Burroughs

Reference Manual

B 20 Systems COBOL II

(Relative to Release Level 4.0)

Distribution Code SA

Priced Item
Printed in U.S.A.
May 1985

1180122

Burroughs

Reference
Manual

B 20 Systems COBOL II

(Relative to Release Level 4.0)

Distribution Code SA

Priced Item
Printed in U.S.A.
May 1985

1180122

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v	Origianl
vi	Blank
vii thru xvii	Original
xviii	Blank
1-1 thru 1-4	Original
2-1 thru 2-33	Original
2-34	Blank
3-1 thru 3-103	Original
3-104	Blank
4-1 thru 4-12	Original
5-1 thru 5-27	Original
5-28	Blank
6-1 thru 6-28	Original
7-1 thru 7-31	Original
7-32	Blank
8-1 thru 8-16	Original
9-1 thru 9-5	Original
9-6	Blank
10-1, 10-2	Original
11-1 thru 11-7	Original
11-8	Blank
12-1 thru 12-7	Original
12-8	Blank
13-1 thru 13-3	Original
13-4	Blank
A-1 thru A-3	Origianl
A-4	Blank
B-1	Original
B-2	Blank
C-1 thru C-26	Original
D-1 thru D-3	Original
D-4	Blank
E-1 thru E-5	Original
E-6	Blank
F-1 thru F-18	Original
G-1 thru G-3	Original
G-4	Blank
H-1, H-2	Original
I-1 thru I-5	Original
I-6	Blank
J-1 thru J-53	Original
J-54	Blank
K-1 thru K-12	Original
1 thru 18	Original

PREFACE

NOTATION TO THIS MANUAL

Throughout this manual, the following notation is used to describe the format of COBOL statements:

1. All words printed in capital letters which are underlined must always be present when the functions of which they are a part are used. An error printout will occur during compilation if the underlined words are absent or incorrectly spelled. The underlining is not necessary when writing a COBOL source program.

2. All words printed in capital letters which are not underlined are used for readability only. They may be written, or not, as the programmer wishes.

3. All words printed in small letters are generic terms representing names which will be devised by the programmer.

4. When material is enclosed in square brackets [ ], it is an indication that the material is an option which may be included or omitted as required.

5. Language features that are noted in the text are language extensions which exceed the ANSI standard.

6. In text, the ellipsis (...) shows the omission of a portion of a source program or a sequence. This meaning becomes apparent in context.

   In the general formats, the ellipsis represents the position at which repetition may occur at the user's option. The portion of the format that may be repeated is determined as follows:

   Given ... in a clause or statement format, scanning right to left, determine the or [immediately to the left of the ...; continue scanning right to left and determine the logically matching or]; the ... applies to the words between the determined pair of delimiters.

7. The term identifier means either a data-name or a subscripted data-name. An identifier takes the following form:

   $$\text{data-name-1} \quad \left[(\quad \left\{ \begin{array}{l} \text{data-name-2} \\ \text{literal-1} \end{array} \right\} \quad) \right]$$

   data-name-2 or literal-1 must be a positive integer in the range 1 to the number of elements in the table.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

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

SECTION 1

INTRODUCTION

WHAT IS L/II COBOL?

COBOL (COmmon Business Oriented Language) is the most widely and extensively used language for the programming of commercial and administrative data processing.

L/II COBOL is based on the ANSI COBOL as specified in "American National Standard Programming Language COBOL" (ANSI X3.23 1974). The L/II COBOL implementation has been selected from both levels of ANSI COBOL. The following modules are fully implemented at Level 1:

- Nucleus
- Table Handling
- Sequential Input and Output
- Relative Input and Output
- Indexed Input and Output
- Segmentation
- Library
- Inter-Program Communication
- Debug

In addition the following modules are fully implemented at Level 2:

- Nucleus
- Table Handling
- Sequential Input and Output
- Relative Input and Output
- Indexed Input and Output
- Inter-Program Communication
- Sort Merge

The Communications module is accepted by the compiler for syntax checking only. The Run-Time System will not execute this module if included in a program.

This manual is intended as a reference work for L/II COBOL programmers and material from the ANSI COBOL language standard document is included.

Along with the ANSI implementation L/II COBOL also contains several language extensions specifically oriented to the small computer environment and for compatibility with some larger mainframe applications. These enable a L/II COBOL program to format CRT screens for data input and output (DISPLAY and ACCEPT), READ and WRITE text files efficiently and define external file names at run time.

