

No statement may appear to the right of the COPY statement on the same source card.

COPY during the PROCEDURE or ENVIRONMENT divisions must follow a SECTION or paragraph-name, and all information contained in the library file is included and can be fully referenced.

On a COPY during the DATA DIVISION, the FD file-name, or the level 01 data-name preceding the COPY is saved and the relative constructs from the library file are discarded. For example, the statement

```
FD MASTER-INPUT COPY "MASTER".
```

will cause the library file titled MASTER to be inserted into the source program immediately following the COPY statement. The source program must refer to the FD file-name as MASTER-INPUT, not as MASTER. The library FD file-name will appear on the output listing, but cannot be referenced in the source program.

Library texts copied from the library are flagged on the output listing by an "L" preceding the sequence number.

In Option 2, a word is defined as being any COBOL word that is not a COBOL reserved word. For example, the following statement reflects non-reserved COBOL words AAA,BBB and 1234, where AAA and BBB are data-names and 1234 is a COBOL word:

```
MULTIPLY AAA BY BBB, THEN GO TO 1234.
```

If the COPY REPLACING option is specified, each word-1 or data-name-1 stipulated will be replaced by the word-2 or data-name-2 entries specified in the option. Data-names may not be subscripted, indexed, or qualified.

Use of the COPY REPLACING option requires that the "library-name" COBOL source image file be present on disk, prior to compilation of the source program containing the COPY REPLACING option. The use of this option <u>will not</u> cause alteration of the library file residing on disk.

In Option 2, literals contained in a library file cannot be replaced by literals, words, or data-names. If an integer is used for a word and it is the last entry in a replacing list, it must be followed by a blank and then a period. For example:

```
COPY BERMAN REPLACING AAA BY HOURS,
BBB BY PAY-SCALE, 1234 BY 58b.
```

The COPY REPLACING option is exceptionally useful for conversion of generalized COBOL source-language library routines into specific and well-named routines within a given program. For example, a generalized COBOL source-language library routine may use the following data-names for the purposes shown.

Data-name	Purpose
AAA	Monthly hours worked per employee.
BBB	Employee pay-rate.
CCC	Employee social security number.
DDD	Employee income tax rate.
EEE	Employee year to date gross income.
FFF	Employee year to date net income.
GGG	Employee gross pay for month.
	Employee net pay for the month.
.	.
.	.
.	.
1234	Specifies a GO TO exit from the routine.

A program calling upon the above generalized routine can replace the non-descript data-names with descriptive names as defined in the program's record description or WORKING-STORAGE area. For example:

```
COPY...REPLACING AAA BY HOURS-WORKED
COPY...REPLACING BBB BY RATE-OF-PAY
COPY...REPLACING CCC BY SOC-SEC-NR
COPY...REPLACING DDD BY INC-TAX-RATE
COPY...REPLACING EEE BY YR-TO-DATE-GROSS
COPY...REPLACING FFF BY YR-TO-DATE-NET
COPY...REPLACING GGG BY THIS-MONTHS-GROSS
COPY...REPLACING HHH BY THIS-MONTHS-NET
                        .
                        .
                        .
COPY...REPLACING 1234 BY WRITE-EMPLOYEE-DRAFT.
```

The specified source program data-names and exit points will be inserted into the library file routine at every occurrence of the assigned generalized names within the routine.

Library Creation. A library file will be created only during a COBOL compilation each time a source card is encountered that contains an "L" in column 7 followed by a library-name on that same card. A library-file may contain up to a maximum of 20,000 card images.

Each library file in the source program will be terminated when a card containing an "L" in column 7 followed by all blanks or another library-name is encountered.

Once a file has been created, it may be COPYed by other programs, or by the creating program in succeeding FD, 01, or procedure COPY statements.

The source data used to create an original library file will also be compiled into the object program at the point of appearance.

All assigned library-names must be unique to other library-names contained in the library, to preserve the integrity of the COBOL library system.

Library files to be used with the COPY verb can be created by a user program which creates a card image file on disk. The compiler will automatically accept any blocking the user may desire.

DISPLAY

The function of this verb is to provide for the printing of low-volume data, error messages, and operator instructions on the console typewriter.

The construct of this verb is:

```
DISPLAY    {literal-1   }    [,{literal-2   }    ...]
           {identifier-1}      {identifier-2}

[UPON      {SPO         }]
           {mnemonic-name}
```

Each literal may be any figurative constant except ALL.

All special registers (DATE, TIME, TALLY, SW_1 ... SW_n, etc.) may be DISPLAYed.

The DISPLAY statement causes the contents of each operand to be written on the supervisory printer (SPO), from the MCP SPO queue, to ensure that a program is not operationally deterred while a message is printing.

If a figurative constant is specified as one of the operands, only a single character of the figurative constant is displayed.

The data-names may be subscripted or indexed and can be COMPUTATIONAL or DISPLAY items.

An infinite amount of characters may be displayed with one statement. The compiler will supply automatic carriage returns and line feeds, if appropriate.

The DISPLAY series option will cause the literals or identifiers to be printed on one line and, if required, the compiler will cause automatic carriage returns and line feeds for information extending to other lines of print. The compiler will format each line so that a partial word at an end of a line will not be printed on that line and continued on the following lines.

When mnemonic-name is used, it must appear in the SPECIAL-NAMES paragraph equated to the hardware-name SPO.

DIVIDE

The function of this verb is to divide one numerical data-item into another and set the value of an item equal to the result.

The construct of this verb contains the following two options:

Option 1:

```
DIVIDE  [MOD]  {literal-1   }  INTO  identifier-2  [ROUNDED]
               {identifier-1}
   [;ON SIZE ERROR statement-1 [;ELSE statement-2] ]
```

Option 2:

```
DIVIDE  [MOD]  {literal-1   }  {BY  }  {literal-2   }
               {identifier-1}  {INTO}  {identifier-2}
   GIVING  identifier-3 [ROUNDED]
 [ REMAINDER identifier-4 [ROUNDED] ]
   [;ON SIZE ERROR statement-1 [;ELSE statement-2] ]
```

Identifier-3 and identifier-4 of Option 2 may refer to elementary numeric-edited items.

Each literal must be a numeric literal.

Division by zero is not permissible and, if executed, will result in a size error indication. This can be handled programmatically, either by doing a zero test prior to the division or by the use of the SIZE ERROR clause. If SIZE ERROR is not written, an attempt to divide by zero will result in program termination.

All identifiers must refer to elementary numeric items.

In Option 1, the value of the operand preceding the word INTO will be divided into the operand following INTO and the resulting quotient stored as the new value of the latter.

The use of the BY option will cause literal-1, identifier-1 to be divided by literal-2, identifier-2, whereas the INTO option will cause literal-1, identifier-1 to be divided into literal-2, identifier-2.

In Option 2, the resulting quotient will be stored as the new value of identifier-3. The value of the operands immediately to the left of the word GIVING will remain unchanged.

The ROUNDED option and ON SIZE ERROR clause and truncation are the same as those discussed for the ADD statement.

The size of the operands is determined by the sum of the divisor and the quotient. The sum of the two cannot exceed 99 digits.

The use of the MOD option will cause the remainder to be placed in identifier-2 of Option 1 and identifier-3 of Option 2. The remainder will be carried to the same degree of accuracy as defined in the PICTURE of the quotient, and all extra positions will be filled with zeros.

Literals cannot be used as dividends.

The use of the REMAINDER option will cause the remainder to be placed in identifier-4, and identifier-3 will contain the quotient, unless the MOD option is also included. If the MOD option is included, both identifier-3 and identifier-4 will contain the remainder.

DUMP

The DUMP statement causes messages to be displayed on the line printer instead of the console printer. The syntax is as follows:

```
DUMP [list]
```

where [list] is a list of data-names, literals, and blanks (for spacing).

The DUMP statement must be used in conjunction with the MONITOR declaration because it uses the same WRITE routine.

EXAMINE

The function of this verb is to replace a specified character, and/or to count the number of occurrences of a particular character in a data item.

The construct of the verb contains the following two options:

Option 1:

```
EXAMINE  identifier-1

TALLYING  {ALL       }  {literal-1   }  [REPLACING BY  {literal-2   }]
          {LEADING   }  {identifier-2}                  {identifier-3}
          {UNTIL FIRST}
```

Option 2:

```
EXAMINE  identifier-1  REPLACING  {ALL       }
                                  {LEADING   }
                                  {[UNTIL] FIRST}

   {literal-3   }  BY  {literal-4   }
   {identifier-4}      {identifier-5}
```

The description of identifier-1 must be such that USAGE is DISPLAY explicitly or implicitly.

Each literal used in an EXAMINE statement must consist of a single DISPLAY character. Figurative constants will automatically represent a single DISPLAY character.

Examination proceeds as follows:

a. For items that are not numeric, examination starts at the leftmost character and proceeds to the right. Each 8-bit character in the item specified by the data-name is examined in turn. Any reference to the first character means the left-most character.

b. If an item referenced by the EXAMINE verb is numeric, it must consist of numeric characters and may possess an operational sign. Examination starts at the leftmost character (excluding the sign) and proceeds to the right. Each character except the sign is examined in turn. Regardless of where the sign is physically located, it is completely ignored by the EXAMINE verb. Any reference to the first character means the leftmost numeric character.

The TALLYING option creates an integral count (i.e., a tally) which replaces the value of a special register called TALLY. The count represents the number of:

a. Occurrences of literal-1 or identifier-2 when the ALL option is used.
b. Occurrences of literal-1 or identifier-2 prior to encountering a character other than literal-1 or identifier-2 when the LEADING option is used.
c. Characters not equal to literal-1 or identifier-2 encountered before the first occurrence of literal-1 or identifier-2 when the UNTIL FIRST option is used.

When either of the REPLACING options is used (i.e., with or without TALLYING), the replacement rules are as follows:

a. When the ALL option is used, then literal-2 or identifier-3 or literal-4 or identifier-5 is substituted for each occurrence of literal-1 or identifier-2 or literal-3 or identifier-4.
b. When the LEADING option is used, the substitution of literal-2 or identifier-3 or literal-4 or identifier-5 terminates as soon as a character other than literal-1 or identifier-2 or literal-3 or identifier-4 or the right-hand boundary of the data item is encountered.
c. When the UNTIL FIRST option is used, the substitution of literal-2 or identifier-3 or literal-4 or identifier-5 terminates as soon as literal-1 or identifier-2 or literal-3 or identifier-4 or the right-hand boundary of the data item is encountered.
d. When the FIRST option is used, the first occurrence of literal-3 or identifier-4 is replaced by literal-4 or identifier-5.

The field called TALLY is a 5-digit field provided by the compiler. Its usage is COMPUTATIONAL and will be reset to zero automatically when the EXAMINE... TALLY option is encountered.

EXIT

The function of this verb is to provide a terminating point for a PERFORM loop, whenever required.

The construct of this verb is:

EXIT.

If the EXIT statement is used, it must be preceded by a paragraph-name and appear as a single one-word paragraph. EXIT is documentational only, but if used, must follow the rules of COBOL.

The EXIT is normally used in conjunction with conditional statements contained in procedures referenced by a PERFORM statement. This allows branch paths within the procedures to rejoin at a common return point.

If control reaches an EXIT paragraph and no associated PERFORM or USE statement is active, control passes through the EXIT point to the first sentence of the next paragraph.

GO TO

The function of this verb is to provide a means of interrupting out of the sequential, sentence by sentence, execution of code, and to permit continuation at some other location indicated by the procedure-name(s).

The construct of this verb has the following two options:

Option 1:

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

Option 2:

```
GO TO procedure-name-1 [, procedure-name-2] ... , procedure-name-3
    DEPENDING ON identifier
```

Each procedure-name is the name of a paragraph or section in the PROCEDURE DIVISION of the program.

Whenever a GO TO statement (represented by Option 1) is executed, control is unconditionally transferred to a procedure-name, or to another procedure-name if the GO TO statement has been changed by an ALTER statement.

A GO TO statement is unrestricted as to where it branches to in a segmented program. It can call upon any segment at either the section level or paragraph levels.

In Option 1, when the GO TO is referred to by an ALTER statement, the following rules apply, regardless of whether or not procedure-name is specified:

a. The GO TO statement must be the only statement in the paragraph.

b. If the procedure-name is omitted, and if the GO TO statement is not referenced by an ALTER statement prior to the first execution of the GO TO statement, the MCP will cause the job to be terminated.

If a GO TO statement represented by Option 1 appears in an imperative statement, it must appear as the only or the last statement in a sequence of imperative statements.

In Option 2, GO TO... DEPENDING... may specify up to 1023 procedure-names in a single statement. The data-name in the format following the words DEPENDING ON must be a numeric elementary item described without any positions to the right of the assumed decimal point. Furthermore, the value must be positive in order to pass control to the procedure-names specified. Control will be transferred to procedure-name-1 if the value of the identifier is 1, to procedure-name-2 if the value is 2, etc. If the value of the identifier is anything other than a positive integer, or if its value is zero, or its value is higher than the number of procedure-names specified, control will be passed to the next statement in normal sequence. For example:

```
GO TO MFG, RE-SALE, STOCK, DEPENDING ON S-O.
```

VALUE OF S-O	GO TO PROCEDURE-NAME
-1	next statement
0	next statement
1	MFG
2	RE-SALE
3	STOCK
4	next statement

IF

The IF statement causes a condition to be evaluated. The subsequent action of the object program depends on whether the value of the condition is true or false.

The construct for the IF statement is as follows:

IF condition-1; {statement-1 / NEXT SENTENCE} [; ELSE {statement-2 / NEXT SENTENCE}]

Statement-1 and statement-2 represent either a conditional statement or an imperative statement, and either may be followed by a conditional statement. The semicolons are optional.

The phrase ELSE NEXT SENTENCE may be omitted if it immediately precedes the terminal period of the sentence.

When an IF statement is executed, the following action is taken:

a. If the condition is true, the statements immediately following the condition (represented by statement-1) are executed, and control then passes implicitly to the next sentence unless statement-1 causes some other transfer of control.
b. If the condition is false, either the statements following ELSE are executed or, if the ELSE clause is omitted, the next sentence is executed.

When an IF statement is executed and the NEXT SENTENCE phrase is present, control passes explicitly to the next sentence, depending on the truth value of the condition and the placement of the NEXT SENTENCE phrase in the statement.

IF statements within IF statements may be considered as paired IF and ELSE combinations, proceeding from left to right; thus, any ELSE encountered is considered to apply to the immediately preceding IF that has not already been paired with an ELSE.

When control is transferred to the next sentence, either implicitly or explicitly, control passes to the next sentence as written or to a return mechanism of a PERFORM or a USE statement.

The method of evaluating conditional expressions allows early exit, once the truth value of the expression has been determined. If the expression contains procedure calls on user intrinsics or makes use of implied subjects, the expression is evaluated fully.

MOVE

The MOVE statement transfers data, in accordance with the rules of editing, to one or more data areas.

The construct for the MOVE statement consists of the following two options:

Option 1:

```
MOVE  {identifier-1}  TO  identifier-2  [, identifier-3] ...
      {literal     }
```

Option 2:

```
MOVE  {CORRESPONDING}  identifier-1  TO  identifier-2
      {CORR         }
```

Identifier-1 and literal represent the sending field; identifier-2, identifier-3 represent the receiving fields. Literal may be any literal or figurative constant consistent with the class of the receiving field.

Option 1 provides for multiple receiving fields. The data designated by the literal or identifier-1 will be moved first to identifier-2, then to identifier-3, etc. Subscripting or indexing associated with identifier-1 is evaluated only once, immediately before data is moved to the first receiving field. The notes referencing identifier-2 also apply to the other areas.

The result of the statement:

```
MOVE A(SUB) TO SUB, B(SUB)
```

would produce the same result as:

```
MOVE A(SUB) TO TEMP.
MOVE TEMP TO SUB.
MOVE TEMP TO B(SUB).
```

Elementary Moves. Any more in which the sending and receiving items are both elementary items is an elementary move. All other moves are defined as group moves. Every elementary item belongs to one of these five categories:

a. Numeric.
b. Numeric Edited.
c. Alphabetic.
d. Alphanumeric.
e. Alphanumeric Edited.

See the PICTURE clause description in section 6 for a detailed discussion of these categories. Group items, non-numeric literals, and all figurative constants, except ZEROS and SPACES, are classed as alphanumeric. Numeric literals and the figurative constant ZEROS are classed as numeric. The figurative constant SPACES is classed as alphabetic.

Illegal Elementary Moves. The rules governing illegal elementary moves are as follows:

1. A numeric-edited item, alphanumeric edited item, SPACES, or an alphabetic item cannot be moved to a numeric or numeric edited item.
2. A numeric literal, ZEROS, a numeric data item, or a numeric edited item cannot be moved to an alphabetic data item.
3. A non-integer numeric literal or a non-integer numeric data item cannot be moved to an alphanumeric or alphanumeric edited data item.

Legal Elementary Moves. The rules governing legal elementary moves are as follows:

4. When an alphanumeric or alphanumeric edited item is a receiving field, justification and any necessary space filling takes place as defined under the JUSTIFIED clause. If the size of the sending field is greater than the size of the receiving field, the excess characters are truncated on the right after the receiving item is filled.

   If the sending field is described as being signed numeric, the operational sign will not be moved. If the sign occupies a separate character position (KSIGN), that character will not be moved and the size of the sending field will be considered to be one less than its actual size.

   For example:

   Given these data descriptions:

   ```
   77 S PIC K9999.
   77 R PIC X(6).
   ```

   Then the statements:

   ```
   MOVE -124 TO S.
   MOVE S TO R.
   ```

   will result in R being equal to "0124 "

5. When a numeric or numeric edited item is the receiving field in an elementary move, data is moved algebraically (that is, values are moved, characters are not moved). Therefore, if the data in the sending field is not numeric, zone bits will be stripped and the data will be modified. Alignment by decimal point and any necessary zero-filling takes place as defined under the JUSTIFIED clause, except where zeros are replaced because of editing requirements.

   When a signed numeric item is the receiving field, the sign of the sending field is placed in the receiving field. Conversion of the sign representation takes place as necessary. If the sending field is unsigned, a positive sign is generated for the receiving field.

   When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.

   When an alphanumeric item is the sending field, data is moved as if the sending item was described as an unsigned numeric integer.

6. When the receiving field is alphabetic, justification and any necessary space filling takes place as defined under the JUSTIFIED clause. If the size of the sending field is greater than the size of the receiving field, the excess characters are truncated on the right, after the receiving field is filled.

Group Moves. A group move is any move in which either the sending field or the receiving field is a group item. Group moves are handled as alphanumeric to alphanumeric moves, regardless of the class of the receiving field and without consideration for the individual elementary or group items contained within either the sending or receiving area.

Translation. Any necessary translation of data from one form of internal representation to another, i.e., ASCII to EBCDIC, EBCDIC to hexadecimal, etc., will be done for any elementary or group move in which data is moved non-algebraically. The type of translation depends on the usages of the sending and receiving data items. Data items declared within the sending or receiving fields are not considered.

For example, moving an elementary numeric item of type integer to an alphanumeric item causes the absolute value of the elementary item to be converted to characters of the same size as their destination. Then they are placed in their destination, left-justified, with spaces in any character positions to the right.

INDEX DATA ITEMS

An index data item cannot be used as an operand in a MOVE statement. The SET statement must be used to move index data items.

VALID MOVE COMBINATIONS

Figure 7-1 shows the valid combinations of sending and receiving fields permitted in COBOL.

When Option 2 is used, selected items within identifier-1 are moved, with any required editing, to selected areas within identifier-2. Identifier-1 and identifier-2 must be group items. Items are selected by matching the data-names of items defined within identifier-1 with like data-names of areas defined within identifier-2, according to the rules specified in the discussion of the CORRESPONDING option. The resulting operation on each of the sets of matched data items proceeds as if an Option 1 MOVE had been specified.

RECEIVING / SENDING		ALPHABETIC	AN		AE	DISPLAY NUMERIC		CMP NUMERIC		NE
			GROUP	ELEM		INTEGER	REAL	INTEGER	REAL	
ALPHABETIC		(1)	(1)	(1)	(6)	*	*	*	*	*
AN	GROUP	(1)	(1)	(1)	(1)	(1)	(1)	(3)	(3)	(1)
	ELEM	(1)	(1)	(1)	(6)	(2)	(2)	(2)	(2)	(7)
AE		(1)	(1)	(1)	(6)	*	*	*	*	*
DISPLAY NUMERIC (DN OR LIT)	INTEGER	*	(1)	(1)	(6)	(2)	(2)	(2)	(2)	(7)
	REAL	*	(1)	*	*	(2)	(2)	(2)	(2)	(7)
CMP NUMERIC	INTEGER	*	(5)	(5)	(4)	(2)	(2)	(2)	(2)	(7)
	REAL	*	(1)	*	*	(2)	(2)	(2)	(2)	(7)
NE		*	(1)	(1)	(6)	*	*	*	*	*
HEX LIT (TREATED AS INTEGER LIT)		*	(1)	(1)	(6)	(2)	(2)	(2)	(2)	(7)
NON-NUM LIT		(1)	(1)	(1)	(6)	(2)	(2)	(2)	(2)	(7)

	NON-NUMERIC MOVE	NUMERIC MOVE	LEFT JUST.	JUST. BY DECIMAL	SPACE FILL	ZERO FILL ON RIGHT	ZERO FILL ON LEFT	ANY NECESSARY TRANSLATION	SENDING ZONES STRIPPED	PROPER ZONES STRIPPED OR SUPPLIED BY INTERPRETER	EDITING PERFORMED
(1)	✓		✓		✓			✓			
(2)		✓		✓			✓			✓	
(3)	✓		✓			✓			✓		
(4)	✓		✓		✓			✓		✓	✓
(5)	✓		✓		✓					✓	
(6)	✓		✓		✓			✓			✓
(7)		✓		✓			✓				✓

*ILLEGAL

Figure 7-1. Valid MOVE Statement Combinations

MULTIPLY

The function of this verb is to multiply two operands and store the results in the last-named field (which must be a numeric data-name).