The programmer wishing to transport an existing COBOL program to run under L/II COBOL must check that the individual language features he has used are supported by L/II COBOL. The COBOL SECTION statements in the Segmentation feature can be performed using the PERFORM statement.

Burroughs COBOL programs are created using a conventional text editor. The Compiler compiles the programs and produces intermediate code, which is executed by the Run-Time System. A listing of the Burroughs COBOL program is provided by the Compiler during compilation. Error messages are inserted in the listing. Interactive Debugging facilities are provided for run-time use, and these are described in Appendix J.

PROGRAM STRUCTURE

A COBOL program consists of four divisions:

1. IDENTIFICATION DIVISION - An identification of the program
2. ENVIRONMENT DIVISION - A description of the equipment to be used to compile and run the program
3. DATA DIVISION - A description of the data to be processed
4. PROCEDURE DIVISION - A set of procedures to specify the operations to be performed on the data

Each division is divided into sections which are further divided into paragraphs which in turn are made up of sentences.

Within these subdivisions of a COBOL program, further subdivisions exists as clauses and statements. A clause is an ordered set of COBOL elements that specify an attribute of an entry, and a statement is a combination of elements in the Procedure Division that include a COBOL verb and constitute a program instruction.

FORMATS AND RULES

GENERAL FORMAT

A general format is the specific arrangement of the elements of a clause or a statement. Throughout this document a format is shown adjacent to information defining the clause or statement. When more than one specific arrangement is permitted, the general format is separated into numbered formats. Clauses must be written in the sequence given in the general formats. (Clauses that are optional must appear in the sequence shown if they are used). In certain cases, stated explicitly in the rules associated with a given format, the clauses may appear in sequences other than that shown. Applications, requirements or restrictions are shown as rules.

SYNTAX RULES

Syntax rules are those rules that define or clarify the order in which words or elements are arranged to form larger elements such as phrases, clauses, or statements. Syntax rules also impose restrictions on individual words or elements.

These rules are used to define or clarify how the statement must be written, i.e., the order of the elements of the statement and restrictions on what each element may represent.

GENERAL RULES

A general rule is a rule that defines or clarifies the meaning or relationship of meanings of an element or set of elements. It is used to define or clarify the semantics of the statement and the effect that it has on either execution or compilation.

ELEMENTS

Elements which make up a clause or a statement consist of uppercase words, lowercase words, level-numbers, brackets, braces, connectives and special characters (See Section 2).

SOURCE FORMAT

The COBOL source format divides each COBOL source record into 72 columns. These columns are used in the following way:

Columns 1 - 6	Sequence number
Column 7	Indicator area
Column 8 - 11	Area A
Columns 12 - 72	Area B

SEQUENCE NUMBER

A sequence number of six digits may be used to identify each source program line.

INDICATOR AREA

An asterisk * in this area marks the line as documentary comment only. Such a comment line can appear anywhere in the program after the Identification Division header. Any characters from the ASCII character set can be included in Area A and Area B of the line.

A stroke /, in the indicator area acts as a comment line above but causes the page to eject before printing the comment.

A "D" in the indicator area represents a debugging line. Areas A and B may contain any valid COBOL sentence.

A "-" in the indicator area represents the continuation of a nonnumeric literal. The first non-blank character in Area B of the continuation line must be a quotation mark. The literal continues with the first character after the quotation mark. All spaces at the end of the continued line are significant.

Section names and paragraph names begin in Area A and are followed by a period and a space. Level indications FD, 01 and 66, 77 and 88 begin in Area A and are followed in Area B by the appropriate file and record description.

Program sentences may commence anywhere in Area A and Area B. More than one sentence is permitted in each source record.

SECTION 2

COBOL CONCEPTS

LANGUAGE CONCEPTS

CHARACTER SET

The most basic and indivisible unit of the language is the character. The set of characters used to form L/II COBOL character-strings and separators includes the letters of the alphabet, digits and special characters. The character set consists of the characters defined below:

0 to 9
A to Z
a to z (Reserved and User Word Characters
  read as: A to Z)
Space

+	Plus sign
-	Minus sign or hyphen
*	Asterisk
/	Oblique Stroke/Slash
=	Equal sign
$	Dollar sign
.	Full stop or decimal point
,	Comma or decimal point
;	Semicolon
"	Quotation mark
(	Left Parenthesis
)	Right Parenthesis
>	Greater than symbol
<	Less than symbol

The L/II COBOL language is restricted to the above character set, but the content of non-numeric literals, comment lines and data may include any of the characters from the ASCII character set. (See Appendix B).