The construct of this verb is:

```
MULTIPLY {literal-1   } BY {literal-2   }
         {identifier-1}    {identifier-2}

  [GIVING identifier-3] [ROUNDED]

  [;ON SIZE ERROR statement-1  [;ELSE statement-2]
```

All rules specified under the ADD statement regarding the presence of editing symbols in operands, the ON SIZE ERROR option, the ROUNDED option, the GIVING option, truncation, and the editing results apply to the MULTIPLY statement, except the maximum operand size is 125 digits for the sum of two operands.

The identifiers must be elementary item references. If GIVING is used, identifier-3 may be an elementary edited numeric item. In all other cases, the identifiers used must refer to elementary numeric items only.

If the GIVING option is used, the result of the multiplication replaces the contents of identifier-3; otherwise, it replaces the contents of identifier-2. If GIVING is not used, literal-2 is not permitted, i.e., identifier-2 must appear.

NOTE

The function of this statement is to allow the programmer to write explanatory statements in his program which are to be produced on the source program listing for documentation purposes.

The constructs of this statement are:

Option 1: Paragraph NOTE:

```
paragraph-name. NOTE any comment.
```

Option 2: Paragraph NOTE:

```
NOTE any comment.
```

Option 3: Sentence NOTE:

```
NOTE any comment.
```

Any combination of the characters from the allowable character set may be included in the character string of a NOTE statement.

If a NOTE sentence is the first sentence of a paragraph, the entire paragraph is considered to be commentary. Either Option 1 or Option 2 may be used as NOTE statements on a paragraph level.

If a NOTE statement appears as other than the first sentence of a paragraph, only the sentence constitutes a commentary. After the word NOTE is encountered, the first period followed by a space will cause the compiler to resume compilation unless the new sentence commences with the word NOTE.

Refer to the paragraph entitled CONTINUATION INDICATOR (section 3) for an explanation of comments (* or / in column 7) appearing anywhere within the source program.

OPEN

The function of this verb is to initiate the processing of both input and output files. The MCP performs checking or writing, or both, of labels and other input-output operations.

The construct of this verb is:

Option 1:

```
OPEN
{ [ INPUT file-name-1  [ { WITH LOCK [ACCESS] } ]  [file-name-2...] ] ...
                         { REVERSED             }
                         { WITH NO REWIND       }
  [ OUTPUT file-name-3  [WITH NO REWIND]  [file-name-4 ...] ] ...
  [   { INPUT-OUTPUT }  file-name-5  [file-name-6...] ] ...
  [   { I-O          }
  [   O-I file-name-7  [file-name-8...] ] ...
} ...
```

Option 2:

```
OPEN  { OUTPUT       }  file-name
      { I-O          }
      { INPUT-OUTPUT }
    [WITH PUNCH]  [ WITH { PRINT128  } ]  [WITH STACKERS]
                         { INTERPRET }
```

More than one of the options in Option 2 may be specified in the OPEN statement; for example, OPEN OUTPUT file-name WITH PUNCH INTERPRET STACKERS. When the PUNCH option is used, it implies a 96-character PUNCH operation.

When the PRINT 128 is used, it will require a work area of 96 + 128, or 224 characters. The first 96 will be for the punch output. The next 128 positions will be for printing on the card.

When the INTERPRET option is used, the output area need only be 96 characters to cause the punch data to be printed on the card.

When the STACKERS option is used alone, it will cause a write with no data transfer to the stacker selected. If used in conjunction with the other options, data will be transferred also. When the STACKERS option is not used, the cards will be selected to the default stacker on a READ or WRITE.

File-names must not be those defined as being SORT files.

At least one of the options must be specified before a file can be read.

The I-O, INPUT-OUTPUT, and O-I options pertain to disk storage files.

The OPEN statement must be executed prior to the first SEEK, READ, or WRITE statement for that file.

A second OPEN statement for a file cannot be executed prior to the execution of a CLOSE statement for that file.

A file area will not exist in memory until an OPEN statement is executed, which in turn, causes the MCP to allocate memory for the file work area, and any alternate areas or buffers. The MCP will obtain the needed information from the File Parameter Block to determine the file's characteristics. Once the file has been OPENed, memory will remain allocated until the file is programmatically CLOSEd.

The OPEN statement does not obtain or release the first data record. A READ or WRITE statement must be executed to obtain or release, respectively, the first data record.

When the first label is to be checked or written, the user's beginning label subroutine is executed if it is specified by a USE statement.

The REVERSED and the NO REWIND options can only be used with sequential, single-reel tape files.

If the peripheral ASSIGNed to the file permits rewind action, the following rules apply:

a. When neither the REVERSED nor the NO REWIND option is specified, execution of the OPEN statement for the file will cause the file to be positioned ready to read the first data-record.
b. When either the REVERSED or the NO REWIND option is specified, execution of the OPEN statement does not cause the file to be positioned. When the REVERSED option is specified, the file must be positioned at its physical end. When the NO REWIND option is specified, the file must be positioned at its physical beginning.
c. When the NO REWIND option is specified, it applies only to sequential, single-reel files stored on magnetic tape units.

When the REVERSED option is specified, the subsequent READ statements for the file makes the data-records available in reverse record order starting with the last record. Each record will be read into its record-area, and will appear as if it has been read from a forward-moving file.

If an input file is designated with the OPTIONAL clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION, the object program causes an interrogation to the MCP, for the presence or absence of a pertinent file. If this file is not present, the first READ statement for this file causes the imperative statement in the AT END clause to be executed only when the operator has responded with an optional file "mix index OF" message.

The I-O or INPUT-OUTPUT option permits the OPENing of a disk file for input and/or output operations. This option demands the existence of the file to be on the disk and cannot be used if the file is being initially created; that is, the file to be OPENed must be present in the MCP disk directory, or has been previously created and CLOSEd in the same run of the program.

When any input file option is used, the MCP immediately checks the MCP disk directory to see if the file is present, or if it has been created and CLOSEd in the same program run. The system operator will be notified in its absence, and the file can then be loaded if it is available or the program can be DSed (discontinued). If the decision is to load the file, the operator does so and then notifies the MCP to proceed with the program, by means of a "mix-index OK" message.

The O-I option is identical to OPEN I-O, with the exception that with the O-I option the file is assumed to be a new file to the disk directory. The OPEN O-I option will short cut the usual method of initially creating I-O work files within a program, e.g., OPEN OUTPUT, WRITE record(s), CLOSE WITH RELEASE, OPEN I-O, etc. The O-I option does not, nor was it intended to, replace the OPEN I-O option, since the use of OPEN O-I assumes that a new file is to be created each time.

During processing of mass storage files for which the ACCESS MODE is SEQUENTIAL, the OPEN statement supplies the initial address of the first record to be accessed.

The contents of the data-names specified in the FILE-LIMIT clause of the FILE-CONTROL paragraph (at the time the file is OPENed) are used for all checking operations while that file is OPEN. The FILE-LIMIT clause is dynamic only to this extent.

OPEN

When an OPEN OUTPUT statement is executed for a magnetic tape file, the MCP searches the assignment table for an available scratch tape, writes the label if specified by the program, and executes any USE declaratives for the file. If no scratch tape is available, a message to the operator is typed and the program is suspended until the operator mounts such a tape or one becomes available due to the termination of a multiprogramming program.

OPENing of subsequent reels of multi-reel tape files is handled automatically by the MCP and requires no special consideration by the programmer.

PERFORM

The function of this verb is to depart from the normal sequence of execution in order to execute one or more procedures, either a specified number of times or until a specified condition is satisfied. Following this departure, control is automatically returned to the normal sequence.

The construct of this verb has the following four options:

Option 1:

```
PERFORM procedure-name-1 [{THRU    } procedure-name-2]
                         {THROUGH}
```

Option 2:

```
PERFORM procedure-name-1 [{THRU    } procedure-name-2]
                         {THROUGH}

{integer-1   } TIMES
{identifier-10}
```

Option 3:

```
PERFORM procedure-name-1 [{THRU    } procedure-name-2]
                         {THROUGH}

    UNTIL condition-1
```

Option 4:

```
PERFORM procedure-name-1 [{THRU    } procedure-name-2]
                         {THROUGH}

VARYING {index-name-1 } FROM {index-name-2 }      BY
        {identifier-1 }      {identifier-2 }
                             {literal-2    }

{identifier-3} UNTIL condition-1 [AFTER {index-name-4 }
{literal-3   }                          {identifier-4 }

FROM {index-name-5 }      BY {identifier-6 }
     {identifier-5 }         {literal-6    }
     {literal-5    }
```

```
UNTIL condition-2]   [AFTER {index-name-7}   FROM
                            {identifier-7}

{index-name-8}          BY    {identifier-9}
{identifier-8}                {literal-9}
{literal-8}

UNTIL condition-3]
```

PERFORM is the means by which subroutines are executed in COBOL. The subroutines may be executed once, or a number of times, as determined by a variety of controls. A given paragraph may be PERFORMed by itself, in conjunction with another paragraph, control may pass through it in sequential operation, and it may be the object of a GO statement, all in the same program.

Each identifier represents a numeric elementary item. Identifier-10 must be described as an integer.

Each literal represents a numeric literal.

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:

a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return occurs after the last statement of procedure-name-1.

b. If procedure-name-1 is a section name and procedure-name-2 is not specified, then the return occurs after the last statement of the last paragraph in procedure-name-1.

c. If the procedure-name-2 is specified and it is a paragraph name, then the return occurs after the last statement of the paragraph.

d. If the procedure-name-2 is specified and it is a section name, then the return occurs after the last sentence of the last paragraph in the section.

There is no necessary relationship between procedure-name-1 and procedure-name-2, except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. In particular, GO TO and PERFORM statements may occur between procedure-name-1 and the end of

procedure-name-2. If there are two or more direct paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement, to which all of these paths must lead.

If control passes to these procedures by means other than a PERFORM statement, control passes thru the last statement of the procedure to the following statement, unless a PERFORM statement is executed during execution of these procedures.

If a statement within procedure-name-1 or procedure-name-2 contains a nested PERFORM, object program control will pass to the procedure-name contained in the nested statement, and the procedure will be accomplished. Program control will automatically return to the next sentence following the executed PERFORM statement. Nested PERFORM statements are allowed to any reasonable depth. However, the procedure named must return to the statement following the previously executed PERFORM and cannot contain a GO TO out of range of procedure-name-1 or procedure-name-2.

A PERFORM statement is not restricted by overlayable segment boundaries and may reference a procedure-name anywhere within the PROCEDURE DIVISION.

Option 1 is the basic PERFORM statement. A procedure referred to by this type of PERFORM statement is executed once, and then control passes to the statement following the PERFORM statement.

Option 2 is the TIMES option and, when used, the procedures are performed the number of times specified by identifier-10 or integer-1. The value of identifier-10 or integer-1 must be positive. Control is transferred to the statement following the PERFORM statement. If the value is zero, control passes immediately to the statement following the PERFORM sentence. Once the PERFORM statement has been initiated, any reference to, or manipulation of, identifier-10 will not affect the number of times the procedures are executed.

Option 3 is the UNTIL option. The specified procedures are performed until the condition specified by the UNTIL condition is TRUE. At this time, control is transferred to the statement following the PERFORM statement. If the condition is TRUE at the time that the PERFORM statement is encountered, the specified procedure is not executed.

In option 4, when one identifier is varied, identifier-1 is set equal to the current value of identifier-2, or literal-2. If the condition is false, the sequence of procedures, procedure-name-1 thru procedure-name-2, is executed once. The value of identifier-1 is augmented by the specified increment or decrement (identifier-3), and condition-1 is evaluated again. The

cycle continues until this expression is true; at this point, control passes to the statement following the PERFORM statement. If the condition is true at the beginning of execution of the PERFORM, control passes directly to the statement following the PERFORM statement. Figure 7-2 illustrates the logic of the PERFORM statement when one identifier is varied.

In option 4, when two identifiers are varied, identifier-1 and identifier-4 are set to the current value of identifier-2 and identifier-5, respectively. At the start of the PERFORM statement, condition-1 is evaluated; if ture, control is passed to the statement following the PERFORM statement; if false, condition-2 is evaluated. If condition-2 is false, procedure-name-1 thru procedure-name-2 is executed once, after which identifier-4 is augmented by identifier-6, and condition-2 is evaluated again. The cycle of execution and augmentation continues until this condition is ture. When condition-2 is true, identifier-4 is set to the current value of identifier-5; identifier-1 is augmented by identifier-3, and condition-1 is re-evaluated. The PERFORM statement is completed if condition-1 is ture; if not, the cycles continue until condition-1 is true.

Figure 7-3 illustrates the logic of the PERFORM statement when two identifiers are varied.

During the execution of the procedures associated with the PERFORM statement, any change to the VARYING variable (identifier-1 and index-name-1), the BY variable (identifier-3), the AFTER variable (identifier-4 and index-name-4), of the FROM variable (identifier-2, index-name-2, identifier-5 and index-name-5) will be taken into consideration and will affect the operation of the PERFORM statement.

When two identifiers are varied, identifier-4 goes thru a complete cycle (FROM, BY, UNTIL) each time identifier-1 is varied.

At the termination of the PERFORM statement, identifier-4 contains the current value of identifier-5. Identifier-1 has a value that exceeds the last used setting by an increment or decrement, as the case may be, unless condition-1 was true when the PERFORM statement was entered, in which case identifier-1 contains the current value of identifier-2.

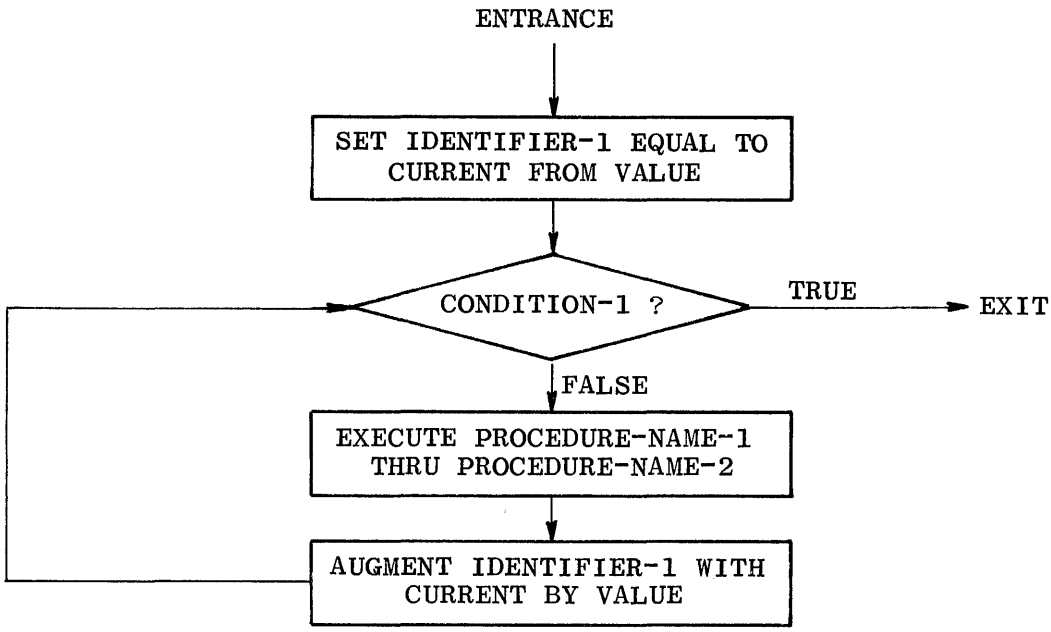


Figure 7-2. PERFORM Statement Varying One Identifier

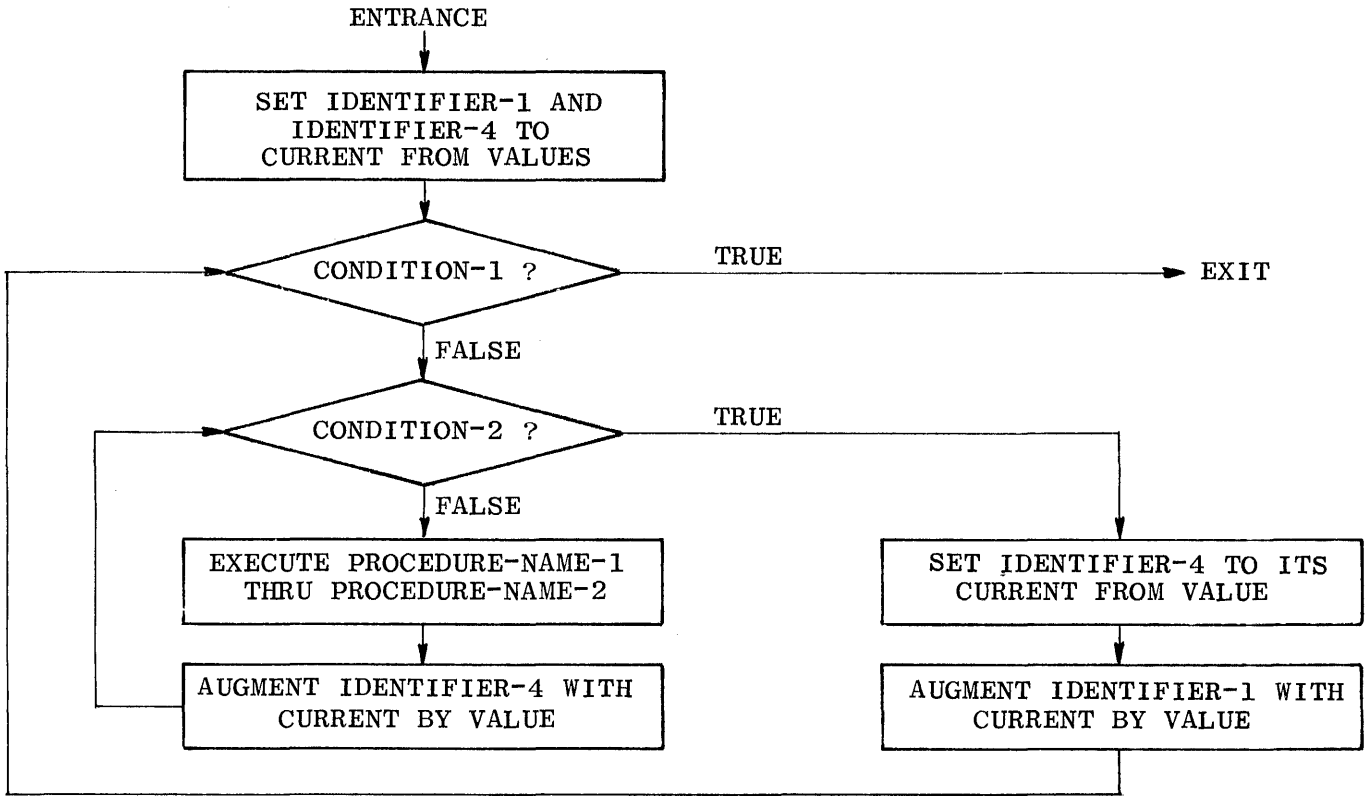
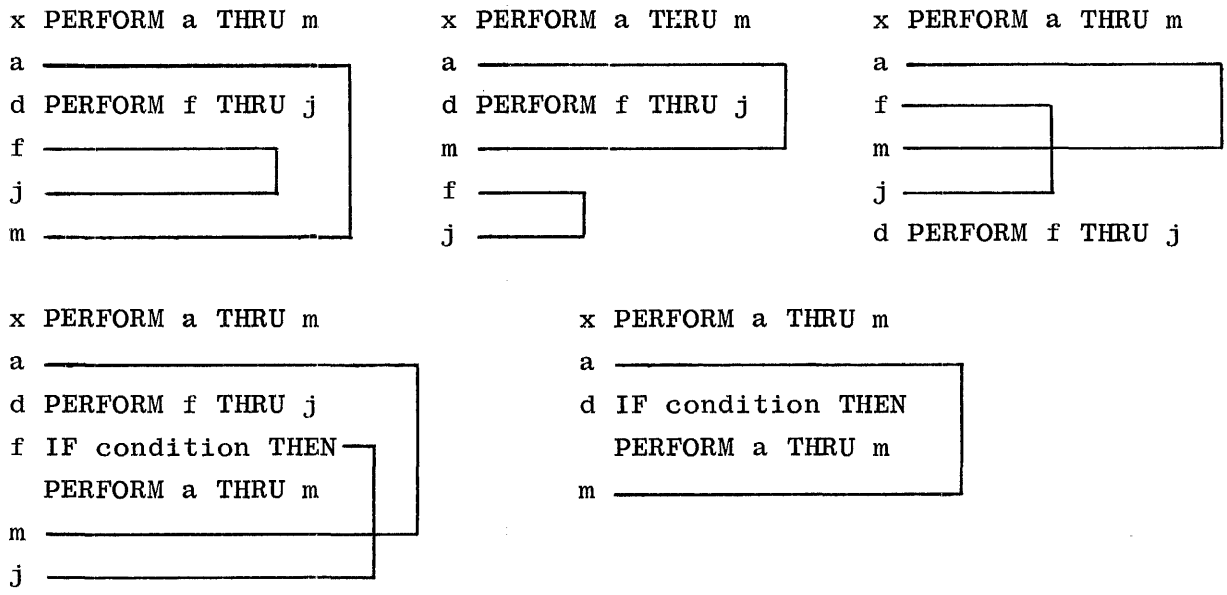


Figure 7-3. PERFORM Statement Varying Two Identifiers

In Option 4 where three conditions are required to control the number of iterations that a given procedure is to be PERFORMed, the mechanism is the same as for two-conditional control except that identifier-7 goes through a complete cycle each time that identifier-6 is added to identifier-4, which in turn goes through a complete cycle each time that identifier-1 is varied.

After the completion of option 4, identifier-4 and identifier-7 contain the current value of identifier-5 and identifier-8, respectively. Identifier-1 has a value that exceeds its last used setting by one increment or decrement value, unless condition-1 is true when the PERFORM statement is entered, in which case identifier-1 contains the current value of identifier-2.

Since the return control information is placed in the stack rather than being directed through instruction address modification, a PERFORM statement executed within the range of another PERFORM is not restricted in the range of paragraph names it may include. The examples shown below are permitted and will execute correctly.



READ

The functions of this verb are twofold, namely:

a. During processing of sequential input files, a READ statement will cause the next sequential logical record to be moved from the input buffer area to the record work area, thus making the record available to the program.

   All sequential records will be physically read into the buffer area of the file. Physical READs are performed as a function of the MCP. The READ statement permits the performance of a specified statement when an end-of-file condition is detected by the MCP.

b. For random file processing, the READ statement communicates with the MCP to explicitly cause the reading of a physical record from a disk file, and also allows performance of a specified imperative statement if the contents of the associated ACTUAL KEY data item is found to be invalid.

The construct of this verb is:

```
READ file-name RECORD [INTO identifier] [;{AT END        } statement-1
                                          {INVALID KEY   }
   [;ELSE statement-2]]
```

The AT END of file clause is used for non-disk files or for disk files being processed in the sequential access mode. If no AT END or INVALID KEY clause is stated, and one of these conditions occurs, the program will be terminated with a DS or DP message.

If, during execution of a READ statement with AT END, the logical end-of-file is reached and an attempt is made to READ that file, the statement specified in the AT END phrase is executed. After the execution of the imperative statement of the AT END phrase, a READ statement for that file must not be given without prior execution of a CLOSE statement and an OPEN statement for that file.

When the AT END clause is specified in a conditional sentence, all exits within the sentence are controlled by using the rules pertaining to the matching of IF...ELSE pairs. For example:

```
IF AAA = BBB THEN READ FILE-A, AT END
     GO TO WRAP-UP, ELSE NEXT SENTENCE, ELSE STOP RUN.
```

a. When AAA does not equal BBB, control will be passed to STOP RUN.
b. When AAA equals BBB, FILE-A is read, end-of-file is tested and if the result is TRUE program control will be transferred to the WRAP-UP procedure; however, a result of FALSE will cause program control to be transferred to the next sentence.

The INVALID KEY applies to files that are ASSIGNed to disk. The access of the file is controlled by the value contained in ACTUAL KEY.

An AT END or INVALID KEY clause must be specified when reading a file described as containing FILE-LIMITS.

An OPEN statement must be executed for a file prior to the execution of the first READ statement for that file.

When a file consists of more than one type of logical record, these records automatically share the same storage area and are equivalent to an implicit redefinition of the area. Only the information that is present in the current record is available.

If the INTO option is specified, the current record is MOVEd from the input area to the area specified by identifier according to the rules for the MOVE statement without the CORRESPONDING option.

When the INTO option is used, the record being read is available in both the data area associated with data-name and the input record area.

If a file described with the OPTIONAL clause is not present, the imperative statement in the AT END phrase is executed on the first READ. The standard End-of-File procedures are not performed. (See the OPEN and USE statements, and the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION.)

If the end of a magnetic tape file is recognized during execution of a READ statement, the following operations are carried out:

a. The standard ending reel label procedure and the user's ending reel label procedure, if specified by the USE statement, are performed. The order of execution of these two procedures is specified by the USE statement.
b. A tape swap is performed.

c. The standard beginning reel label procedure and the user's beginning label procedure, if specified, are executed. The order of execution is again specified by the USE statement.
d. The first data record on the new reel is made available.

READ with INVALID KEY is used for disk files in the random access mode. The READ statement implicitly performs the functions of the SEEK statement, except for the function of the KEY CONVERSION option for a specific disk file. If the contents of the associated ACTUAL KEY data item is out of the range indicated by FILE LIMITS, the INVALID KEY phrase will be executed.

For random disk files, the sensing of an INVALID KEY does not preclude further READs on that file, nor must the file be closed and reopened before such READs are performed.

RELEASE

The function of this verb is to cause records to be transferred to the initial phase of a SORT operation.

The construct of this verb is:

```
RELEASE record-name [FROM identifier]
```

A RELEASE statement may only be used within the range of an input procedure associated with a SORT statement.

Record-name and data-name must name different memory areas when specified.

The RELEASE statement causes the contents of record-name to be released to the initial phase of a sort. Record-name will be transferred to the specified sort-file (SD) and becomes controlled by the sort operation.

In the FROM option, the contents of data-name are MOVEd to record-name, then the contents of record-name are released to the initial phase of a sort. Moving takes place according to the rules specified for the MOVE statement without the CORRESPONDING option.

When control passes from the input procedure, the SD file consists of all records placed in it by the execution of RELEASE statements.

RETURN

The function of this verb is to obtain sorted records from the final phase of a SORT operation.

The construct of this verb is:

```
RETURN file-name  RECORD  [INTO identifier]
    ; AT END statement-1 [;ELSE statement-1]
```

File-name must be a sort file with a Sort File Description (SD) entry in the DATA DIVISION.

A RETURN statement may only be used within the range of an output procedure associated with a SORT statement for file-name.

Records automatically share the same area when a file consists of more than one type record and only the information pertinent to the current record is available.

The execution of the RETURN statement causes the next record, in the order specified by the keys listed in the SORT statement, to be made available for processing in the record area associated with the SORT file (SD).

Moving is performed according to the rules specified for the MOVE statement without the CORRESPONDING option.

When the INTO option is specified, the sorted data is available in both the input-record area and the data-area specified by data-name.

RETURN statements may not be executed within the current SORT output procedure after the AT END clause has been executed.

SEARCH

The function of this verb is to cause a search of a table to locate a table-element that satisfies a specific condition and, in turn, to adjust the associated index-name to indicate that table-element.

The construct of this verb has the following two options:

Option 1:

```
SEARCH identifier-1  [ VARYING { index-name-1 }  ]
                               { identifier-2 }

    [;AT END imperative-statement-1]

    ;WHEN condition-1    { imperative statement-2 }
                         { NEXT SENTENCE          }

   [;WHEN condition-2    { imperative statement-3 }  ...]
                         { NEXT SENTENCE          }
```

Option 2:

```
SEARCH ALL identifier-1  [;AT END imperative-statement-4]

    ;WHEN condition-3    { imperative statement-5 }
                         { NEXT SENTENCE          }
```

Identifier-1 must not be subscripted or indexed, but its description in the DATA DIVISION must contain an OCCURS clause and an INDEXED BY clause.

When Option 2 is specified, the description of identifier-3 may optionally contain the ASCENDING/DESCENDING KEY clause.

When the VARYING option is used, identifier-2 must be described as USAGE IS INDEX, or as the name of a numeric elementary item described without any positions to the right of the assumed decimal point. Identifier-2 will be incremented at the same time as the occurrence number (and by the same amount) represented by the index-name associated with identifier-1.

When Option 1 is used, condition-1, condition-2, etc., may be comprised of any conditional as described by the IF verb.

When Option 2 is used, condition-3 may consist of a relational condition incorporating the relation EQUAL, or a condition-name condition where the VALUE clause that describes the condition-name contains only a single literal. Condition-3 may be a compound condition formed from simple conditions of the type just mentioned, with AND being the only acceptable connective.

When Option 2 is used, any data-name that appears in the KEY option of identifier-3 may appear as the subject or object of a test, or be the name of the conditional variable with which the tested condition-name is associated.

When Option 1 is used, a serial type search operation takes place, starting with the current index setting. The search is immediately terminated if, at the start of execution of the statement, the index-name associated with data-identifier-1 contains a value that corresponds to an occurrence number that is greater than the highest permissible occurrence number for identifier-1. Then, if the AT END option is specified, statement-1 is executed; if AT END is not specified, control passes to the NEXT SENTENCE.

When Option 1 is used, if at the start of execution of the SEARCH statement, the index-name associated with identifier-1 contains a value that corresponds to an occurrence number that is not greater than the highest permissible occurrence number for identifier-1, the SEARCH statement will begin evaluating the conditions in the order that they are written, making use of index settings wherever specified, to determine the occurrences of those items to be tested. If none of the conditions are satisfied, the index-name for identifier-1 is incremented to obtain a reference to the next occurrence. The process is repeated using the new index-name setting for identifier-1, which corresponds to a table element which exceeds the last setting by one more occurrence, until such time as the highest permissible occurrence number is exceeded, in which case the SEARCH terminates as indicated in the previous paragraph.

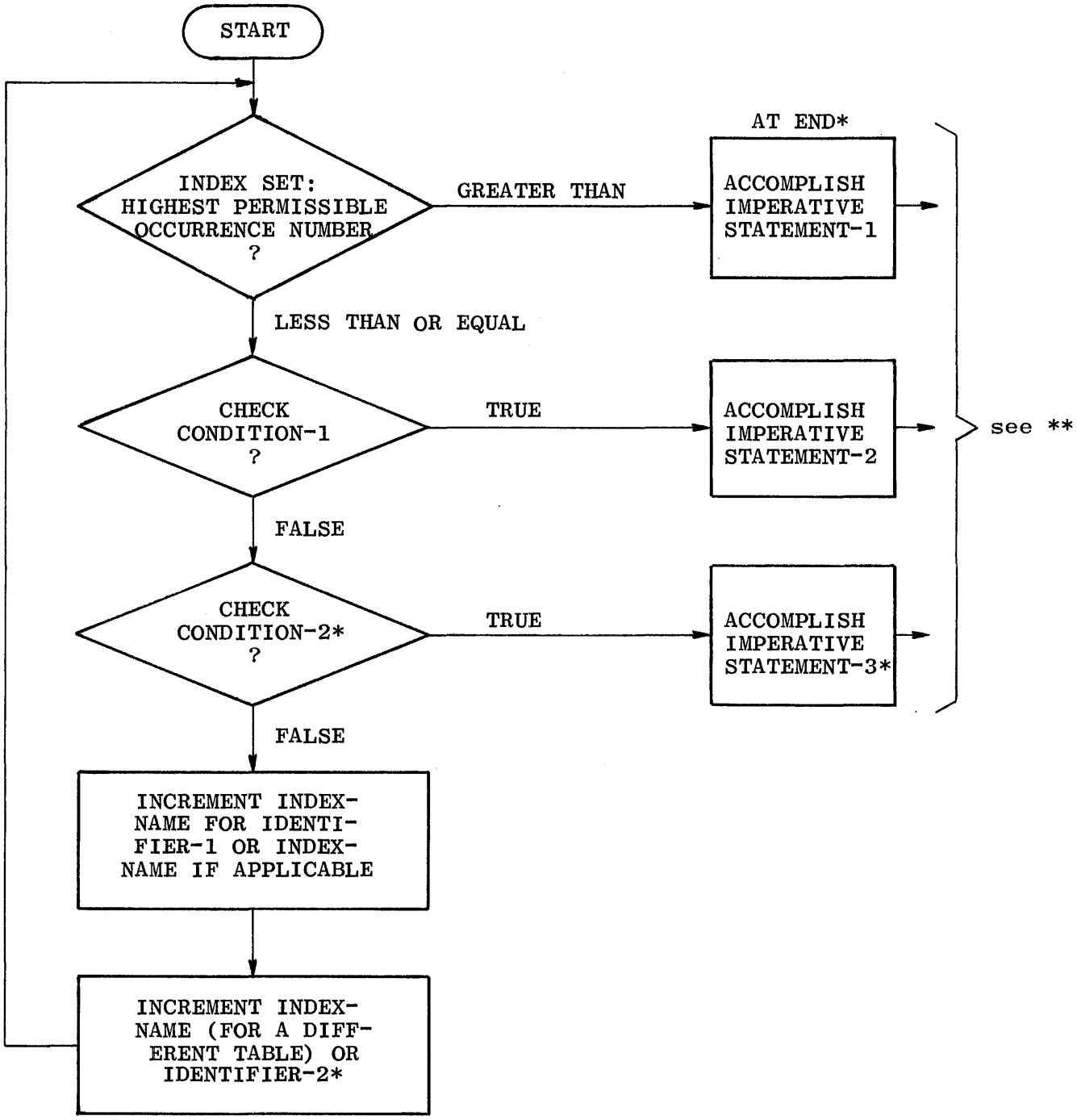
When Option 1 is used, if one of the conditions is satisfied upon its evaluation the SEARCH terminates immediately and the imperative statement associated with that condition is executed; the index-name remains set at the occurrence which caused the condition to be satisfied.

In Options 1 and 2, if the specified imperative statements do not terminate with a GO statement, then program control will pass to the next sentence, after the execution of the imperative statement.

In the VARYING option, if index-name-1 appears in the INDEXED BY option of identifier-1, then that index-name will be used for the SEARCH; otherwise, the first index-name given in the INDEXED BY option of another table entry, the occurrence number represented by index-name-1 is incremented by the same amount as, and at the same time as, the occurrence number represented by the index-name associated with identifier-1 is incremented.

In Option 2, the initial setting of the index-name for data-name-3 is ignored, the effect being the same as if it were SET to 1.

In Options 1 and 2, if identifier-1 and identifier-3 constitute an item in a group, or a hierarchy of groups, whose description contains an OCCURS clause, then each of these groups must also have an index-name associated with it. The settings of these index-names are used throughout the execution of the SEARCH statement to refer to data-names-1 and 3, or to items within its structure. These index settings are not modified by the execution of the SEARCH statement (unless stated as index-name-1), and only the index-name associated with identifier-1 and identifier-3 (and identifier-2 or index-name-1) is incremented by the SEARCH. Figure 7-4 provides an example of SEARCH operation as related to Option 1.



* These operations are only included when called for in the SEARCH statement.

** Each of the control transfers is to NEXT SENTENCE unless the imperative statement ends with a GO statement.

Figure 7-4. Example of Option 1 SEARCH Statement

SEEK

The function of this verb is to initiate the accessing of a disk file record for subsequent reading and/or writing. The construct of this verb is:

SEEK file-name RECORD [WITH KEY CONVERSION]

The specification of the KEY CONVERSION clause indicates that the user-provided USE FOR KEY CONVERSION section in the DECLARATIVE SECTION is to be executed prior to the execution of the SEEK statement. If there are no DECLARATIVES for KEY CONVERSION in a SEEK statement, then the KEY CONVERSION clause will be ignored.

A SEEK statement pertains only to disk storage files in the random access mode and may be executed prior to the execution of each READ and WRITE statement.

The SEEK statement uses the contents of the data-name in the ACTUAL KEY clause as the location of the record to be accessed. At the time of execution, the determination is made as to the validity of the contents of the ACTUAL KEY data item for the particular disk storage file. If the key is invalid, the imperative statement in the INVALID KEY clause of the next executed READ or WRITE statement for the associated file is executed.

Two SEEK statements for a disk storage file may logically follow each other. Any validity check associated with the first SEEK statement is negated by the execution of a second implicit or implied SEEK statement.

An implied SEEK is executed by the MCP whenever an explicit SEEK is missing for the specified record. An implied SEEK never executes any USE KEY CONVERSION Declaratives.

If a READ/WRITE statement for a file ASSIGNed to DISK is executed, but an explicit SEEK has not been executed since the last previous READ or WRITE for the file, then the implied SEEK statement is executed as the first step of the READ/WRITE statement.

An explicit alteration of ACTUAL KEY after the execution of an explicit SEEK has been performed, but prior to a READ/WRITE, will cause the initiation of an implied SEEK of the initial record in the sequence. For example,

a. If ACTUAL KEY is 10, then
b. READ record 10, then
c. MOVE 50 to ACTUAL KEY, then
d. WRITE record 50.

An implied SEEK of record 50 will be performed between actions c and d, above.

SET

The SET statement establishes reference points or offsets operations by setting index-names associated with table elements.

The construct of this verb has the following two options:

Option 1:

```
SET  {identifier-1}  [,  {identifier-2}]  ...  TO  {identifier-3}
     {index-name-1}      {index-name-2}            {index-name-3}
                                                   {integer-1   }
```

Option 2:

```
SET  index-name-4  [, index-name-5]  ...  {UP BY  }  {identifier-4}
                                          {DOWN BY}  {integer-2   }
```

All references to identifier-1 and index-name-1 apply equally to identifier-2 and index-name-2, respectively.

All identifiers must name either index data items, or elementary items described as an integer, except that identifier-4 must not name an index data item. When integer-1 is used, it must be a positive integer. Index-names are considered related to a given table and are defined by being specified in the INDEXED BY phrase of the OCCURS clause.

If index-name-3 is specified, the value of the index before the execution of the SET statement must correspond to an occurrence number of an element in the associated table.

If index-name-1, index-name-2 is specified, the value of the index after the execution of the SET statement must correspond to an occurrence number of an element in the associated table. The value of the index associated with an index-name after the execution of a SEARCH or PERFORM statement may be undefined.

In option 1, the following action occurs:

a. Index-name-1 is set to a value causing it to refer to the table element that corresponds in occurrence number to the table element referenced by index-name-3, identifier-3, or integer-1. If identifier-3 is an index data item, or if index-name-3 is related to the same table as index-name-1, no conversion takes place.

b. If identifier-1 is an index data item, it may be set equal to either the contents of index-name-3 or identifier-3 where identifier-3 is also an index data item; no conversion takes place in either case.

c. If identifier-1 is not an index data item, it may be set only to an occurrence number that corresponds to the value of index-name-3. Neither identifier-3 nor integer-1 can be used in this case.

d. The process is repeated for index-name-2, identifier-2, etc., if specified. Each time, the value of index-name-3 or identifier-3 is used as it was at the beginning of the execution of the statement. Any subscripting or indexing associated with identifier-1, etc., is evaluated immediately before the value of the respective data item is changed.

In option 2, the contents of index-name-4 are incremented (UP BY) or decremented (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of integer-2 or identifier-4; thereafter, the process is repeated for index-name-5, etc. Each time the value of identifier-4 is used as it was at the beginning of the execution of the statement.

Data in the figure 7-5 represents the validity of various operand combinations in the SET statement. The parenthetical comment references the lettered paragraphs above.

SENDING ITEM	RECEIVING ITEM		
	INTEGER DATA ITEM	INDEX-NAME	INDEX DATA ITEM
Integer Literal	No (c)	Valid (a)	No (b)
Integer Data Item	No (c)	Valid (a)	No (b)
Index-Name	Valid (c)	Valid (a)	Valid (b)*
Index Data Item	No (c)	Valid (a)*	Valid (b)*
*No conversion takes place.			

Figure 7-5. SET Statement Operand Combinations

SORT

The function of this verb is to sort an input file of records by transferring such data into a disk sort-file (work file) and sorting those records on a set of specified keys. The final phase of the sort operation makes each record available from the sort-file, in sorted order, to an output procedure or to an output file.

The construct of this verb is:

```
SORT [TAG-KEY | INPLACE] file-name-1

     [{PURGE | RUN | END}  ON ERROR]

          ON {DESCENDING | ASCENDING}  KEY data-name-1 [,data-name-2] ...]

          [ON {DESCENDING | ASCENDING}  KEY data-name-3 [,data-name-4] ...] ...

{INPUT PROCEDURE IS section-name-1 [{THRU | THROUGH} section-name-2]}
{USING file-name-2 [LOCK | PURGE | RELEASE]                         }

{OUTPUT PROCEDURE IS section-name-3 [{THRU | THROUGH} section-name-4]}
{GIVING file-name-3 [LOCK | RELEASE]                                 }
```

When the TAG-KEY option is used, sorting is performed on keys rather than on the entire record. The record numbers are placed in sorted order in the GIVING file-name, which must specify a record size of 8 digits and should be blocked 45. The TAG-KEY option prohibits use of INPUT or OUTPUT procedures.