LANGUAGE STRUCTURE

The individual characters of the language are concatenated to form character-strings and separators. A separator may be concatenated with another separator or with a character-string. A character-string may only be concatenated with a separator. The concatenation of character-strings and separators forms the text of a source program.

Separators

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

1. The punctuation character space is a separator. Anywhere a space is used as a separator, more than one space may be used.

2. The punctuation characters comma, semicolon and period, when immediately followed by a space, are separators. These separators may appear in a COBOL source program only where explicitly permitted by the general formats, by format punctuation rules (see FORMATS and RULES in Section 1), by statement and sentence structure definitions (see STATEMENTS and SENTENCES in this Section), or reference format rules (See REFERENCE FORMAT in this Section).

3. The punctuation characters right and left parenthesis are separators. Parenthesis may appear only in balanced pairs of left and right parentheses delimiting subscripts, indices, arithmetic expressions, or conditions.

4. The punctuation character quotation mark is a separator. An opening quotation mark must be immediately preceded by a space or left parenthesis; a closing quotation mark must be immediately followed by one of the separators space, comma, semicolon, period, or right parenthesis.

   Quotation marks may appear only in balanced pairs delimiting nonnumeric literals except when the literal is continued. (See CONTINUATION OF LINES in this Section).

5. The separator space may optionally immediately precede all separators except the following:

   a. As specified by reference format rules see REFERENCE FORMAT in this Section.

   b. The separator closing quotation mark. In this case, a preceding space is considered as part of the nonnumeric literal and not as a separator.

6. The separator space is optional and can immediately follow any separator except the opening quotation mark. In this case, a following space is considered as part of the nonnumeric literal and not as a separator.

Any punctuation character which appears as part of the specification of a PICTURE character-string (See Section 3) or numeric literal is not considered as a punctuation character, but rather as a symbol used in the specification of that PICTURE character-string or numeric literal. PICTURE character-strings are delimited only by the separators space, comma, semicolon, or period.

The rules established for the formation of separators do not apply to the characters which comprise the contents of nonnumeric literals, comment-entries, or comment lines.

Character-Strings

A character-string is a character or a sequence of contiguous characters which forms a L/II COBOL word, a literal, a PICTURE character-string, or a comment-entry. A character-string is delimited by separators.

COBOL Words

A COBOL word is a character-string of not more than 30 characters which forms a user defined word, a system-name, or a reserved word. Within a given source program these classes form disjoint sets; a COBOL word may belong to one and only one of these classes.

User-Defined Words

A user-defined word is a COBOL word that must be supplied by the user to satisfy the format of a clause or statement. Each character of a user-defined word is selected from the set of characters 'A', 'B', 'C', ... 'Z', 'a', 'b', 'c', ...'z', '0', ...'9', and '-', except that the '-' may not appear as the first or last character. The exception to this rule is a text-name which must be a normal alphanumeric literal.

User-defined word types which are implemented are as follows:

alphabet-name
cd-name
condition-name
data-name
file-name
index-name
level-number
mnemonic-name
paragraph-name
program-name
record-name
routine-name
section-name
segment-number
text-name

Within a given source program, 11 of these 15 types of user-defined words are grouped into nine disjoint sets. The disjoint sets are:

alphabet-names
cd-names
condition-names, data-names, and record-names
file-names
index-names
mnemonic-names
paragraph-names
program-names
routine-names
section-names
text-names

All user-defined words, except segment-numbers and level-numbers, can belong to one and only one of these disjoint sets. Further, all user-defined words within a given disjoint set must be unique, because no other user-defined word in the same source program has identical spelling or punctuation. (See UNIQUENESS OF REFERENCE in this Section).

With the exception of paragraph-name, section-name, level-number and segment-number, all user-defined words must contain at least one alphabetic character. Segment-numbers and level-numbers need not be unique; a given specification of a segment-number or level-number may even be identical to a paragraph-name or section-name.

Condition-Name

A condition-name is a name which is assigned to a specific value, set of values, or range of values, within a complete set of values that a data item may assume. The data item itself is called a conditional variable.

Condition-names may be defined in the Data Division or in the SPECIAL-NAMES paragraph with the Environment Division where a condition-name must be assigned to the ON STATUS or OFF STATUS, or both, of the run time switches.