When the INPLACE option is used, the amount of disk space used for sorting is minimized. The record sizes for file-name-2 and file-name-3 must be the same as file-name-1.

File-name-1 must be described in a Sort File Description (SD) entry in the DATA DIVISION, and file-name-2 and file-name-3 must be described in a File Description (FD) entry.

Section-name-1 specifies the name of the input procedure to be used before each record is passed to the sort-file, and section-name-3 specifies the output procedure to be used to obtain each sorted record from the sort-file.

Each data-name must represent data-items described in records associated with file-name-1. Data-names following the word KEY are listed from left to right, in the order of decreasing significance, without regard to their division into optional KEY clauses.

The PROCEDURE DIVISION of a source program may contain more than one SORT statement appearing anywhere in the program, except in the DECLARATIVES portion or in the input/output procedures associated with a SORT statement.

The input procedure must consist of one or more sections that are written consecutively and which do not form a part of an output procedure. The input procedure must include at least one RELEASE statement in order to transfer records to the sort-file after the object program has accomplished the required input data manipulation specified in the procedure. Input procedures can select, create and/or modify records, one at a time, as specified by the programmer.

There are three restrictions placed on procedural statements within an input or output procedure:

a. The procedure must not contain any SORT statements.
b. The input or output procedures must not contain any transfers of program control outside the range of the procedure; ALTER, GO and PERFORM statements within the procedure are not permitted to refer to procedure-names outside of the input or output procedure.
c. The remainder of the PROCEDURE DIVISION must not contain any transfers of program control to points within the input or output procedure; ALTER, GO, and PERFORM statements in the remainder of the PROCEDURE DIVISION must not refer to procedure-names within the range of the input or output procedure.

The output procedure must consist of one or more sections that are written consecutively and which do not form a part of an input procedure. The output procedure must include at least one RETURN statement in order to make each sorted record available for processing. Output procedures can select, create, and/or modify records, one at a time, as they are being returned from the sort-file.

When the ASCENDING clause is specified, the sorted sequence of the affected records is from the lowest to the highest value, according to the binary EBCDIC collating sequence.

When the DESCENDING clause is specified, the sorted sequence of the affected records is from the highest to the lowest value according to the binary EBCDIC collating sequence.

The SD record description of the sort-file must contain fully defined data-name KEY items in the relative positions of the record, as applicable. A rule to follow when using these KEY items is that when a KEY item appears in more than one type of record, the data-names must be relatively equivalent in each record and may not contain, or be subordinate to, entries containing an OCCURS clause.

When an INPUT procedure is specified, object-program control will be passed to that procedure automatically as an implicit function of encountering the generated SORT verb object code compiled into the program. The compiler will insert a "return-to-the-sort" mechanism at the end of the last section in the input procedure, and when program control passes the last statement of the input procedure, the records that have been RELEASED to file-name-1 are sorted.

If the USING option is specified, all records residing in file-name-2 will be automatically transferred to file-name-1, upon encountering the generated SORT verb object code. At the time of execution of the SORT statement, file-name-2 must not be OPEN. The SORT statement automatically performs the function necessary to OPEN, READ, USE and CLOSE file-name-2. If file-name-2 is a disk file, it must be in the Disk Directory before the SORT intrinsic is called.

If an output procedure is specified, object-program control will be passed to that procedure automatically as an implicit function when all records have become sorted. The compiler will insert a "return-to-the-object program" mechanism at the end of the last section in the output procedure; and when program control passes the last statement of the output procedure, the object program will execute the next statement following the pertinent SORT statement.

If the GIVING option is specified, all sorted records residing in file-name-1 are automatically transferred to the OUTPUT file as specified in file-name-3. At the time of execution of the SORT statement, file-name-3 must not be OPEN. File-name-3 will be automatically OPENed before the sorted records are transferred from the sort-file and, in turn, will be automatically CLOSEd after the last record in the sort-file has been transferred.

The ON ERROR option is provided to allow programmers some control over irrecoverable parity errors when input output procedures are not present in a program. PURGE will cause all records in a block containing an irrecoverable parity error to be dropped, and processing will be continued after a SPO message has been printed that gives the relative position in the file of the bad

block. This option is always assumed if no other has been defined. RUN will cause the faulty block to be used by the program and will provide the same console printer message as defined for PURGE. END will cause the usual DS or DP console printer message.

The PURGE, LOCK, and RELEASE options may be used to specify the type of file close on file-name-2 and file-name-3. Refer to description of CLOSE verb in this section. The options only apply to the USING/GIVING options.

Example:

```
SORT file-name-1 ASCENDING KEY data-name-1
USING file-name-3 PURGE
GIVING file-name-3 LOCK.
```

Beginning and ending label USE procedures are provided as follows when input/output procedures are present in the SORT statement:

a. OPEN INPUT file-name.
   USE. . . (The programmer's USE procedure will be invoked).
b. OPEN OUTPUT file-name.
   USE. . . (The programmer's USE procedure will be invoked).
c. CLOSE INPUT file-name.
   USE. . . (The programmer's USE procedure will be invoked; however, the contents of the ending input label will not be available to the USE procedure).
d. CLOSE OUTPUT file-name.
   USE. . . (The programmer's USE procedure will be invoked; however, the ending label will have been written prior to execution of the USE procedure).

NOTE

The above action provide label USE procedures at beginning and ending of files, but not during switching of reels of multi-reel files.

STOP

The function of this verb is to halt the object program temporarily or to terminate execution.

The construct of this verb is:

$$\underline{\text{STOP}} \left\{ \begin{matrix} \underline{\text{RUN}} \\ \text{literal} \end{matrix} \right\}$$

If the word RUN is used, then all files which remain OPEN will be CLOSED automatically. New files ASSIGNED to DISK will be CLOSED WITH PURGE and all others will be CLOSED WITH RELEASE. All storage areas for the object program are returned to the MCP and the job is then removed from the MCP mix.

The STOP RUN is not used for temporary stops within a program. STOP RUN must be the last statement of the program execution sequence.

If the literal option is used, the literal will be DISPLAYed on the console printer and the program will be suspended. When the operator enters the MCP continuation message mix-index AX, program execution resumes with the next sequential operation. This option is normally used for operational halts to cause the system's operator to physically accomplish an external action.

SUBTRACT

The function of this verb is to subtract one data item, or the sum of two or more, numeric data items from another item, and set the value of an item equal to the result(s).

The construct of this verb has the following three options:

Option 1:

```
SUBTRACT  {literal-1   }  [{literal-2   } ...]  FROM
          {identifier-1}  {identifier-2}

   identifier-m [ROUNDED] [identifier-n [ROUNDED] ...]
   [;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

Option 2:

```
SUBTRACT  {literal-1   }  [{literal-2   } ...]  FROM
          {identifier-1}  {identifier-2}

  {literal-m   }
  {identifier-m} GIVING identifier-n [ROUNDED] [,identifier-o [ROUNDED]]...
     [;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

Option 3:

```
SUBTRACT  {CORR          }  identifier-1 FROM identifier-2
          {CORRESPONDING}

   [ROUNDED] [;ON SIZE ERROR  statement-1 [;ELSE statement-2] ]
```

In Options 1 and 2, the identifiers used must refer only to elementary numeric items. If Option 2 is used, the data-description of identifier-n and identifier-o may be an elementary numeric edited item.

All rules specified under the ADD statement with respect to the operand size, presence of editing symbols in operands, the ON SIZE ERROR option, the ROUNDED option, the GIVING option, truncation, the editing results, the handling of intermediate results, and the CORR or CORRESPONDING option apply to the SUBTRACT statement.

When the GIVING option is not used, a literal may not be specified as the minuend.

When dealing with multiple subtrahends, the effect of the subtraction will be as if the subtrahends were first summed, and then the sum subtracted from the minuends.

TRACE

The function of this verb is to create documentation of all normal and/or control mode processing events and to output this data on a line printer.

The construct of this verb is:

```
TRACE  20
```

When a TRACE statement is encountered during object-program execution, the following actions will take place at that point in the program:

The 20 option will cause a memory dump to be taken of locations that are base relative to the program's memory assignment. Processing will continue after the memory "snapshot."

USE

The function of this verb is to specify procedures for any input/output error and/or label handling which are in addition to the standard procedures supplied by the MCP, and to calculate the ACTUAL KEY for files assigned to DISK.

The construct of this verb has the following three options:

Option 1:

```
USE AFTER STANDARD ERROR PROCEDURE ON { file-name-1 [,file-name-2] ... }
                                      { INPUT                          }
                                      { OUTPUT                         }
                                      { INPUT-OUTPUT                   }
                                      { I-O                            }
                                      { O-I                            }
```

Option 2:

```
USE { AFTER  } STANDARD { BEGINNING }
    { BEFORE }          { ENDING    }

  [ { REEL } ] LABEL PROCEDURE ON { file-name-1 [,file-name-2] ... }
    { FILE }                      { INPUT                          }
                                  { OUTPUT                         }
```

Option 3:

```
USE FOR KEY CONVERSION ON file-name-1 [,file-name-2...].
```

A USE statement, when present, must immediately follow a section header in the DECLARATIVE portion of the PROCEDURE DIVISION and must be followed 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.

If the file-name option is used as part of Option 2, the File Description entry for the file-name must not specify a LABEL RECORDS ARE OMITTED clause.

A USE statement specified for input and/or output files associated with the SORT verb will not be executed when executing the SORT unless an INPUT and/or OUTPUT PROCEDURE has been included in the program.

The USE statement itself is never executed rather, it defines the conditions calling for the execution of the USE procedures.

If neither REEL nor FILE is included in Option 2, the designated procedures are executed for both REEL and FILE labels. The REEL option is not applicable to mass storage files.

Within a given format, a file-name must not be referred to implicitly or explicitly in more than one USE statement.

USE procedures will be executed by the MCP:

a. After completion of the standard I/O error retry routine (this applies only to option 1), the record in error has been read; therefore, another READ cannot appear in the USE section, since the MCP is performing the section because of a previous READ which has been completed. Upon completion of the USE procedure, control is returned to the statement following the READ which detected the error condition. In the case of blocked or unblocked magnetic tape input, the tape will be ready to read the next record as soon as the Option 1 procedure is completed.
b. The USE AFTER STANDARD BEGINNING clause designates that the procedure following the clause must be called upon to check data on input magnetic tape beginning-file-labels, or to insert data as an output magnetic tape beginning-file-label before it is written.
c. When the USE BEFORE STANDARD ENDING clause designates that a following procedure must be called upon to check user created data contained on input magnetic tape ending file labels or to insert data onto the user's portion of an output magnetic tape ending file label before it is written.
d. Prior to any SEEK WITH KEY CONVERSION statement on files named in the USE FOR KEY CONVERSION statement.

References to common label items need not be qualified by a file-name within a USE statement. A common label item is defined as being an elementary data item that appears in every magnetic tape beginning and/or ending file-label record, but does not appear in any data record of the program.

A common label item must have the same name, description, and relative position in every magnetic tape file-label record and may only be referenced while in a USE...LABEL PROCEDURE for that file.

If the INPUT or OUTPUT option is specified, the USE...LABEL PROCEDUREs do not apply when files are described as having LABEL RECORDS OMITTED.

There must not be any reference to non-declarative procedures within a USE procedure. Conversely, in the non-declarative portion there must be no reference to procedure-names that appear in the declarative portion, except that a PERFORM statement may refer to a USE declarative or to the procedures associated with such USE declaratives.

Option 2 is not applicable to disk files.

NOTE

USE AFTER STANDARD ENDING and USE BEFORE STANDARD BEGINNING are both illegal entries in B 1700 COBOL.

WRITE

The function of this verb is to release a logical record for an output file. It is also used to vertically position forms in the printer. For mass storage files, the WRITE statement also allows the performance of a specified imperative statement if the contents of the associated ACTUAL KEY item are found to be invalid.

The construct of this verb has the following two options:

Option 1:

```
WRITE  record-name [FROM identifier-1]

[ { AFTER  }                ( { integer-1    }  LINES          ) ]
[ { BEFORE } ADVANCING      ( { identifier-2 }                 ) ]
[                           (   TO CHANNEL   { integer-2    }  ) ]
[                                            { identifier-3 }  ]

  [,AT { END-OF-PAGE } imperative-statement]
       { EOP         }

    [ TO { ERROR     } { literal-1    } ]
         { AUXILIARY } { identifier-4 }
         { STACKER   }
```

Option 2:

```
WRITE record-name   [FROM identifier]
   [;INVALID KEY statement-1 [;ELSE statement-2]]
```

An OPEN statement for a file must be executed prior to execution of the first WRITE statement for that file.

The record-name must be defined in the DATA DIVISION by means of an 01 level entry under the FD entry for the file. The record-name and identifier-1 must not be the same name, or be in two files that have the same record area.

The ADVANCING option allows the control of vertical positioning of each record on the printed page. The options are as follows:

a. When LINES is used, identifier-2 must be declared as PC 99 COMPUTATIONAL or integer-1 must be a positive integral value of 00 thru 99.
b. WRITE BEFORE ADVANCING is more efficient than AFTER ADVANCING.

c. When CHANNEL is used, identifier-3 or integer-2 must contain a positive integral value of 01 ... 11. Identifier-3 must be declared as PC 99 COMPUTATIONAL. The MCP will advance the line printer's carriage to the carriage control channel specified.

The END-OF-PAGE option applies to a file that has been assigned to a printer. When the END-OF-PAGE punch in the carriage control tape on the printer is detected, the END-OF-PAGE branch will occur.

Option 2 must be used for writing on disk files.

If the FROM option is specified, the data is moved from the areas specified by identifier-1 in option 1, to the output area, according to the rules specified for the MOVE statement without the CORR or CORRESPONDING option. After execution of the WRITE statement is completed, the information in identifier-1 is available, even though that record-name is not available.

When the WRITE statement is executed at object time, the logical record is released for output and is no longer available for referencing by the object program. Instead, the record area is ready to receive items for the next record to be written. If blocking is called for by the COBOL program, the records will be automatically blocked by the MCP.

Short blocks of records which were written during EOF or EOJ will be of no programmatic concern to the user when using the file as input at a later time.

If a write error is detected during a magnetic tape write operation, the tape record in error will be erased and a rewrite will be attempted further down the tape until the record is finally written correctly. A punch or printer write error will result in a message to the operator. The COBOL programmer need not include any USE procedures to handle write errors.

The shortest allowable blocks which can be written on 7 and 9 channel magnetic tape units are 7 and 16 bytes respectively.

If a CLOSE statement has been executed for a file, any attempt to WRITE on the file until it is OPENed again will result in an error termination.

For files which are being accessed in a SEQUENTIAL manner, the INVALID KEY clause is executed when the end of the last segment of the file (last record) has been reached and another attempt is made to WRITE into the file. The last segment of a file is specified in the FILE-LIMITS clause or the FILE CONTAINS clause. Similarly, for files being accessed in a RANDOM manner, the INVALID

KEY clause will be executed whenever the value of the ACTUAL KEY is outside the defined limits. An INVALID KEY entry must be specified when writing to a file described as containing FILE-LIMITS.

Records will be written onto DISK in either a SEQUENTIAL or RANDOM manner according to the rules given under ACCESS MODE. For RANDOM accessing, SEEK statements may be explicitly used for record determination as defined under ACCESS MODE, SEEK, and READ.

If the size and blocking of records being accessed in a RANDOM manner is such that a WRITE statement must place a record into the middle of a block without disturbing the other contents of the block, then an implicit SEEK will be given to load the block desired (provided that an explicit SEEK has not been given). If the file is being processed for INPUT-OUTPUT, then either an explicit or implicit SEEK for a READ statement will suffice to load the block between the READ and WRITE statements.

If the value of the ACTUAL KEY is changed after a SEEK statement has been given and prior to the WRITE statement, an implied SEEK will be performed and the WRITE will use the record area selected by the implied SEEK as the output record area. The value contained in the ACTUAL KEY will not be affected.

For RANDOM access, when records are unblocked, the use of a SEEK statement related exclusively to WRITE is unnecessary, and may result in an extra loading of the record from disk, because the compiler is, in general, unable to distinguish between SEEK statements that are intended to be related to a READ and those intended to be related to a WRITE.

The card record being written will be selected to the ERROR or to the AUXILIARY stackers if indicated in the particular WRITE being executed.

ZIP

The function of this verb is to cause the MCP to execute a control instruction contained within the operating object program.

The construct of this verb is:

ZIP data-name

Data-name (any level) must be assigned a value equivalent to the information contained in the MPC control card. ZIP may be used for programmatic scheduling of subordinate object programs contained in the Systems Program Library or to accomplish any of the MCP control functions as performed through the console printer or card reader.

In the statement ZIP TO-CALL-PGM2, the DATA DIVISION of the source program could contain the following entry:

```
01  TO-CALL-PGM2      PIC X(12), VALUE IS "EXECUTE PGM2".
```

The MCP will be called upon when the object program encounters the ZIP statement and will reference data-name (TO-CALL-PGM2 in the above example) to find out which control function is being called for. Using the above example, the MCP will schedule PGM2. When the time comes and the priority for PGM2 is recognized and memory space becomes available, the MCP will retrieve PGM2 from the program library and place it in the MIX for subsequent operation. The program containing the ZIP verb will proceed to the next sequential instruction following the ZIP.

CODING THE PROCEDURE DIVISION

Figure 7-6 illustrates the manner in which the PROCEDURE DIVISION can be coded.

BURROUGHS COBOL CODING FORM

ADDITIONS, DELETIONS AND CHANGES

PROGRAM: PROCEDURE DIVISION CODING | COBOL DIVISION | PAGE 1 OF 1

PROGRAMMER: DARIN | DATE | IDENT 73 80

PAGE NO. 1 3	LINE NO. 4 6	7	A 8 11	B 12 22 32 42 52 62 72
			PROC	EDURE DIVISION.
			DISK-	BACK SECTION.
			OPENER.	
				OPEN INPUT DISK-IN OUTPUT PRINT-OUT.
				MOVE 1 TO DISK-CONTROL.
				PERFORM HEADER.
			READING.	
				READ DISK-IN INTO DISK-PART.
				ADD 1 TO DISK-CONTROL.
				MOVE DISK-PART TO CARD-IMAGE.
				IF FINAL-CARD = "ENDER" GO TO FINISH.
				IF COUNTER > 37 MOVE 1 TO COUNTER GO TO SKIPPER.
				WRITE PRINT-REC BEFORE ADVANCING 2 LINES.
				ADD 2 TO COUNTER. GO TO READING.
			SKIPPER.	
				WRITE PRINT-REC FROM NEW-PRINT.
				PERFORM HEADER GO TO READING.
			HEADER.	
				MOVE SPACES TO PRINT-REC. WRITE PRINT-REC BEFORE CHANNEL 1.
				WRITE PRINT-REC FROM TITLE BEFORE ADVANCING 2 LINES.
				MOVE 3 TO COUNTER.
			FINISH.	
				CLOSE DISK-IN PRINT-OUT.
				STOP RUN.
			END-OF-JOB.	

Figure 7-6. Coding of PROCEDURE DIVISION

SECTION 8

B 1700 COBOL READER-SORTER

GENERAL

This section defines the B 1700 COBOL language facilities for handling reader-sorter files.

ENVIRONMENT DIVISION REQUIREMENTS

File Control

Each reader-sorter being used must have a file assigned to it by means of the following:

SELECT file-name ASSIGN TO READER-SORTER
RESERVE literal ALTERNATE AREAS,
ACTUAL KEY IS data-name.

Generally, a minimum of nine alternate areas should be specified to prevent any documents from going to the reject pocket. The precise number is dependent on the size of documents being read and the type of reader-sorter being used.

The ACTUAL KEY specifies the data area where document information is placed by the MCP for use in a specified COBOL USE routine. This area must be in the WORKING-STORAGE SECTION and the length must be a multiple of 112 characters. The first 24 numeric computational digits of the ACTUAL KEY contain result descriptor information for the document just read. The rest of the field contains the data read from the document, right justified, with no blank-fill. This field should be USAGE DISPLAY UNSIGNED. Document information is available in the ACTUAL KEY area during the time control is in the USE routine. At any other time, the area is undefined. The format of the result descriptor is:

NOTE

If a numeric value of 1 appears in result descriptor digits 1 through 17, the condition is considered true; otherwise, the condition is considered false and will contain a numeric value of zero.

DIGIT	DESCRIPTION	DIGIT	DESCRIPTION
1	OPERATION COMPLETE	11	TOO LATE TO READ (*)
2	EXCEPTION CONDITION	12	JAM (*)
3	NOT READY (*)	13	MISSORT (*)
4	UNENCODED DOCUMENT - NEED TO POCKET SELECT	14	BATCH TICKET - NEED TO POCKET SELECT
5	RESERVED	15	HALT - NO ITEMS BEING READ
6	CAN'T READ CHARACTER IN DOCUMENT	16	RESERVED
7	RESERVED	17	TOO LATE TO POCKET SELECT (MAIN LINE ONLY)
8	RESERVED	18-20	RESERVED
9	RESERVED	21-24	POCKET NUMBER (**) (MAIN LINE ONLY)
10	DOUBLE DOCUMENT (*)		

* Implies an invalid pocket must be selected if the condition is true.
** This pocket field is defined as USAGE DISPLAY.

Example:

```
01  THE-ACTUAL-KEY.
    02  THE-RESULT-DES.
        03  IOCOMPLETE     PC  9        CMP.
        03  AN-EXCEPTION   PC  9        CMP.
        03  NOT-READY      PC  9        CMP.
        03  THE-REST       PC  9(21)    CMP.
    02  THE-DOCUMENT       PC  X(100).
```

I-O-Control

The APPLY clause is required and specifies the specific reader-sorter read station(s) to be used. The construct of this verb is:

$$\underline{\text{APPLY}} \quad \left\{ \begin{matrix} \underline{\text{MICR}} \\ \underline{\text{OCR}} \end{matrix} [2] \quad \left[\begin{matrix} \underline{\text{MICR}} \\ \underline{\text{OCR}} \end{matrix} \right] \right\} \text{file-name } [\text{,file-name}] \ \ldots$$

Presently, only a single read station reader-sorter is implemented, so the clause is limited to:

$$\underline{\text{APPLY}} \quad \left\{ \begin{matrix} \underline{\text{MICR}} \\ \underline{\text{OCR}} \end{matrix} \right\} \quad \text{file-name} \quad [\text{,file-name}] \ \ldots$$

DATA DIVISION REQUIREMENTS

File Section

The FILE SECTION must contain an FD for each reader-sorter file selected. The record area(s) for these files should be the same type and length as the actual key data area specified for that file. The same information available in the actual key data area during pocket select is available again to the programmer

in the file record area, after each READ of that sorter file is executed during main-line processing. ("Main-line" refers to PROCEDURE DIVISION code not within a USE routine.) It is suggested that the ACTUAL KEY data area and the reader-sorter record area be formatted identically, since document information (result descriptor and data) will appear in the record area exactly as it appeared in the key area.

PROCEDURE DIVISION REQUIREMENTS

A USE procedure must be specified for each reader-sorter, or a run time error will occur. The construct of this verb is:

USE FOR READER-SORTER POCKET file-name.

File-name refers to the reader-sorter file. The USE verb must immediately follow a section header in the DECLARATIVE portion of the PROCEDURE DIVISION. The USE procedure is entered when the MCP realizes a need for pocket selection.

The USE procedure must contain a CONTROL verb which will cause the document to be pocketed. The construct of this verb is:

CONTROL file-name [STOP-FLOW] POCKET { literal / data-name }

File-name refers to the reader-sorter file. Data-name must be declared PICTURE 99 COMPUTATIONAL and contains the pocket number selected. A data-name value of 31 represents a valid reject pocket selection. A value of @FF@ in data-name represents an invalid pocket selection of the reject pocket. Certain exception conditions (see result descriptor format) require this invalid pocket. The STOP-FLOW option will stop the reader-sorter after that document is pocketed.

Since the pocket selection may be required at any poin in time, the main-line code may be interrupted at any time and control transferred to the USE procedure, i.e., the USE procedure is executed on an "as-needed" basis, and the main-line code is executed on an "as-time-is-available" basis.

There may not be any code outside the USE procedure that would cause a CONTROL...POCKET to be executed.

No I/O verb other than CONTROL...STOP-FLOW...POCKET may be executed during the USE procedure

Other variations of the CONTROL verb may be used in the MAIN-LINE portion of the program. These constructs are:

CONTROL file-name { BATCH-COUNT / POCKET-LIGHT } identifier

Identifier must be an unsigned integer.

BATCH-COUNT causes the batch counter on the reader-sorter to be advanced by 1.

POCKET-LIGHT causes the MCP to issue a pocket light op.

MICR CHARACTER TYPES

MICR control characters are:

EBCDIC		Description	Printer Graphic
0111	1011	Amount	#
0111	1100	Transit	@
0111	1010	On-Us	:
0111	1101	End of Document	'
0101	1100	Can't Read	*

MICR special characters include the control characters plus the hyphen.

MICR characters include MICR special characters plus the numeric 0 through 9.

The FORMAT verb is used to break up document information into subfields, based on MICR CONTROL characters. The construct of this verb is:

FORMAT identifier-1 INTO identifier-2 ON SIZE ERROR statement

Both identifiers must be USAGE DISPLAY. Identifier-2 is to be composed of nine fields and each field must be 20 characters in length. Data movement from low to high order. Beginning with the last character of identifier-1:

a. Move to the least significant position of a 20-character subfield of identifier-2,

   1. the source field character if it is a MICR control, and use the next source character in step b, (delimiter found); otherwise,

   2. a space and use the source character in step b (legal delimiter) for transaction code field)

b. Continue to move data until

   1. a MICR control character (excluding"cant't reads") is found, or

   2. the 20th character subfield is exceeded.

c. If a control character was found in step b, then

      1. if it is equal to the character placed in the least-significant position of the subfield, then it is moved to the next position of the same subfield, the rest of the subfield is

space-filled and the next source character is to be used in step a (complete field found, ≤ 20 characters); otherwise,

2. the rest of the subfield is space-filled and the control character is used in step a (field delimiters not equal).

d. If the subfield size was exceeded in step b,

1. The ON SIZE ERROR statement is executed and the contents of identifier-2 are undefined if document subfield is longer than 20 characters.

e. Steps a through c are repeated until

1. an end-of-document character is found in identifier-1 and is moved as in step a,
2. identifier-1 is exhausted, in which case an end of document is moved as in step a.
3. identifier-2 is exhausted, in which case the ON SIZE ERROR statement is executed. The contents of identifier-2 are undefined.

The COBOL source FORMAT statement will cause a FORMAT S-OP to be generated

```
MCF   COPX1,   COPX2
```

COPX1 corresponds to identifier-1
COPX2 corresponds to identifier-2

Example 1: (b = blank)

Identifier-1

```
':654321:@8765-4321@:765-4321:97#0987654321#
```

Identifier-2

```
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb'
bbbbbbbbbbbb:654321:
bbbbbbbbb@8765-4321@
bbbbbbbbbb:765-4321:
bbbbbbbbbbbbbbbbb97b
bbbbbbbb#0987654321#
```

Example 2:

Identifier-1

```
'654321:8765-4321@*765-4321*97#*98765432**
```

Identifier-2

bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbb'
bbbbbbbbbbbb654321:
bbbbbbbbbb8765-4321@
bbbbbbb*765-4321*97#
bbbbbbbb*98765432**b

The FORMAT verb may be used in both the USE routine and in main line code.

The ON SIZE ERROR path will be taken (1) when the subfield on the document being formatted exceeds 20 characters in length, (2) when the document being formatted was non-encoded (as indicated by digit 4 of the result descriptor), (3) when the total number of MICR characters plus "can't read" characters in identifier-1 exceeds the length of identifier-2.

Considerations After the Format Verb Has Been Executed

If a CANT.READ (*) character is placed into any subfield, the most significant character of that subfield will be set to a -1 (@D1@), and if there are no CANT.READ characters in a subfield, the most-significant character will be set to a blank (@40@).

If the first (least-significant) character of a subfield is sensed as a CANT.READ, it is treated as a non-control character and stored into the subfield after a blank (@40@) is stored as the least-significant character.

The MICR-EDIT verb is used to edit document subfields and also to count the number of MICR characters in the field. The construct of the verb is:

```
MICR-EDIT identifier-1 INTO identifier-2
```

Identifier-1 and identifier-2 must be alphanumeric. The MICR-EDIT verb may appear anywhere in the PROCEDURE division.

This verb moves identifier-1 to identifier-2, right-justified, deleting all MICR special characters (except "can't reads") and spaces. Each deleted character of identifier-1 causes the remaining unmoved portions to be shifted right one position in the identifier-2. Identifier-2 is left zero-filled, if necessary. A count of all characters, except CANT READ, moved from identifier-1 is provided in the COBOL special register TALLY.

The COBOL source MICR-EDIT statement will cause a MICR-EDIT S-OP to be generated

MCE COPX1, COPX2, COPX3 is generated.

COPX1 corresponds to identifier-1 (Source)
COPX2 corresponds to identifier-2 (Receiving)
COPX3 corresponds to TALLY

	EXAMPLE 1:	EXAMPLE 2:
Source and receiving:	Picture PC X(20)	PC X(20)
Source:	bbbbbbbbbb12-345678@	bbbbbbbbbb12-345678*
Receiving:	00000000000012345678	00000000000012345678*
TALLY	8	8

where,
b = blank and * = can't read.

The READ verb can only appear in the PROCEDURE DIVISION main-line code, and is used to retrieve document information for processing. The construct of this verb is:

READ file-name [INTO identifier-1]

The result descriptor for the READ will be placed in the first 24 digit positions of file-name's record area. The data that is read is stored following the descriptor; consequently, the file record size declared should be large enough to handle the maximum length of data expected. (There is no blank-fill if the data length is less than that declared.)

If INTO is specified, data will be moved from the file record area into identifier-1, according to the rules for COBOL moves. Note that in all cases, this READ will retrieve information on a document that has already been pocket-selected. The MCP, however, keeps track of how far behind the main-line processing is from the USE routine pocket-selecting,and issues a STOP-FLOW to the reader-sorter to allow main-line to catch up. Presently, a difference of six documents will cause a STOP-FLOW.

Programming Considerations

USE Routine

a. The pocket selection command must be the last instruction executed in the USE ROUTINE. If any instructions follow the POCKET command, they will not be executed.

b. If the exception digit is turned on in the result descriptor, the following exceptions should be tested until the problem area is located:

NOT READY (*)
UNENCODED DOCUMENT
CAN'T READ
DOUBLE DOCUMENT (*)
TOO LATE TO READ (*)
JAM (*)
MISSORT (*)
BATCH TICKET

If the above exceptions are not true, then the document was read correctly and should be processed normally.

* Implies an invalid pocket must be selected if the condition is true, @FF@

c. When a NOT READY, DOUBLE DOCUMENT, TOO LATE TO READ, JAM, or MISSORT is encountered in the USE ROUTINE, the result descriptor will reflect that condition or multiple conditions, and the programmer must pocket select an invalid pocket, @FF@. In addition, the invalid data is received by the USE ROUTINE and should not be processed by the programmer.

d. When an UNENCODED DOCUMENT is detected in the USE ROUTINE, the document should be pocket-selected to the reject pocket, @31@, also the data is invalid and should not be processed by the program. Also, the result descriptor will reflect this condition.

e. When the programmer exceeds the time allotted to pocket-select that document, the result descriptor will reflect the "too late to pocket condition," which can be tested in main memory but not in the USE ROUTINE.

f. When the programmer desires to issue either a POCKET-LIGHT or a BATCH-COUNT command in main-line code, the USE ROUTINE must have the following statement:

CONTROL file-name [STOP-FLOW] POCKET { literal / identifier }

g. Only the STOP-FLOW and POCKET commands may be used in the USE ROUTINE as communicates to the READER-SORTER.

h. If the MICR FORMAT or MICR-EDIT statements are used in the USE ROUTINE, those work areas must not be the same areas used in MAIN LINE.

i. If both MAIN LINE and the USE ROUTINE use the EXAMINE statement, the TALLY register must be saved at the beginning of the USE ROUTINE and restored at the end of the USE ROUTINE (before the pocket selection command is executed). This is required because the Main Line may be interrupted at any time to service the READER-SORTER.

Main Line

a. Only the POCKET-LIGHT and BATCH-COUNT statements may be issued in MAIN LINE.

b. All result descriptors and checks or exception conditions received in the USE ROUTINE will also be received by MAIN LINE.

c. If the exception digit is turned on in the result descriptor, the following exceptions should be tested:

NOT READY
UNENCODED DOCUMENT
CAN'T READ
DOUBLE DOCUMENT
TOO LATE TO READ
TOO LATE TO POCKET SELECT
JAM
MISSORT
BATCH TICKET

If the above exceptions are not ture, then the document was read correctly and should be processed normally.

d. When a NOT READY, UNENCODED DOCUMENT, DOUBLE DOCUMENT, TOO LATE TO READ, JAM, or MISSORT is encountered in the MAIN LINE ROUTINE, the condition(s) will be reflected by the result descriptor. Also, the data that is received by the MAIN LINE is invalid and should not be processed by the programmer.

e. Before a POCKET-LIGHT or BATCH-COUNT statement is issued in MAIN LINE, the 15th digit of the result descriptor must be tested to ensure that the READER-SORTER is not in flow mode. Also, a counter should be kept in the USE ROUTINE and compared against in MAIN LINE to ensure that this is the document to act on.

f. When the MICR FORMAT or MICR-EDIT statements are used in MAIN LINE, those work areas must not be the same area as used in the USE ROUTINE.

g. The last two bytes of the result descriptor contain the pocket number to which that item was pocket-selected. Examples:

   1. F1F0 - the document was pocketed in pocket 10.

   2. F3F1 - the document was pocketed in the REJECT POCKET.

   If the USE ROUTINE selected the invalid pocket, @FF@, the MCP will return a value greater than 32 in the pocket number of the result descriptor.

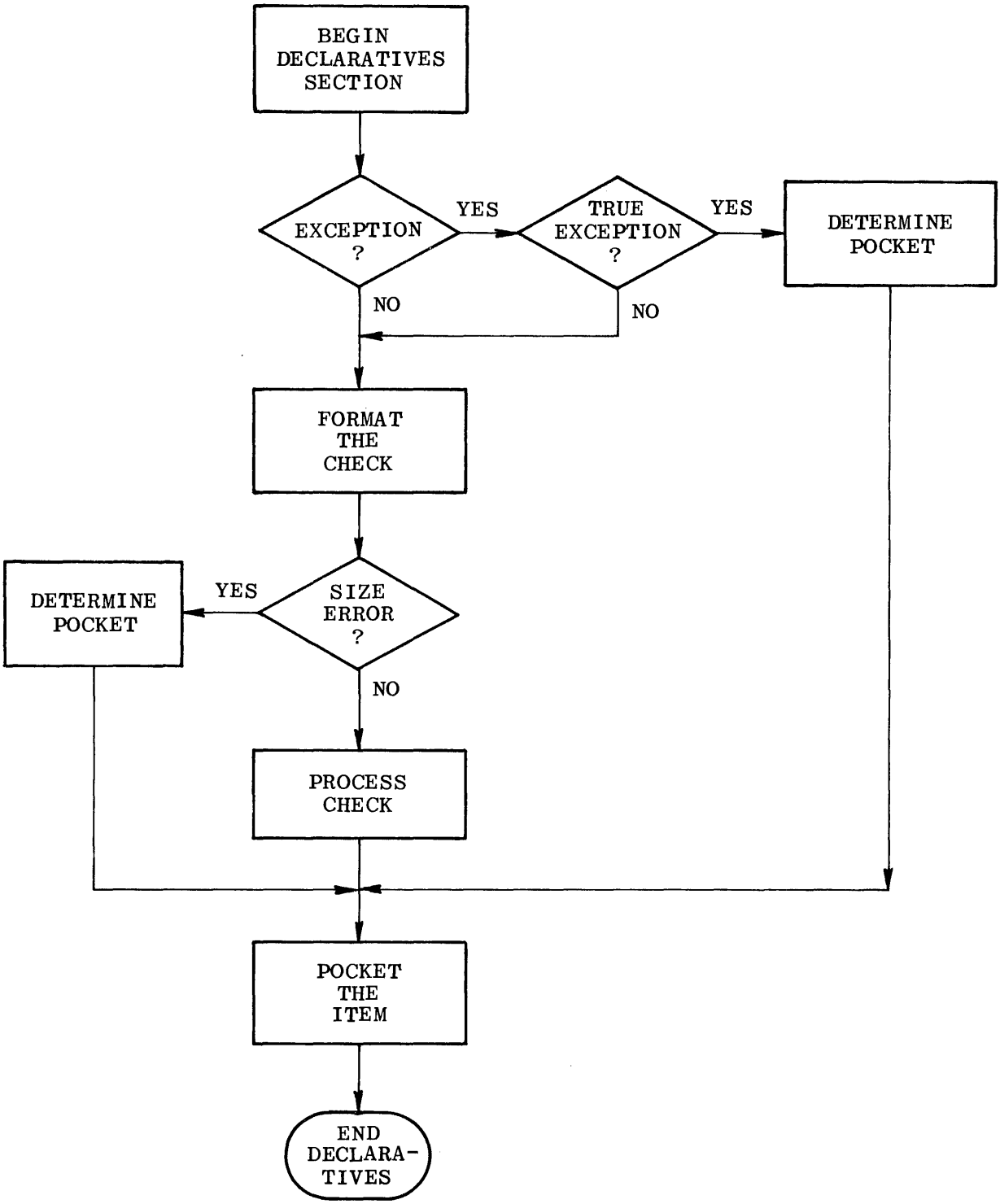
TIMING REQUIREMENTS

After a document starts through the reader-sorter, there is a time span (dependent on the type of sorter being used), during which the device must receive pocket-select information. If the time limit is exceeded, the document will be rejected, the result descriptor will reflect this condition, and processing will continue. The programmer therefore must ensure that minimal time is spent in the USE ROUTINE.

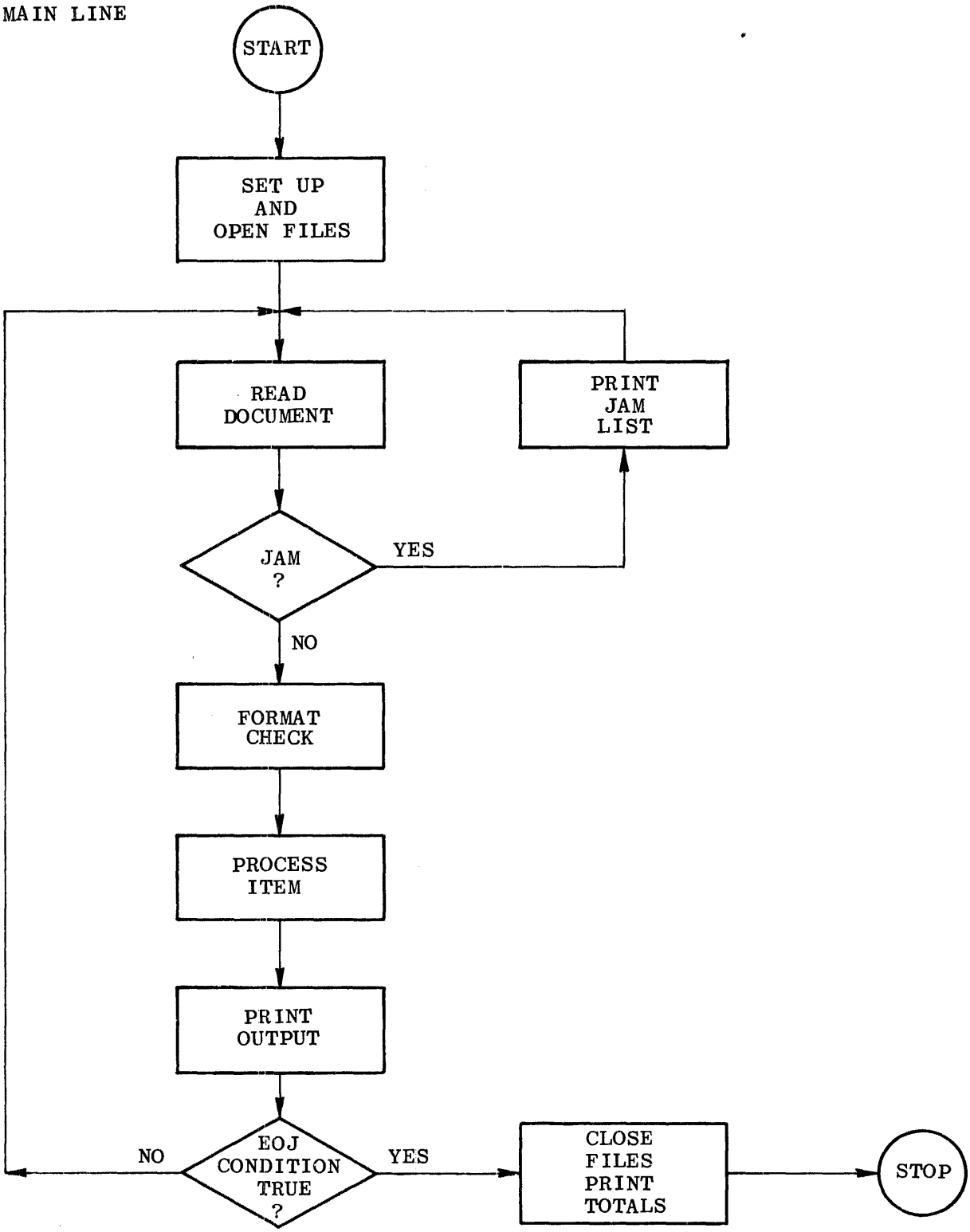
The following sample program reads, checks, and prints out the check image, along with the result descriptor for that check. A running total is kept of the check amount fields, and is printed at the end.

NOTE
No MICR-EDIT verbs are required in this program.

USE ROUTINE SAMPLE PROGRAM



MAIN LINE


START
SET UP AND OPEN FILES
READ DOCUMENT
PRINT JAM LIST
JAM ?
YES
NO
FORMAT CHECK
PROCESS ITEM
PRINT OUTPUT
EOJ CONDITION TRUE ?
NO
YES
CLOSE FILES PRINT TOTALS
STOP

SECTION 9

DATA COMMUNICATIONS

GENERAL

This section deals with the COBOL constructs of the PROCEDURE DIVISION required to activate the data communications equipment as defined by the ASSIGN to hardware-name clause.