A condition-name is used only in the RERUN clause or in conditions as an abbreviation for the relation condition; this relation condition posits that the associated conditional variable is equal to one of the set of values to which that condition-name is assigned.

Mnemonic-Name

A mnemonic-name assigns a user-defined word to an implementor-name. These associations are established in the SPECIAL-NAMES paragraph of the Environment Division. (See SPECIAL-NAMES in Section 3).

Paragraph-Name

A paragraph-name is a word which names a paragraph in the Procedure Division. Paragraph-names are equivalent if, and only if, they are composed of the same sequence of the same number of digits and/or characters.

Section-Name

A section-name is a word which names a section in the Procedure Division. Section names are equivalent if, and only if, they are composed of the same sequence of the same number of digits and/or characters.

Other User-Defined Names

See the glossary in Appendix C for definitions of all other types of user-defined words.

System-Names

A system-name is a COBOL word which is used to communicate with the operating environment. Each character used in the formation of a system-name must be selected from the set of characters 'A', 'B', 'C', ...'Z', 'a', 'b', ... 'z', 'O', ... '9' and '-', except that the '-' may not appear as the first or last character.

There are three types of system-names:

1. computer-name
2. implementor-name
3. language-name

Within a given implementation these three types of system-names form disjoint sets; a given system-name may belong to one and only one of them.

The system-names listed above, are individually defined in the glossary in Appendix C.

Reserved Words

A reserved word is a COBOL word that is one of a specified list of words which may be used in COBOL source programs, but which must not appear in the programs as user-defined words or system-names. Reserved words can only be used as specified in the general formats. (See Appendix A).

There are six types of reserved words:

1. Key words
2. Optional words
3. Connectives
4. Special registers
5. Figurative constants
6. Special-character words

Key Words

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

Key words are of three types:

1. Verbs such as ADD, READ, and ENTER.
2. Required words, which appear in statement and entry formats.
3. Words which have a specific functional meaning such as NEGATIVE, SECTION, etc.

Optional Words

Within each format, uppercase words that are not underlined are called optional words and may appear at the user's option. The presence or absence of an optional word does not alter the semantics of the COBOL program in which it appears.

Connectives

Series connectives link two or more consecutive operands: , (separator comma) or ; (separator semicolon).

Figurative Constants

Certain reserved words are used to name and reference specific constant values. These reserved words are specified under Figurative Constant Values in this chapter.

Literals

A literal is a character-string whose value is implied by an ordered set of characters of which the literal is composed or by specification of a reserved word which references a figurative constant. Every literal belongs to one of two types, nonnumeric or numeric.

Nonnumeric Literals

A nonnumeric literal is a character-string delimited on both ends of quotation marks and consisting of any allowable character in the computer's character set. Allowed are nonnumeric literals of 1 through 128 characters in length. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used. The value of a nonnumeric literal in the object program is the string of characters itself, except:

1. The delimiting quotation marks are excluded, and

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

All other punctuation characters are part of the value of the nonnumeric literal rather than separators; all nonnumeric literals are category alphanumeric. (See The PICTURE Clause in Section 3). In addition, hexadecimal binary values can be attributed to nonnumeric literals by expressing literals as: X"nn", where n is a hexadecimal character in the set 0-9 A-F; nn may be repeated up to 128 times, but the number of hex digits must be even.

Numeric Literals

A numeric literal is a character-string whose characters are selected from the digits '0' through '9', the plus sign, the minus sign, and/or the decimal point. The implementation allows for numeric literals of 1 through 18 digits in length. The rules for the formation of numeric literals are as follows:

1. A literal must contain at least one digit.

2. A literal must not contain more than one sign character. If a sign is used, it must appear as the leftmost character of the literal. If the literal is unsigned, the literal is positive.

3. A literal must not contain more than one decimal point. The decimal point is treated as an assumed decimal point, and may appear anywhere within the literal except as the rightmost character. If the literal contains no decimal point, the literal is an integer.

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

4. The value of a numeric literal is the algebraic quality represented by the characters in the numeric literal. Every numeric literal is category numeric. (See The PICTURE Clause in Section 3). The size of a numeric literal in standard data format characters is equal to the number of digits specified by the user.

Figurative Constant Values

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

The figurative constant values and the reserved words used to reference them are shown in Table 2-1.

Table 2-1. Figurative Constants and their Reserved Words

CONSTANT	REPRESENTATION
ZERO	Represents the value '0', or one or more of the character '0' depending on context.
ZEROS ZEROES	
SPACE SPACES	Represents one or more of the character space from the computer's character set.
HIGH-VALUE HIGH-VALUES	Represents one or more of the character that has the highest ordinal position in the program collating sequence.
LOW-VALUE LOW-VALUES	Represents one or more of the character that has the lowest ordinal position in the program collating sequence.
QUOTE QUOTES	Represents one or more of the character "". The word QUOTE or QUOTES cannot be used in place of a quotation mark in a source program to bound a nonnumeric literal. Thus, QUOTE ABD QUOTE is incorrect as a way of stating the nonnumeric literal "ABD".
ALL literal	Represents one or more characters of the string of characters comprising the literal. The literal must be either a nonnumeric literal or a figurative constant other than ALL literal. When a figurative constant is used, the word ALL is redundant and is used for readability only.

When a figurative constant represents a string of one or more characters, the length of the string is determined by the compiler from context according to the following rules:

1. When a figurative constant is associated with another data item, as when the figurative constant is moved to or compared with another data item, the string of characters specified by the figurative constant is repeated character by character on the right until the size of the resultant string is equal to the size in characters of the associated data item. This is done prior to and independent of the application of any JUSTIFIED clause that may be associated with the data item.

2. When a figurative constant is not associated with another data item, as when the figurative constant appears in a DISPLAY or STOP statement, the length of the string is one character.

   DISPLAY SPACE is, of course, an exception.

A figurative constant may be used wherever a literal appears in a format, except that whenever the literal is restricted to having only numeric characters in it, the only figurative constant permitted is ZERO (ZEROS, ZEROES).

When the figurative constants HIGH-VALUE(S) or LOW-VALUE(S) are used in the source program, the actual character associated with each figurative constant depends upon the program collating sequence specified. (See The OBJECT-COMPUTER Paragraph, and The SPECIAL-NAMES Paragraph in Section 3).

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

PICTURE Character-Strings

A PICTURE character-string consists of certain combinations of characters in the COBOL character set used as symbols. See The PICTURE Clause in Section 3 for the PICTURE character-string and for the rules that govern their use.

Any punctuation character which appears as part of the specification of a PICTURE character-string is not considered as a punctuation character, but rather as a symbol used in the specification of that PICTURE character-string.

Comment-Entries

A comment-entry is an entry in the Identification Division that may be any combination of characters from the computer's character set.

CONCEPT OF COMPUTER INDEPENDENT DATA DESCRIPTION

To make data as computer independent as possible, the characteristics or properties of the data are described in relation to a standard data format rather than an equipment-oriented format. This standard data format is oriented to general data processing applications and uses the decimal system to represent numbers (regardless of the radix used by the computer) and the remaining characters in the L/II COBOL character set to describe nonnumeric data items.

Concept of Levels

A level concept 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.

The most basic 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 are 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.

Level-Numbers

A system of level-numbers shows the organization of elementary items and group items. Since records are the most inclusive data items, level-numbers for records start at 01. Less inclusive data items are assigned higher (not necessarily successive) level-numbers not greater in value than 49. A maximum of 49 levels in a record is allowed. There are special level-numbers, 66, 77 and 88 which are exceptions to this rule (see below). Separate entries are written in the source program for each level-number used.

A group includes all group and elementary items following it until a level-number less than or equal to the level-number for that group is encountered. All items which are immediately subordinate to a given group item must be described using identical level-numbers greater than the level-number used to describe that group item.

Three types of entries exist for which there is no true concept of level. These are:

1. Entries that specify elementary items or groups introduced by a RENAMES clause

2. Entries that specify noncontiguous working storage and linkage data items

3. Entries that specify condition-names.

Entries describing items by means of RENAMES clauses for the purpose of regrouping data items have been assigned the special level-number 66.

Entries that specify noncontiguous data items, which are not subdivsions of other items, and are not themselves subdivided, have been assigned the special level-number 77.

Entries that specify condition-names, to be associated with particular values of a conditional variable, have been assigned the special level-number .88.

Concept of Classes of Data

The five categories of data items (See The PICTURE Clause in Chapter 3) 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 except for an index data 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. Table 2-2 depicts the relationship of the class and categories of data items.