SPECIFIC VERB FORMATS

The following differences exist in READ, WRITE, and USE when I/O is to be performed on remote files.

The READ statements will wait for a message and suspend program execution if no messages are queued for that program.

AT END will be executed when the datacom network controller receives a QC message, or when the MCS (if one exists) does an MCS.COMMUNICATE with MESSAGE.TYPE set to 1 and MESSAGE.VARIANT set to 3.

WRITE requires MESSAGE-TYPE to be set to 0 if an actual key is used, as well as TEXT-LENGTH set to the actual message length, and STATION-RSN to be set to the correct relative station number of the terminal to which the message is to be sent.

USE AFTER STANDARD ERROR PROCEDURE ON file-name must be specified to avoid a DS or DP condition when library request sets in NDL are used. (This is not a problem but is a standard procedure to follow.) The execution of TERMINATE ERROR by the Network Controller will always invoke the USE procedure in the associated application program.

SECTION 10

INTER-PROGRAM COMMUNICATION

GENERAL

This section describes the COBOL inter-program communication (core-to-core transfer), which is achieved by means of "message queues" in memory. (A queue is treated by the MCP and compilers almost exactly as if it were a normal file.) A discussion on the implementation of queues is followed by the syntax for COBOL queues.

QUEUES FILES

A queue file or queue file family may consist of one or several queues called subqueues. The number of queues is primarily constrained by memory, since there is a resident queue dictionary entry for each queue. With each queue file that belongs to a queue family, there must be associated a number, zero-relative, by which it is referenced. This number is provided in the key part of the read or write communicate. The messages in the queues are considered blocks of data which are transferred to and from the MCP I/O buffer according to the specifications in the communicate and the File Parameter Block (FPB). The MCP will handle the blocking and deblocking of the buffer.

In the case of single queues, the MCP accesses the queue by a pointer in the file's File Information Block (FIB). The name of that queue is supplied in the FPB as a multi-file ID and file ID.

In the case of multiple queues (queue family) the file name for a queue is created from the FPB entry for the multi-file ID and from the key passed in the read or write communicate.

During OPEN, the MCP associated an FIB with one or several queues. The number of queues that a user wishes to associate with an FIB is determined by the contents of the "number-of-buffers-requested" field within the FIB. Note that this does not mean the user sets the field by assigning buffers. See the appropriate syntax in this discussion. With each new queue, a dictionary entry and a buffer are created for each queue in the file. The queue dictionary contains the name of the queue, a pointer to the first message, a pointer to the last message (all messages are linked), a user count, and a link to the

next and previous dictionary entries. The size of the buffer is determined, in the usual manner, from the requested record size and number of records per block. For a family of n subqueues, the MCP creates a dictionary entry for queues of the following name: family ID/ ##NNNNNNNN, where NNNNNNNN ranges from 0 to n.

On a write communicate, the logical record in the work area is transferred to the queue file's write buffer. If the buffer is full, the MCP attempts to transfer the buffer to the appropriate queue. If a buffer is transferred to the queue, then any programs awaiting the queue are notified. The length of the message in the queue will be the block size of the file that wrote it to the queue, but the length of the message as delivered to a program is dependent on the block size of the file that reads it.

On a read communicate from multiple queues, it is possible to request a block of data from a specific subqueue, or to request a block of data from any queue. In the latter case, the message associated with that queue family for the longest period of time will be delivered. If the key passed in the read communicate is 0, the oldest message in the queue family is transferred to the work area from the appropriate buffer; otherwise, the top (first-in) message from the subqueue corresponding to the key is delivered. The buffer is then checked; if it is empty, the MCP transfers a queue entry to the buffer. If it is successful, a logical record is transferred from the buffer to the work area. (If the present key is different from that of the previous key, the MCP transfers a message from the appropriate queue to the buffer.) The logical record is then transferred to the work area. The buffer is checked and, if empty, a message from the queue is transferred to the buffer. The logical record is then transferred to the work area. Note that when the message is transferred from a queue to a buffer, it is transferred as a character string. All rules for the handling of character strings are followed.

The read and write communicates may request the status of two conditions: invalid key number (a queue number greater than the number in the queue family or file contains clause) and end of file. The queue number passed dynamically as part of the communicate (KEY) has the potential for being out of range of the number of queues requested at OPEN time. If requested, the MCP will report the condition; otherwise, the MCP will terminate the issuing program with an invalid key message. On a read communicate, the user may also request a report of end of file. EOF for a queue file means that at the time a user program issued the read, the buffer was empty and there was no message in the queue. If no EOF reporting is requested, the MCP will place the process reading the queue in a wait state.

NOTE

A message does not get placed in the queue until a write buffer associated with that queue becomes full. Hence, it is possible to do several logical writes to a queue, in the case of a blocked queue file, yet the reading process is receiving EOF conditions.

During CLOSE, the MCP will flush all write buffers to the appropriate queue. The user count for the queue will be decremented, and when it is equal to 0, the queue directory, message list, and other associated space are released.

QUEUES IN COBOL

As with other files, queue files must be selected, described in the File Section, opened, read, written and closed in the PROCEDURE DIVISION. The compiler will generate the communicates and control information necessary for the MCP. Inter-program communication may be utilized together with, or independently from, NDL, or any other Data Communications functions.

File-Control

```
SELECT  file-name  ASSIGN TO { QUEUE }  ACTUAL KEY IS data-name-1
```

QUEUE is used for inter-program communication in general. Optionally, each type may specify an ACTUAL KEY. If specified for QUEUE files, the KEY must be described as PC 9(8) CMP.

File Section

```
FD
  FILE CONTAINS integer          { QUEUE  }
                                 { QUEUES }

  VALUE OF IDENTIFICATION IS     { data-name }
                                 { literal   }
```

The FILE CONTAINS clause specifies the number of queues and must be specified if the number is greater than 1. The FILE clause determines the queue family size (single or multiple). The literal of the VALUE clause is of the for "multi-file-name" or "multi-file-name"/"file-name". For QUEUE files, the first 10 characters of the multi-file-name are used to identify the QUEUE. BLOCK and RECORD clauses do apply, as with other files.

Procedure Division

```
OPEN    { INPUT          }
        { INPUT-OUTPUT   }      file-name.
        { OUTPUT-INPUT   }
        { OUTPUT         }

WRITE file-record  [FROM data-name]
READ file-name     [INTO data-name]   AT END any-statement
```

If the AT END is specified, the statement will be executed when no data is available; otherwise, the program is suspended until data is available. On a WRITE to a queue, the KEY must have the value set before the WRITE is issued. On the READ from a queue, the value of the KEY must be set before the READ is issued. If the KEY value is 0, it means a READ for the oldest message in the queue.

On a READ, if exception conditions are to be handled, they must be handled by a "USE AFTER STANDARD ERROR" procedure.

```
CLOSE    {queue-file}
```

When the queue file is closed and the user count is equal to 0, the queue space is released. None of the options of the CLOSE verb apply to queue files.

SECTION 11

COBOL COMPILER CONTROL

GENERAL

The COBOL compiler, in conjunction with the Master Control Program, allows for various types of actions during compilation and is explained in the text that follows.

COMPILATION CARD DECK

Control of the COBOL source-language input is derived from presenting the compilation card deck, illustrated in figure 11-1, to the MCP.

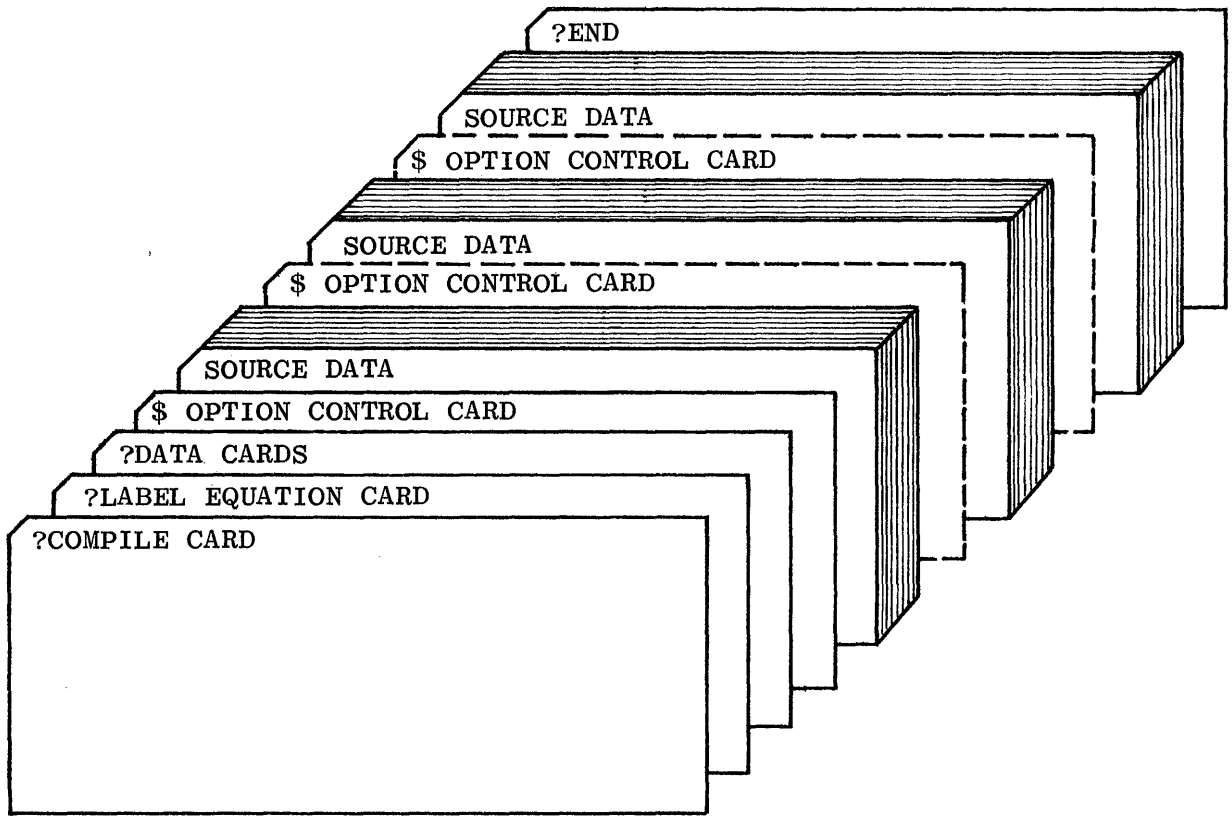


Figure 11-1. Compilation Card Deck

The compilation card deck is comprised of several cards; these cards, along with a detailed discussion of their function, are presented in the paragraphs that follow.

?COMPILE CARD

The first input control card instructs the MCP to call the COBOL compiler and to compile the indicated program-name (P-N) by means of one of the following options:

a. To compile and run the resultant object program, the card is coded:

```
?COMPILE P-N WITH COBOL
```

b. To compile for a syntax check only, the card is coded:

```
?COMPILE P-N WITH COBOL SYNTAX
```

c. To compile and place the resultant object code into the Systems Library, the card is coded:

```
?COMPILE P-N WITH COBOL LIBRARY
```

d. To compile and place the resultant object code into the Systems Library, and then run the object program, the card is coded:

```
?COMPILE P-N WITH COBOL SAVE
```

NOTE

The word WITH is for readability only and may be excluded from the above statements.

The absence of the ?COMPILE card will cause the System Operator to manually execute one of the above options through the SPO, using the MCP's CC notation in place of the invalid character ("?").

MCP LABEL CARD

The second control card, excluding Label Equation cards, is the MCP LABEL Card and is formatted in the following form:

```
?DATA CARDS (indicates EBCDIC or BCD source language input).
```

The absence of the MCP LABEL card will cause the message

```
**NO FILE file-name program-name = mix-index
```

to be displayed on the SPO. The System Operator will not know the proper IL message to give the MCP (because of the options involved), without specific instructions by the programmer.

$OPTION CONTROL CARD

The third card, excluding Label Equation Cards, is the COBOL compiler option control card ($ sign in column 7). This card is used to notify the compiler as to which options are required during the compilation. If this card is omitted, $CARD LIST CHECK SINGLE will be assumed. There must be at least one space between each item on the control card. The options may be in any order. Columns 1 through 6 of the $ card are used for sequence numbers. Any number of $ cards may be used and may appear anywhere in the source deck. The options specified will become either active or inactive from that point on. The options available for the COBOL compiler option control cards are as follows:

ANSI	Causes the compiler to inhibit certain non-ANSI extensions.
CARD	This option is for documentation only. The input is from the source language cards or from paper tape.
CHECK	Causes the compiler to check for sequence errors and print a warning message for each sequence error. The CHECK option is set, by decault, at the beginning of each compile, but may be terminated with the NO option.
CODE	Lists the object code following each line of source code from the point of insertion.
CONTROL	Prints the $ Option Control Cards on the output listing. The LIST option must be set.
DOUBLE	Causes the output listing to be printed in a double-spaced format.
HEX CODE	Causes all addresses on the CODE listing to be in hexadecimal format. If this option is omitted, the addresses will be in decimal format.
LIST	Creates a single-spaced output listing of the source language input, with error and warning message (or both), where required.
LISTP	This option causes the compiler to force listing of source images and to print errors as they occur.
MERGE	Primary input is from a source other than a card reader and may be merged with a patch deck in the

card reader. It is assumed to be from a disk file, with a file-ID of COBOLW/SOURCE, by default. If it is desirable to change the input file-ID or change the input device from disk to tape, a label equation card must be used. The NEW option may be used with the MERGE option to create a new output source file plus changes.

NEW — Creates a NEW output source file with changes, if any, entered through the use of the MERGE option, but does not include the compiler option cards, if any, which must be merged in from the card reader when the compilation is from disk or tape. The output file will be created on disk, by default, with the file-ID of COBOLW/SOURCE. If it is desirable to change the output file-ID device from disk to tape, a label equation card must be used.

NO — When the NO option precedes one of the above options (with the exception of MERGE which cannot be terminated), it will terminate the function of that option.

NO DEBUG — When this option is specified, the compiler will not generate monitor object code even though the statements are left in the source program. This permits the user to approximate a conditional compile for the debugging facilities.

NO SEQ — Terminates the SEQ option and resumes using the sequence number in the source statement as it is read in.

NOCOP — This option causes the compiler to generate current operand table entries in-line in the code. This option requires more memory in order to run, but it will increase execution speed by approximately 2 per cent.

Non-numeric literal — Inserted in columns 73-80 of all following card image for creation of a new source file and/or listing. This option can be reset or

	set by a subsequent control card, with the area between the quote marks containing blank characters.
REFERENCE	Provides monitoring of data-names specified in the MONITOR declaration and referenced in the program, even if the data-name values are unchanged. This option must be used in conjunction with the MONITOR statements.
SEQ	Starts resequencing the output listing and the new source file, if applicable, from the last sequence number read in and it increments the sequence number by 10 or by the last increment presented in a previous $ option card. When resequencing starts at the beginning of the program source statements, the sequence will start with 000010.
SEQ nnnnnn	Starts resequencing the output listing and new source file, if applicable, from the sequence number specified by nnnnnn, incrementing the sequence numbers by 10.
SEQ + nnnnnn	Starts resequencing the output listing and new source file, if applicable, from the last sequence number read in, incrementing by the number specified by +nnnnnn. When resequencing starts at the beginning of the program source statements, the sequence will start with 000010.
SEQ nnnnnn+nnnnnn	Starts resequencing the output listing and new source file, if applicable, from the sequence number specified by nnnnnn, incrementing by the value of +nnnnnn.
SINGLE	Causes the output listing to be printed in a single-spaced format.
SPEC	This option negates the CONTROL and LIST options and causes only the syntax errors and associated source code to be printed if syntax errors occur. Otherwise, the CONTROL and LIST options remain in effect.

SUPPRESS	Suppresses all warning messages except sequence error messages. The sequence error message can be suppressed with the NO CHECK option.

The NEW option does not have to be included when operating with a tape or disk source input, thus allowing temporary source language alterations without creating a new source output file.

The MERGE option without the NEW option allows a disk or tape input file to be referenced and to have external source images included from the card reader on the output listing and in the object program. A new output file will not be created.

Columns 1-6 of the compiler option control card may be left blank when compilation is from cards. A sequence number is required when compilation is from tape or disk, if the insertion of the $ option is requested within the source input.

SOURCE DATA CARD

Source data cards follow the $ option control cards. The following source cards are used to create an updated version of the source input file or to cause temporary changes to the tape or disk source language input:

a. VOID nnnnnn Patch Card. The punch sequence number in card columns 1-6 is followed by a $ in column 7, and then the word VOID. This will delete the source records from the sequence number in the first six positions of the VOID card through the sequence number specified by nnnnnn. If "n" is left blank only the source record identified by the sequence number in the VOID card will be deleted from the compilation and the output listing, tape or disk files.

b. Change or Addition Patch Card. Punch sequence number in card columns 1-6 and changed or added source language data in applicable card columns. These cards must be in the proper sequence for the source input file in order to be properly merged into that file.

The COBOL compiler has the capability of merging inputs from two sources (punched cards or paper tape, either of which may be merged with magnetic tape or disk) on the basis of the sequence numbers.

When merging inputs, the output compilation listing will indicate all inserts and replacements (or both).

All of the $ options may be inserted at any point within the source language input data. Once an option has been set it will remain set until reset with the NO option is another $ option card. In the case of the non-numeric literal it must be reset by coding a non-numeric literal with blanks.

LABEL EQUATION CARD

This card may be used to change a compiler file-name in order to avoid duplication of file-names when operating in a multiprogramming environment.

The label equation card must be used in conjunction with the MERGE and NEW options when the primary input or output is from magnetic tape, the input disk file does not have a file-ID of SOURCE, or when a file-ID other than COBOLW/SOURCE is desired for the new disk output file.

The format for the LABEL EQUATION CARD is:

```
?FILE internal file-name
users choice of file-IDs, file-attributes ... ;
```

The label equation card (or cards), if used, must end with a semicolon, must immediately follow the ?COMPILE ... control card, and precede the MCP LABEL control card (refer to figure 11-1).

The internal file-names and external file-ids of the COBOL compiler are used for label equation as follows:

INTERNAL FILE-NAME	EXTERNAL FILE-ID	DESCRIPTION
CARDS	CARDS	Input file from the card reader. If $ MERGE is used this file will be merged with the input file on disk or tape. The default input is from the card reader.
SOURCE	COBOLW/SOURCE	Input file from disk or tape when the MERGE option is used. The default input is from disk.
NEWSOURCE	COBOLW/SOURCE	Output file to disk or tape for a NEW source file when the NEW option is used. The default output is to disk.
LINE	LINE	Source output listing to the line printer.

The following are examples of the label equation uses.

Example 1:

To compile a COBOL program from the card reader and create a copy of the source program blocked five on a disk file with the file-ID of COBOL/TEST1, the following Label Equation (FILE) cards could be used:

```
? COMPILE P-N WITH COBOL SYNTAX
? FILE NEWSOURCE NAME COBOL/TEST1 RECORDS.BLOCK 5;
? DATA CARDS
  $ CARD LIST DOUBLE NEW
... SOURCE PROGRAM DECK ...
? END
```

To create the same program file on magnetic tape, use the following FILE card:

```
? FILE NEWSOURCE NAME COBOL/TEST1 TAPE RECORDS.BLOCK 5;
```

Example 2:

To compile a COBOL program from a disk file which had been created by the default option of the $ NEW option and to create a new source file on disk with the file-ID of TEST2, the following FILE card could be used:

```
? COMPILE P-N WITH COBOL SYNTAX
? FILE NEWSOURCE NAME = TEST2;
? DATA CARDS
  $ MERGE NEW
... PATCH CARDS IF ANY ...
? END
```

If the input file had a file-ID of COBOL/TEST1, in place of the default file-ID of SOURCE the following FILE card should have also been used in the above example.

```
? FILE SOURCE NAME COBOL/TEST1;
```

APPENDIX A

RESERVED WORDS

This appendix lists all the reserved words recognized by the B 1700 COBOL compiler.