Table 2-2 Data Levels, classes and categories

LEVEL OF ITEM	CLASS	CATEGORY
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Numeric Edited Alphanumeric	Alphanumeric Edited Alphanumeric
Non-Elementary Group	Elementary	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

Selection of Character Representation and Radix

The value of a numeric item may be represented in either binary or decimal form, depending on the equipment. In addition, there are several ways of expressing decimal. Since these representations are actually combinations of bits, they are commonly called binary-coded decimal forms. The four standard formats used for storing numeric data in CIS COBOL are as follows:

1. As alphanumeric stored one per byte in ASCII representation.

2. As numeric characters defined by USAGE IS DISPLAY (See The USAGE Clause in Section 3) one per byte in ASCII representation. If they are signed and the sign is specified as INCLUDED, bit 6 of the leading or trailing byte of the field is set for negative, depending on the field definition. If a SEPARATE sign is specified as a one byte ASCII + or -, a sign is added as the leading or trailing byte. If no SIGN clause is specified, bit 6 of the trailing digit is set to indicate negative by default.

3. As numeric characters defined by USAGE IS COMP or COMPUTATIONAL in pure binary form. If the field is signed, the number is held in its twos-complement form. Storage is then dependent on the number of 9's in the PICTURE clause (See The PICTURE Clause in Section 3) and on whether the field is SIGNed or not (See The SIGN Clause in Section 3).

Table 2-3 shows the storage requirements for each COMP(UTATIONAL) PICTURE Clause.

Table 2-3. Numeric Data Storage for the COMP(UTATIONAL) PICTURE Clause.

Bytes Required	Number of Characters Signed	Unsigned
1	1-2	1-2
2	3-4	3-4
3	5-6	5-7
4	7-9	8-9
5	10-11	10-12
6	12-14	13-14
7	15-16	15-16
8	17-18	17-18

4. As numeric characters defined by USAGE IS COMPUTATIONAL-3 or USAGE IS COMP-3 in packed internal decimal form. Storage is dependent on the number of 9's in the PICTURE clause. The decimal numbers are stored as signed strings of variable length of 1 through 18 digits. The sign of the packed decimal number is always stored in place of the least significant quartet of the low order byte. Each byte contains two decimal positions (four bits per digit) and the digits (0 - 9) are encoded as BCD numbers (0000 - 1001). Numbers are represented in the field as right-justified values with a + or - sign as shown in the example below. The maximum number of digits permitted in arithmetic operands is 18.

EXAMPLE:

a. For COMPUTATIONAL-3 and PICTURE 9999, the number +1234 would be stored as follows:

```
... 0    1    2    3    4    F
  0000 0001 0010 0011 0100 1111

                        1 byte
```

where F represents the non-printing plus sign.

b. For COMPUTATIONAL-3 and PICTURE S9999, the number +1234 would be stored as follows:

Storage would be as in a above except that the least significant digit would be replaced by C (1100) representing the plus sign.

c. For COMPUTATIONAL-3 and PICTURE S9999, the number -1234 would be stored as follows:

Storage would be as in a above except that the least significant byte would be replaced by D (1101) representing the minus sign.

Table 2-4 shows the storage requirements for each COMP-3 clause.

Table 2-4. Numeric Data Storage for the COMPUTATION-3 PICTURE clause.

Bytes Required	Number of Characters (Signed or Unsigned)
1	1
2	2-3
3	4-5
4	6-7
5	8-9
6	10-11
7	12-13
8	14-15
9	16-17
10	18

Algebraic Signs

Algebraic signs fall into two categories: operational signs, which are associated with signed numeric data items and signed numeric literals to indicate their algebraic properties; and editing signs, which appear on edited reports to identify the sign of the item.

The SIGN Clause permits the programmer to state explicitly, the location of the operational sign. The Clause is optional; if it is not used, operational signs will be represented as defined by setting bit 6 of the trailing digit for ASCII numbers. (See above).

Editing signs are inserted into a data item through the use of the sign control symbols of The PICTURE Clause.

Standard Alignment Rules

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

1. If the receiving data item is described as numeric:

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

   b. When an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately following its rightmost character and is aligned as in paragraph a. above.

2. If the receving data item is a numeric edited data item, the data moved to the edited item is 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.

3. If the receiving data item is alphanumeric (other than a numeric edited data item), alphanumeric edited or alphabetic, the sending data is moved to the receiving character positions and aligned at the leftmost character position in the data item with space fill or truncation to the right, as required.