ABOUT
ACCEPT
ACCESS
ACTUAL
ADD

ADVANCING
AFTER
ALL
ALL-AT-OPEN
ALPHABETIC

ALTER
ALTERNATE
ALTERNATING
AND
APPLY

ARE
AREA
AREAS
ASCENDING
ASCII

ASSIGN
AT
AUTHOR
AUXILIARY
BACKUP

BATCH-COUNT
BEFORE
BEGINNING
BINARY
BLANK

BLOCK
BY
BZ
CARD96
CASSETTE

CHANNEL
CHARACTERS
CLOCK-UNITS
CLOSE
CMP

CMP-1
CMP-3
CODEFILE
COMMA
COMP

COMP-1
COMP-3
COMPUTATIONAL
COMPUTATIONAL-1
COMPUTATIONAL-3

COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONVERSION

COPY
CORR
CORRESPONDING
CREATE
CRUNCH

CURRENCY
CURRENT
CYLINDER
DATA
DATA-BASE
DATASET

DATE
DATE-COMPILED
DATE-WRITTEN
DB
DDL-NUMBER

DECIMAL-POINT
DECLARATIVES
DELETE
DEMAND
DEPENDING

DESCENDING
DISC
DISK
DISK-DFC1
DISK-DFC2

DISK-DPC1
DISK-DPC2
DISK-HPT
DISKPACK
DISK-PPC2

DISPLAY
DIVIDE
DIVISION
DM-STATUS
DOWN

DUMP
ELSE
ENABLE
END
END-OF-JOB

END-OF-PAGE
END-TRANSIT
ENDING
ENVIRONMENT
EOP

EQUAL
EQUALS
ERROR
EVERY
EXAMINE

EXCEPTION
EXIT
FD
FILE
FILE-CONTROL

FILE-LIMIT
FILE-LIMITS
FILL
FILLER
FIND

FIRST
FLOW
FOR
FORM
FORMAT

FREE
FROM
GIVING
GO
GREATER

HARDWARE-MONITOR
HERE
HIGH-VALUE
HIGH-VALUES
I-O

I-O-CONTROL
ID
IDENTIFICATION
IF
IN

INC-EU
INDEX
INDEXED
INPLACE
INPUT

INPUT-OUTPUT
INQUIRY
INSERT
INSTALLATION
INTERPRET

INTO
INVALID
INVALID-REQUEST
INVOKE
IS

JS
JUST
JUSTIFIED
KEY
LABEL

LAST
LEADING
LEFT
LESS
LIBRARY

LINES
LOCK
LOW-VALUE
LOW-VALUES
MEMORY

MFCU
MICR
MICR-EDIT
MICR-OCR
MOD

MODE
MODIFY
MODULES
MONITOR
MOVE

MULTIPLE
MULTIPLY
NEGATIVE
NEXT
NO

NO-ERRORS
NO-FORMAT
NON-STANDARD
NOT
NOT-READY

NOTE
NULL
NUMERIC
O-I
OBJECT-COMPUTER

OC
OCCURS
OCR
OF
OMITTED

ON
OPEN
OPTIONAL
OR
ORDERING

OTHERWISE
OUTPUT
OUTPUT-INPUT
PACK
PAGE

PC
PERFORM
PIC
PICTURE
POCKET

POCKET-LIGHT
POSITION
POSITIVE
PRINTER
PRINT128

PRIOR
PROCEDURE
PROCEED
PROCESSING
PROCESSOR

PROGRAM-ID
PT-PUNCH
PT-READER
PUNCH
PURGE

Q-EMPTY
Q-FULL
QUEUE
QUEUES
QUOTE
QUOTES

RANDOM
READ
READER
READER-SORTER
RECEIVE

RECORD
RECORDING
RECORDS
RECREATE
REDEFINES

REEL
RELEASE
REMAINDER
REMARKS
REMOTE

REMOVE
RENAMES
REPLACING
RERUN
RESERVE

RETRIEVAL
RETURN
REVERSED
REWIND
RIGHT

ROLLOUT
ROUNDED
RUN
SAME
SAVE

SAVE-FACTOR
SD
SEARCH
SECTION
SECURITY

SEEK
SEGMENT-LIMIT
SELECT
SEND
SENTENCE

SEQUENTIAL
SET
SIGN
SINGLE
SIZE

SORT
SORTER
SOURCE-COMPUTER
SPACE
SPACES

SPECIAL-NAMES
SPO
STACKER
STACKERS
STALEMATE

STANDARD
START-FLOW
STATION
STATIONS
STOP

STOP-FLOW
STORE
STREAM
SUB-QUEUE
SUB-QUEUES

SUBSET
SUBTRACT
SW1
SW2
SW3

SW4
SW5
SW6
SW7
SW8

SY
SYMBOLIC
SYNC
SYNCHRONIZED
TAG

TAG-KEY
TALLY
TALLYING
TAPE
TAPE-MTC1

TAPE-MTC2
TAPE-MTC3
TAPE-MTC4
TAPE-MTC5
TAPE-7

TAPE-9
THAN
THEN
THROUGH
THRU

TIME
TIMES
TO
TODAYS-DATE
TODAYS-NAME

TOP
TRACE
TRANSLATION
UNLOCK
UNTIL

UP
UPDATE
UPON
USAGE
USE

USING
VA
VALUE
VALUES
VARYING

VIA
WHEN
WITH
WORDS
WORK

WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS

ZIP

APPENDIX B

COBOL SYNTAX SUMMARY

IDENTIFICATION DIVISION

```
[MONITOR ...]

 IDENTIFICATION DIVISION.

[PROGRAM-ID.  Any COBOL word.]

[AUTHOR.  Any entry.]

[INSTALLATION.  Any entry.]

[DATE-WRITTEN.  Any entry.]

[DATE-COMPILED.  Any entry - appended with
                 current date and time as
                 maintained by the MCP.]

[SECURITY.  Any entry.]

[REMARKS.  Any entry.  Continuation lines must
           be coded in Area B of the coding form.]
```

MONITOR

```
[MONITOR  [DEPENDING] file-name  ([data-name] ... ;

   [{ALL                          }])  .]
    {procedure-name...}
```

ENVIRONMENT DIVISION

```
 ENVIRONMENT DIVISION.
[[CONFIGURATION SECTION.]

 [SOURCE-COMPUTER ...]

 [OBJECT-COMPUTER ...]

 [SPECIAL-NAMES ...]

 [INPUT-OUTPUT SECTION.]

 [FILE-CONTROL ...]

 [I-O-CONTROL ...]]
```

CONFIGURATION SECTION.

```
Option 1:

SOURCE-COMPUTER.  COPY  library-name
   [, REPLACING word-1 BY word-2
   [, word-3 BY word-4] ...].
```

```
Option 2:

SOURCE-COMPUTER.  {B 1700    }
                  {any entry}  .
```

```
Option 1:

OBJECT-COMPUTER.  COPY  library-name
   [, REPLACING word-1 BY word-2
   [, word-3 BY word-4] ...].
```

```
Option 2:

OBJECT-COMPUTER.  [{B 1700    }]
                  [{any entry}]
   [, [SORT] MEMORY SIZE integer-1 [CHARACTERS]]
   [, DATA SEGMENT-LIMIT IS integer-2 CHARACTERS]
   [, SEGMENT-LIMIT IS priority-number].
```

```
Option 1:

SPECIAL-NAMES.  COPY library-name
[REPLACING word-1 BY word-2
 [, word-3 BY word-4] ...].
```

```
Option 2:

SPECIAL-NAMES.  [CURRENCY SIGN IS literal]
[, implementor-name IS mnemonic-name] ...
[, DECIMAL-POINT IS COMMA].
```

INPUT-OUTPUT SECTION.

Option 1:

FILE-CONTROL. COPY library-name

[REPLACING {word-1 / data-name-1} BY {word-2 / data-name-2 / literal-1}

[, {word-3 / data-name-3} BY {word-4 / data-name-4 / literal-2}] ...].

Option 2:

FILE-CONTROL.

SELECT [OPTIONAL] file-name ASSIGN TO hardware-name-1

[[OR] BACKUP [{TAPE / DISK}]] [FORM] [FOR MULTIPLE REEL] [SINGLE]

[ALL-AT-OPEN] [WORK]

[, RESERVE {NO / integer-1} [ALTERNATE [AREA / AREAS]]]

[, {FILE-LIMIT IS / FILE-LIMITS ARE} {literal-1 / data-name-1} {THRU / THROUGH} {END / literal-2 / data-name-2}

[, {literal-m / data-name-m} {THRU / THROUGH} {literal-n / data-name-n}] ...]

[, ACCESS MODE IS {RANDOM / SEQUENTIAL}]

[, ACTUAL KEY IS data-name-3]

[, PROCESSING MODE IS SEQUENTIAL] . [SELECT] ...

I-O-CONTROL.

```
Option 1:

I-O-CONTROL. COPY library-name
   [REPLACING word-1 BY word-2
   [, word-3 BY word-4] ...].
```

```
Option 2:

I-O-CONTROL.

   [; SAME [RECORD] AREA FOR file-name-2 [, file-name-3] ...]
   [; MULTIPLE FILE {DISKPACK diskpack-id}
                    {TAPE multi-file-id}
        CONTAINS file-name-5 [POSITION integer-2]
        [, file-name-6 [POSITION integer-3]] ...]
   [; APPLY {MICR} {MICR}
            {OCR} [2] [OCR] file-name [file-name] ...] .
```

DATA DIVISION

```
DATA DIVISION.
[FILE-SECTION.
   [{file-description-entry          } [record-description-entry] ...] ...]
    {sort-merge-description-entry    }
[WORKING-STORAGE SECTION.
   [77-level-description-entry]]
    record-description-entry   ]
```

FILE-SECTION.

Option 1:

```
FD file-name COPY library-name
    [ REPLACING { word-1      } BY { word-2      }
                { data-name-1 }    { data-name-2 }
                                   { literal-1   }
      [ , { word-3      } BY { word-4      } ] ... ].
          { data-name-3 }    { data-name-4 }
                             { literal-2   }
```

Option 2:

```
FD file-name [ ; RECORDING MODE IS { ASCII    } ]
                                   { STANDARD }
                                   { NON-STANDARD }

[ ; FILE CONTAINS integer-1 [BY integer-2] { RECORDS  } ]
                                           { STATION  }
                                           { STATIONS }
                                           { QUEUE    }
                                           { QUEUES   }

[ ; BLOCK CONTAINS [integer-3 TO] integer-4 [ RECORDS    ] ]
                                            [ CHARACTERS ]

[ ; RECORD CONTAINS [integer-5 TO] integer-6 CHARACTERS ]

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

[ ; { VA    } OF ID IS { [literal-1/] [literal-2] [/ [literal-3]] }
    { VALUE }          { data-name-3                             }
        [ SAVE-FACTOR IS integer-7 ] ]

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

Option 3:

```
SD sort-file-name COPY library-name
   [ REPLACING { word-1   } BY { word-2   }
               { data-name-1}  { data-name-2}
                               { literal-1  }
   [ , { word-3   } BY { word-4   } ]... ].
       { data-name-3}  { data-name-4}
                       { literal-2  }
```

Option 4:

```
SD sort-file-name
   FILE CONTAINS integer-1 [BY integer-2] RECORDS
   [; RECORD CONTAINS integer-3 CHARACTERS]
   [; BLOCK CONTAINS integer-4 [RECORDS   ]]
                               [CHARACTERS]
   [; DATA { RECORD IS  } data-name-1 [, data-name-2] ...]
           { RECORDS ARE}
```

Option 1:

```
01 data-name-1; COPY library-name
   [ REPLACING { word-1   } BY { word-2   }
               { data-name-2}  { data-name-3}
                               { literal-1  }
       [ , { word-3   } BY { word-4   } ]... ].
           { data-name-4}  { data-name-5}
                           { literal-2  }
```

Option 2:

```
level-number {FILLER       } [; REDEFINES data-name-2]
             {data-name-1  }

   [; {PC     } IS (allowable PICTURE characters)]
      {PIC    }
      {PICTURE}

   [; {BZ                  }]
      {BLANK WHEN ZERO     }

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

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

   [; {OC     } [integer-2 TO] integer-3 TIMES
      {OCCURS }

         [DEPENDING ON data-name-3]

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

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

         [; [USAGE IS] {DISPLAY   }]
                       {CMP       }
                       {CMP-1     }
                       {CMP-3     }
                       {COMP      }
                       {COMP-1    }
                       {COMP-3    }
                       {COMPUTATIONAL  }
                       {COMPUTATIONAL-1}
                       {COMPUTATIONAL-3}
                       {INDEX     }
                       {ASCII     }]

           [; {VA   } [IS ] literal-1 ] .
              {VALUE} [ARE]
```

```
Option 3:
66 data-name-1 RENAMES data-name-2 [{THRU    } data-name-3 ].
                                   {THROUGH }

Option 4:
88 condition-name {VA   }[IS ] literal-1 [{THRU    } literal-2 ].
                  {VALUE}[ARE]           {THROUGH }

   [, literal-3 [{THRU    } literal-4 ]] ... .
                {THROUGH }
```

PROCEDURE DIVISION

```
PROCEDURE DIVISION.
[DECLARATIVES.
 section-name SECTION.  declarative-statement.
 paragraph-name.  [statement.] ...
[paragraph-name.  [statement.] ...]...
[section-name SECTION.  declarative-statement.
 paragraph-name.  [statement.]...
[paragraph-name.  [statement.] ...]...]...
 END DECLARATIVES.]
 [[section-name SECTION [priority-number]].
   paragraph-name.  [statement.] ...
  [[paragraph-name.] ... [statement.] ...]...]...
 [END-OF-JOB.]
```

Verb Formats:

```
ACCEPT identifier [FROM {SPO           }]
                       {mnemonic-name}
```

Option 1:

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

    TO identifier-m [ROUNDED] [identifier-n [ROUNDED] ...]
    [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 2:

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

    GIVING identifier-m [ROUNDED] [, identifier-n [ROUNDED]] ...
    [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 3:

```
ADD {CORR          } identifier-1 TO identifier-2
    {CORRESPONDING}

    [ROUNDED] [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
    [, procedure-name-3 TO [PROCEED TO] procedure-name-4 ...]
```

```
CLOSE file-name-1 [REEL] [WITH {LOCK          }] [, file-name-2...]
                               {PURGE         }
                               {RELEASE       }
                               {NO REWIND     }
                               {REMOVE        }
```

```
COMPUTE identifier-1 [ROUNDED] [, identifier-n [ROUNDED]] ...
        = {identifier-2          }
          {literal-1             }
          {arithmetic-expression }
[; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 1:

```
CONTROL file-name [STOP-FLOW] POCKET {literal    }
                                     {identifier}
```

Option 2:

```
CONTROL file-name {BATCH-COUNT  } identifier
                  {POCKET-LIGHT}
```

Option 1:

```
COPY library-name.
```

Option 2:

```
COPY library-name

  [REPLACING {word-1      } BY {word-2      }
             {data-name-1}     {identifier-1}
                               {literal-1    }

       [, {word-3      } BY {word-4      }  ...] .]
          {data-name-3}     {identifier-2}
                            {literal-2    }
```

```
DISPLAY {literal-1   } [, {literal-2   }  ...]
        {identifier-1}    {identifier-2}

  [UPON {SPO          }]
        {mnemonic-name}
```

Option 1:

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

    [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 2:

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

   GIVING identifier-3 [ROUNDED]
    [REMAINDER identifier-5 [ROUNDED]]
    [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

```
DUMP [file-name]
```

Option 1:

```
EXAMINE identifier-1 TALLING {ALL      } {literal-1   }
                             {LEADING  }
                             {[UNTIL] FIRST} {identifier-2}

   [REPLACING BY {literal-2   }]
                 {identifier-3}
```

Option 2:

```
EXAMINE identifier-1 REPLACING {ALL      } {literal-1   }
                               {LEADING  }
                               {[UNTIL] FIRST} {identifier-2}

   BY {literal-2   }
      {identifier-3}
```

```
EXIT.
```

```
FORMAT identifier-1 INTO identifier-2 ON SIZE ERROR statement
```

Option 1:

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

Option 2:

```
GO TO procedure-name-1 [, procedure-name-2] ...

   , procedure-name-n DEPENDING ON identifier
```

```
IF condition-1; {sentence-1   } [; ELSE {sentence-2   }]
                {NEXT SENTENCE}        {NEXT SENTENCE}
```

```
MICR-EDIT identifier-1 INTO identifier-2
```

Option 1:

```
MOVE {identifier-1} TO identifier-2 [, identifier-3] ...
     {literal-1   }
```

Option 2:

```
MOVE {CORR          } identifier-1 TO identifier-2
     {CORRESPONDING}
```

```
MULTIPLY {literal-1   } BY {literal-2   }
         {identifier-1}    {identifier-2}
    [GIVING identifier-3] [ROUNDED]
    [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 1 Paragraph NOTE:

```
Paragraph-name. NOTE any comment.
```

Option 2 Paragraph NOTE:

```
NOTE. Any comment.
```

Option 3 Sentence NOTE:

```
NOTE. Any comment.
```

Option 1:

```
OPEN
{ [INPUT file-name-1 [{WITH LOCK [ACCESS]}] [file-name-2...]] ...
                      {REVERSED          }
                      {WITH NO REWIND    }
  [OUTPUT file-name-3 [WITH NO REWIND] [file-name-4...]] ...
  [{INPUT-OUTPUT} file-name-5 [file-name-6 ...]] ...
  {I-O          }
  [O-I file-name-7 [file-name-8...]] ...                  } ...
```

Option 2:

```
OPEN {OUTPUT      }
     {I-O         } file-name
     {INPUT-OUTPUT}
        [WITH PUNCH] [WITH {PRINT128 }] [WITH STACKERS]
                           {INTERPRET}
```

Option 1:

```
PERFORM procedure-name-1 [{THRU   } procedure-name-2]
                          {THROUGH}
```

Option 2:

```
PERFORM procedure-name-1 [{THRU    } procedure-name-2]
                         {THROUGH}

  {integer-1   } TIMES
  {identifier-10}
```

Option 3:

```
PERFORM procedure-name-1 [{THRU    } procedure-name-2]
                         {THROUGH}

   VARYING {index-name-1} FROM {index-name-2} BY
           {identifier-1}      {identifier-2}
                               {literal-2   }

  {identifier-3} UNTIL condition-1 [AFTER {index-name-4}
  {literal-3   }                         {identifier-4}

   FROM {index-name-5} BY {index-name-6}
        {identifier-5}    {literal-6   }
        {literal-5   }

   UNTIL condition-2] [AFTER {index-name-7} FROM
                             {identifier-7}

  {index-name-8} BY {identifier-9} UNTIL condition-3]
  {identifier-8}    {literal-9   }
  {literal-8   }
```

Option 1:

```
READ file-name [INTO identifier]
```

Option 2:

```
READ file-name RECORD [INTO identifier]

  [; {AT END    } statement-1 [; ELSE statement-2]]
     {INVALID KEY}
```

```
RELEASE record-name [FROM identifier]
```

```
RETURN file-name RECORD [INTO identifier]

   ; AT END statement-1 [; ELSE statement-2]
```

Option 1:

```
SEARCH identifier-1 [VARYING {index-name-1  }]
                             {identifier-2  }

   [; AT END imperative-statement-1]

    ; WHEN condition-1 {imperative-statement-2}
                      {NEXT SENTENCE         }

   [; WHEN condition-2 {imperative-statement-3}]
                       {NEXT SENTENCE         }
```

Option 2:

```
SEARCH ALL identifier-3 [; AT END imperative-statement-4]

    ; WHEN condition-3 {imperative-statement-5}
                      {NEXT SENTENCE         }
```

```
SEEK file-name RECORD [WITH KEY CONVERSION]
```

Option 1:

```
SET {index-name-1 } [, {index-name-2 } ...] TO {index-name-3 }
    {identifier-1 }    {identifier-2 }         {identifier-3 }
                                               {literal-1    }
```

Option 2:

```
SET index-name-4 [, index-name-5 ...] {UP BY   } {identifier-4}
                                      {DOWN BY } {literal-2   }
```

```
SORT [TAG-KEY | INPLACE] file-name

     [{PURGE | RUN | END} ON ERROR]

  ON {DESCENDING | ASCENDING} KEY data-name-1 [, data-name-2] ...

 [ON {DESCENDING | ASCENDING} KEY data-name-3 [, data-name-4] ... ] ...

 { INPUT PROCEDURE IS section-name-1 [{THRU | THROUGH} section-name-2] }
 { USING file-name-2 [LOCK | PURGE | RELEASE]                          }

 { OUTPUT PROCEDURE IS section-name-3 [{THRU | THROUGH} section-name-4] }
 { GIVING file-name-3 [LOCK | RELEASE]                                  }
```

```
STOP {RUN | literal}
```

Option 1:

```
SUBTRACT {literal-1 | identifier-1} [, {literal-2 | identifier-2} ...] FROM
   identifier-m [ROUNDED] [, identifier-n [ROUNDED] ...]
  [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 2:

```
SUBTRACT {literal-1 | identifier-1} [, {literal-2 | identifier-2} ...] FROM {literal-m | identifier-m}
   GIVING identifier-n [ROUNDED] [, identifier-o [ROUNDED]] ...
  [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

Option 3:

```
SUBTRACT {CORR|CORRESPONDING} identifier-1 FROM identifier-2
    [ROUNDED] [; ON SIZE ERROR statement-1 [; ELSE statement-2]]
```

```
TRACE 20
```

Option 1:

```
USE AFTER STANDARD ERROR PROCEDURE ON { file-name-1 [, file-name-2] ...
                                      | INPUT
                                      | OUTPUT
                                      | INPUT-OUTPUT
                                      | I-O
                                      | O-I                          }.
```

Option 2:

```
USE {AFTER|BEFORE} STANDARD {BEGINNING|ENDING}
    [{REEL|FILE}] LABEL PROCEDURE ON {file-name-1 [, file-name-2] ...
                                     | INPUT
                                     | OUTPUT                        }.
```

Option 3:

```
USE FOR KEY CONVERSION ON file-name-1 [, file-name-2...] .
```

Option 4:

```
USE FOR READER-SORTER POCKET file-name.
```

Option 1:

```
WRITE record-name [FROM identifier-1]
    [{AFTER|BEFORE} ADVANCING {{integer-1|identifier-2} LINES
                              | TO CHANNEL {integer-2|identifier-3}}]
    [; TO {ERROR|AUXILIARY|STACKER} {literal-1|identifier-1}]
    [; AT {END-OF-PAGE|EOP} imperative-statement]
```

```
Option 2:

WRITE record-name [FROM identifier-1]
    [; INVALID KEY any statement [; ELSE any statement]]
```

```
ZIP data-name
```

APPENDIX C

COMPILER ERROR MESSAGES

ERROR NO.	MESSAGE
000	FILE-NAME EXPECTED
001	INTEGER LITERAL REQUIRED
002	INVALID LITERAL
003	RESERVED WORD REQUIRED
004	PARAGRAPH HEADER EXPECTED IN AREA A (COL. 8-11)
005	MISSING DIVISION
006	DOLLAR CARD ERROR
007	"DIVISION" REQUIRED
008	COMPILER ERROR
009	MISSING PERIOD
010	RESERVED WORD OR DATA NAME REQUIRED
011	COPY REPLACING OR MNEMONIC LIST OVERFLOW
012	DUPLICATE MNEMONIC NAME
013	UNIDENTIFIED ITEM
014	IMPROPER LABEL RECORD(S) DECLARATION
015	ILLEGAL NESTED COPY
016	ILLEGAL COPY OPERAND
017	STANDARD OR NON-STANDARD OR ASCII REQUIRED
018	DUPLICATE REPLACING
019	ILLEGAL SUBSCRIPTING
020	ILLEGAL LIBRARY NAME
021	ILLEGAL TYPE
022	ILLEGAL QUALIFICATION
023	ILLEGAL PROGRAM ID
024	"SECTION" REQUIRED
025	MISSING FILE NAME
026	A PARENTHESIS WAS EXPECTED HERE
027	MISSING LABEL QUALIFICATION...MONITOR
028	NO FD OR SD
029	INVALID FD
030	ILLEGAL LEVEL
031	ILLEGAL DATA NAME
032	RELATIONAL OPERATOR REQUIRED
033	PICTURE SIZE ERROR
034	"PROCEDURE" EXPECTED
035	ILLEGAL FD OR SD IN WORKING-STORAGE
036	PRIORITY NUMBER ERROR
037	MISSING IMPLIED LABEL OR LABEL QUALIFICATION
038	MISSING SECTION
039	NO "USE"
040	PARAGRAPH-NAME OR SECTION-NAME REQUIRED
041	VERB OR PARAGRAPH-NAME OR SECTION-NAME REQUIRED
042	MISSING FILE NAME
043	ILLEGAL LABEL RECORD REFERENCE OUTSIDE DECLARATIVES
044	ILLEGAL ARITHMETIC OPERAND
045	MISSING "=" OR "FROM"
046	NO VALID CORRESPONDING OPERANDS

ERROR NO.	MESSAGE
047	COMPOSITE ARITHMETIC SIZE > 125...MAY USE LARGE AMOUNT OF CORE
048	MISSING "END DECLARATIVES"
049	MISSING "DECLARATIVES"
050	ILLEGAL MOVE OPERAND
051	"TO" REQUIRED
052	AN ALPHABETIC ITEM CANNOT BE MOVED TO A NUMERIC ITEM
053	ILLEGAL GROUP TO ELEMENTARY MOVE
054	ILLEGAL "ALL" LITERAL
055	ILLEGAL SUBSCRIPTING OF A SUBSCRIPT
056	SUBSCRIPT NOT S-SIGN OR UNSIGNED
057	SUBSCRIPT NOT NUMERIC INTEGER
058	SUBSCRIPT NOT ELEMENTARY ITEM
059	ILLEGAL MIXING OF INDEX AND SUBSCRIPT
060	EXPLICIT DATA NAME TABLE OVERFLOW
061	THIS DATA NAME IS NOT DESCRIBED IN THE DATA DIVISION
062	QUALIFIER ARRAY TABLE OVERFLOW
063	ILLEGAL QUALIFIER
064	INSUFFICIENT QUALIFICATION
065	OVERLAPPING CORRESPONDING OPERANDS
066	NO MATCHING CORRESPONDING OPERANDS
067	CORRESPONDING NAMES ARE THE SAME
068	FD NAME ILLEGAL FOR CORRESPONDING
069	CORRESPONDING DATA NAME NOT GROUP ITEM
070	DUPLICATE PARAGRAPH OR SECTION NAME
071	LABEL NOT UNIQUE
072	LABEL QUALIFICATION NOT A SECTION
073	ALTER TABLE OVERFLOW
074	QUALIFIER LABEL TABLE OVERFLOW
075	REFERENCED PARAGRAPH OR SECTION DOES NOT EXIST
076	LABEL QUALIFIER IS NOT UNIQUE
077	LABEL RECORD IS NOT AN 01 LEVEL
078	ILLEGAL CONDITIONAL STATEMENT
079	ILLEGAL DOUBLE NEGATIVE
080	INVALID IMPLIED SUBJECT OR MISSING RELATIONAL OPERATOR
081	PICTURE TABLE FULL : RECOMPILE
082	PICTURE SPECIFIED ON A GROUP ITEM
083	RENAMES OPERAND OUT OF RANGE
084	RENAMES OPERAND LEVEL CANNOT BE 01 OR 66 OR 77 OR 88
085	RENAMES OPERAND IS SUBSCRIPTED
086	DUPLICATE NAME
087	"RENAMES" REQUIRED
088	GROUP RENAMES ITEM ADDRESS OR LENGTH NOT 0 MOD 2
089	BLANK WHEN ZERO SPECIFIED FOR NON NUMERIC CLASS
090	JUSTIFIED SPECIFIED FOR NUMERIC OR EDITED NUMERIC CLASS
091	UNSIGNED INTEGER EXPECTED
092	"OCCURS" SPECIFIED FOR LEVEL 01 OR 77
093	VARIABLE LENGTH DISK FILE MUST HAVE SEQUENTIAL ACCESS AND NO FILE LIMITS
094	NON-ZERO VALUE EXPECTED
095	DUPLICATE "VALUE" CLAUSE
096	ILLEGAL "VALUE" LITERAL
097	DATA CLAUSE EXPECTED
098	ILLEGAL DATA CLAUSE FOR GROUP ITEM
099	ILLEGAL 4 BIT SPECIFICATION FOR HARDWARE DEVICE
100	"ZERO" EXPECTED

ERROR NO.	MESSAGE
101	ASCII MAY BE SPECIFIED ON ONLY WORKING-STORAGE LEVEL 01 OR 77
102	MISSING "OCCURS" FOR INDEX-NAME
103	INDEX NAME EXPECTED
104	LEVEL NUMBER EXPECTED
105	LEVEL NOT 01 THRU 49, 66, 77, OR 88
106	LEVEL 77 MUST FOLLOW ONLY WORKING-STORAGE SECTION
107	PICTURE REQUIRED FOR ELEMENTARY DATA NAME
108	NO DATA CLAUSE FOR INDEX DATA ITEM
109	COMPUTATIONAL ITEM NOT NUMERIC
110	COMPUTATIONAL SIGN NOT S OR J
111	IMPROPER REDEFINED NAME
112	LEVEL NUMBER NEQ REDEFINED LEVEL NUMBER
113	REDEFINED OPERAND IS SUBSCRIPTED
114	REDEFINED GROUP ADDRESS IS ODD
115	VALUE CANNOT BE SPECIFIED FOR SUBSCRIPTED ITEM
116	VALUE CANNOT BE SPECIFIED FOR REDEFINED AREA
117	VALUE CONFLICTS WITH GROUP VALUE
118	FILLER ADDED TO PREVIOUS ITEM
119	REDEFINED AREA NEQ REDEFINING AREA
120	USAGE CONFLICTS WITH GROUP USAGE
121	SUBSCRIPT MAXIMUM IS 3
122	INCONSISTENT LEVEL NUMBER
123	01 LEVEL NUMBER EXPECTED
124	"SELECT" EXPECTED
125	FILE PREVIOUSLY SELECTED
126	FILE NOT SELECTED
127	FILE INFO TABLE FULL
128	HARDWARE NAME EXPECTED
129	SD FILE NOT ASSIGNED TO DISK
130	UNIDENTIFIED WORD
131	WORD EXCEEDS 30 CHARACTERS
132	INVALID NUMERIC OR UNDIGIT LITERAL
133	ZERO SIZE LITERAL
134	MISSING RIGHT QUOTE
135	WORD ENDS IN A HYPHEN
136	NO ALPHA CHARACTER IN NAME
137	MISSING LEFT QUOTE
138	TABLE OVERFLOW IN MERGE: RECOMPILE
139	MISSING "BY" OR "INTO"
140	MISSING "GIVING"
141	"MOD" AND "REMAINDER" ARE MUTUALLY EXCLUSIVE
142	MISSING "ERROR"
143	ILLEGAL PERFORM OPERAND
144	ILLEGAL PERFORM "TIMES" OPERAND OR MISSING "TIMES"
145	MISSING "UNTIL"
146	ILLEGAL PERFORM "VARYING" OPERAND
147	ILLEGAL PERFORM "FROM" OPERAND OR MISSING "FROM"
148	ILLEGAL PERFORM "BY" OPERAND OR MISSING "BY"
149	ILLEGAL SET OPERAND
150	FILE-LIMITS SPECIFICATION GIVEN FOR SD FILE
151	ACTUAL KEY MUST BE PC 9(8) COMP (DISK/QUEUE),PC X(112) (SORTER),PC 9(10) (REMOTE)
152	PICTURE REPEAT ERROR
153	PICTURE FLOAT ERROR
154	PICTURE SIGN ERROR
155	PICTURE "P" SPECIFICATION ERROR

ERROR NO.	MESSAGE
156	PICTURE SIZE SPECIFICATION ERROR
157	PICTURE DECIMAL POINT ERROR
158	PICTURE ERROR...IMPROPER CHARACTER PRECEDING FLOAT,ZERO SUPPRESS,OR CHECK PROTECT
159	PICTURE CLASS ERROR
160	PICTURE MASK SIZE (100) EXCEEDED
161	VALUE OF ID CATEGORY IS NUMERIC
162	FILE-LIMIT MUST BE PC 9(8) COMP
163	"THRU" EXPECTED
164	DATA NAME OR INTEGER EXPECTED
165	"RANDOM" OR "SEQUENTIAL" EXPECTED
166	SD FILE MUST BE SEQUENTIAL
167	ACTUAL KEY REQUIRED
168	"BACKUP" EXPECTED
169	FILE-CONTROL CLAUSE EXPECTED
170	BACKUP FOR LINE PRINTER OR PUNCH ONLY
171	ILLEGAL USE OF FILE-NAME OR CONDITION-NAME
172	"MULTI-FILE-ID" EXPECTED
173	FILE NOT ASSIGNED TO TAPE OR DISK
174	I-O CONTROL CLAUSE OR "." EXPECTED
175	APPLY CLAUSE NOT IMPLEMENTED
176	FILE NOT ASSIGNED TO HARDWARE DEVICE
177	DECLARATIVES NOT 1ST ITEM IN PRO.DIV. OR USE NOT BETWEEN SECTION & PARAGRAPH
178	CONDITION-NAME LITERAL REQUIRED
179	"VALUE" REQUIRED
180	VALUE THRU...1ST LITERAL GEQ 2ND LITERAL
181	I/O OPERAND MUST BE 01 RECORD OF A FILE
182	I/O OPERAND CANNOT BE LABEL RECORD
183	I/O OPERAND MUST BE SORT FILE
184	I/O OPERAND MUST BE FILE
185	I/O OPERAND CANNOT BE WORKING-STORAGE 01 RECORD
186	I/O OPERAND CANNOT BE SORT FILE
187	CAN MONITOR ONLY ON FILE
188	SORT KEY NOT WITHIN SCOPE OF SORT FILE
189	SORT STATEMENT NOT PERMITTED IN DECLARATIVES SECTION
190	MISSING "ASCENDING" OR "DESCENDING"
191	SORT KEY CANNOT BE SUBSCRIPTED
192	MISSING SORT KEY
193	NUMBER OF SORT KEYS GREATER THAN 40
194	SIZE OF KEY TOO LARGE
195	MISSING "INPUT"/"OUTPUT"/"USING"/"GIVING"
196	USING/GIVING FILE REC SIZE NEQ SORT REC SIZE
197	MISSING SORT INPUT/OUTPUT PROCEDURE NAME
198	SORT THRU PROCEDURE APPEARS BEFORE BEGINNING POINT
199	GO TO DEPENDING OPERAND MUST BE ELEMENTARY DATA-NAME INTEGER
200	MORE THAN 1 ACCEPT OPERAND
201	MISSING WORD OR LITERAL
202	NOT READABLE/WRITEABLE HARDWARE
203	GO TO DEPENDING LABEL LIMIT (1022) EXCEEDED
204	FILE-LIMITS REQUIRE "AT END" OR "INVALID KEY"
205	ILLEGAL READER-SORTER OPERAND
206	MISSING "DEPENDING" IN GO TO
207	READ/WRITE/SEEK ON SD OR RELEASE/RETURN ON FD
208	SEEK FILE NOT RANDOM DISK
209	ILLEGAL ADVANCING/STACKER OPERAND
210	INVALID I/O OPERAND

ERROR NO.	MESSAGE
211	MINUS SIGN NOT ALLOWED
212	MISSING GO TO LABEL
213	GO TO MUST BE TERMINATED BY "." OR "ELSE"
214	ILLEGAL OPTION FOR I/O DEVICE
215	INVALID OR MISSING OPEN TYPE
216	ATTEMPTED ALTER OF NON-GOTO PROCEDURE-NAME
217	CHANNEL NUMBER GTR 11
218	RESERVED WORD (VERB) REQUIRED
219	CURRENT SECTION MUST HAVE SAME PRIORITY AS REFERENCED GOTO PROC
220	TAPE FILE CANNOT HAVE 2 NAMES
221	EXPECTED A FILE DECLARATION CLAUSE
222	MISSING FILE CONTAINS
223	RECORDS PER AREA MADE MULTIPLE OF RECORDS PER BLOCK
224	DATA DICTIONARY FULL : RECOMPILE
225	REDEFINES NOT ALLOWED ON 01 RECORD OF FILE
226	MISSING FILE RECORD DESCRIPTION
227	BLOCK SIZE NOT MULTIPLE OF MAXIMUM RECORD SIZE
228	ARITHMETIC OPERAND MUST BE ELEMENTARY ITEM
229	ARITHMETIC OPERAND CANNOT BE INDEX ITEM
230	ARITHMETIC OPERAND MUST BE NUMERIC
231	ARITHMETIC LITERAL OPERAND MUST BE NUMERIC
232	FILE LABEL RECORDS OMITTED
233	DUPLICATE USE PROCEDURE
234	MONITOR ALLOWED ONLY ON FILE
235	CANNOT MONITOR ON SORT FILE
236	MONITOR ALLOWED ON LINE PRINTER ONLY
237	OBJECT OF SEARCH MUST BE INDEX
238	MISSING ALTER LABEL
239	DATA-NAME REQUIRED
240	DATA-NAME OR INDEX-NAME REQUIRED
241	WHEN CLAUSE REQUIRED
242	ILLEGAL USE OF RESERVED WORD
243	ITEM NOT DISPLAY
244	DATA LENGTH EXCEEDS 1 CHARACTER
245	ILLEGAL USE OF FIRST IN EXAMINE
246	MISSING FIRST IN EXAMINE
247	SIGN CONDITION OPERAND MUST BE ELEMENTARY NUMERIC OR ARITHMETIC EXPRESSION
248	CLASS TEST OPERAND CANNOT BE ARITHMETIC EXPRESSION
249	CLASS TEST OPERAND MUST BE DISPLAY
250	NUMERIC VERSUS ALPHA COMPARE IS ILLEGAL
251	VIOLATION OF ANSI RULES FOR PERFORMING OVERLAYABLE SEGMENTS
252	RECEIVING FIELD TRUNCATION
253	FIELD TREATED AS 8 BIT DISPLAY
254	SEQUENCE ERROR
255	LITERAL EXCEEDS 160 CHARACTERS
256	CLASS TEST OPERAND CANNOT BE ASCII
257	MISSING SUBSCRIPT
258	DUPLICATE CONDITION SEND/RECEIVE
259	COMPARISON OF INDEX DATA ITEM MUST BE AGAINST INDEX DATA ITEM OR INDEX NAME
260	COMPARISON OPERANDS MUST HAVE SAME USAGE
261	ILLEGAL USE OF "NEXT SENTENCE"
262	CANNOT COMPARE LITERALS
263	CANNOT COMPARE INDEX-NAME VS ZERO OR - LITERAL
264	SORTER FILE RECORD NOT MOD 112 IN LENGTH

ERROR NO.	MESSAGE
265	CANNOT BLOCK SORTER FILE
266	GROUP NAME CANNOT BE "FILLER"
267	IF STATEMENT MUST BE TERMINATED BY "." OR "WHEN"
268	LITERAL SUBSCRIPT CAUSES OUT OF BOUNDS ERROR
269	HARDWARE MUST BE READER-SORTER
270	READER-SORTER ACTUAL KEY NOT IN DATA SEGMENT ZERO
271	RECORDING MODE BINARY NOT ALLOWED
272	MISSING MONITOR DECLARATION (TO DECLARE DUMP FILE)
273	MULTI RECEIVING FIELDS ILLEGAL WITH CORRESPONDING OPTION
274	TAG-KEY GIVING FILE MUST HAVE 01 RECORD OF PC 9(8) CMP
275	TAG-KEY SORT REQUIRES USING FILE AND GIVING FILE OPTIONS
276	COMPILER ERROR IN CODEGEN GET [(<←&$*) ;-/,%=]#@:>+≠**←<*>$
277	COMPILER ERROR IN CODGEN GET.TRASH [(<←&$*) ;-/,%=]#@:>+≠**& $**←<*>$
278	COMPILER ERROR IN CODEGEN GET.POOL [(<←&$*) ;-/,%=]#@:>+≠** &$**←<*>$
279	COMPILER ERROR IN CODEGEN CONTROL [(<←&$*) ;-/,%=]#@:>+≠** &$**←<*>$
280	COMPILER ERROR IN CODEGEN ARITH.EXP [(<←&$*) ;-/,%=]#@:>+≠ **&$**←<*>$
281	COMPILER ERROR IN CODEGEN OPND.OVER [(<←&$*) ;-/,%=]#@:>+≠ **&$**←<*>$
282	COMPILER ERROR IN PROSYN GET [(<←&$*) ;-/,%=]#@:>+≠**&$**←<*>$
283	USAGE DECLARED FOR AN ASCII FILE
284	BLOCK SIZE MADE EQUAL TO MAXIMUM RECORD SIZE
285	CURRENT COMPILER DATA SEGMENT LIMIT OF 000 EXCEEDED.. TEMPORARY SOLUTION=RESEGMENT
286	COMPILER ERROR..COLUMN TOO LARGE..DOES NOT AFFECT COMPILATION [(<←&$*) ;-/,%=]#@:>+≠**&$**←<*>$
287	A ROUTINE IN CODEGEN HAS ENCOUNTERED AN UNEXPECTED TOKEN.. COMPILER ERROR [(<←&$*) ;-/,%=]#@:>+≠**&$**←<*>$
288	CANNOT COMPARE ALPHA VS REAL
289	MINIMUM OF 2 OPERANDS MUST PRECEDE THE WORD GIVING IN AN ADD
290	SUBSCRIPT NOT INTEGER (SPACE REQUIRED BEFORE LIT IF DECIMAL-POINT IS COMMA
291	INVOKED DATASET NAME IS NOT A WORD
292	EXPECTED "DB" AS A LEVEL INDICATOR
293	INVALID DB NAME
294	EXPECTED A DDL-NUMBER
295	DATA-BASE DECLARATION NOT EXPECTED IN THIS SECTION
296	"DATASET" EXPECTED
297	WORKING-STORAGE SECTION OR PROCEDURE DIVISION EXPECTED
298	DATA MANAGEMENT LEVEL 01 DATASET NAME REQUIRED
299	DATA BASE NAME REQUIRED

INDEX

cut along dotted line

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: B 1700 SYSTEMS
COBOL Reference Manual

FORM: 1057197
DATE: March, 1975

CHECK TYPE OF SUGGESTION:

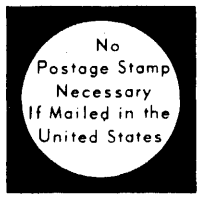
☐ ADDITION ☐ DELETION ☐ REVISION ☐ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

FROM: NAME _____ DATE _____
TITLE _____
COMPANY _____
ADDRESS _____
_____

STAPLE

FOLD DOWN SECOND FOLD DOWN



BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232

Burroughs Corporation
Burroughs Place
Detroit, Michigan 48232

attn: Systems Documentation
Technical Information Organization, TIO-Central

FOLD UP FIRST FOLD UP

Printed in U.S.A March, 1975 1057197