If the JUSTIFIED Clause is specified for the receiving item, these standard rules are modified as described in The JUSTIFIED Clause in Section 3.

Uniqueness of Reference

Qualification

Every user-specified name that defines an element in a COBOL source program must be unique, either because no other name has the identical spelling and hyphenation, or because the name exists within a hierarchy of names such that references to the name can be made unique by mentioning one or more of the higher levels of the hierarchy. The higher levels are called qualifiers and this process that specifies uniqueness is called qualification. Enough qualification must be mentioned to make the name unique; however, it may not be necessary to mention all levels of the hierarchy. Within the Data Division, all data-names used for qualification must be associated with a level indicator or a level-number. Therefore, two identical data-names must not appear as entries subordinate to a group item unless they are capable of being made unique through qualification. In the Procedure Division, two identical paragraph-names must not appear in the same section.

In the hierarchy of qualification, names associated with a level indicator are the most significant, then those names associated with level-number 01, then 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 procedure-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 procedure-name.

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

The general formats for qualification are:

Format 1

$$\left\{\begin{array}{l}\text{data-name-1}\\\text{condition-name}\end{array}\right\}\left[\left\{\begin{array}{l}\underline{\text{OF}}\\\underline{\text{IN}}\end{array}\right\}\text{data-name-2}\right]\ldots$$

Format 2

$$\text{paragraph-name}\quad\left[\left\{\begin{array}{l}\underline{\text{OF}}\\\underline{\text{IN}}\end{array}\right\}\text{section-name}\right]$$

Format 3

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

The rules for qualification are as follows:

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

2. The same name must not appear at two levels in a hierarchy.

3. 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, Environment, and Data Divisions (except in the REDEFINES clause where qualification is unnecessary and must not be used.)

4. 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 referred to from within the same section.

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

6. A name can be qualified even though it does not need qualifications; if there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used. The complete set of qualifiers for a data-name must not be the same as any partial set of qualifiers for another data-name.

   Qualified data-names may have any number of qualifiers up to an implementor-defined limit. This limit must be at least five.

7. If more than one COBOL library is available to the compiler during compilation, text-name must be qualified each time it is referenced.

Subscripting

Subscripts can be used only when reference is made to an individual element within a list or table of like elements that have not been assigned individual data-names (See The OCCURS Clause in Section 4).

The subscript can be represented either 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 subscript may be signed and, if signed, it must be positive. The lowest possible 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.

The subscript, or set of subscripts, that identifies the table element is delimited by the balanced pair of separators left parenthesis and right parenthesis following the table element data-name. The table element data-name appended with a subscript is called a subscripted data-name or an identifier. When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the data organization.

The format is:

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

Indexing

References can be made to individual elements with 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 phrase in the definition of a table. A name given in the INDEXED BY phrase 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-name must be initialized before it is used as a table reference. An index-name can be given an initial value by a SET statement.

Direct indexing is specified by using an index-name in the form of a subscript. Relative indexing is specified when the 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 (when 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 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 format for indexing is:

$$\left\{\begin{array}{l}\text{data-name}\\ \text{condition-name}\end{array}\right\} \ (\ \left\{\begin{array}{l}\text{index-name-1}\\ \text{literal-1}\end{array}\right\} \left[\left\{\begin{array}{l}+\\ -\end{array}\right\}\text{literal-2}\right]$$

$$\left[\ , \left\{\begin{array}{l}\text{index-name-2}\\ \text{literal-3}\end{array}\right\} \left[\left\{\begin{array}{l}+\\ -\end{array}\right\}\text{literal-4}\right] \left[\ , \left\{\begin{array}{l}\text{index-name-3}\\ \text{literal-5}\end{array}\right\} \left[\left\{\begin{array}{l}+\\ -\end{array}\right\}\text{literal-6}\right]\right]\right] \ )$$

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 subscripts or indices necessary to ensure uniqueness.

The general formats for identifiers are:

Format 1:

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

Format 2:

```
data-name-1 [  ( { index-name-1 }        [ {+} literal-2 ]
                 { literal-1    }        [ {-}           ]

[   { index-name-2 } [ {+} literal-4 ]   [   { index-name-3 } [ {+} literal-6 ]] )   ]
[ , { literal-3    } [ {-}           ]   [ , { literal-5    } [ {-}           ]]     ]
```

Restrictions on subscripting and indexing are:

1. A data-name must not itself be subscripted nor indexed when that data-name is being used as an index, or subscript.

2. Indexing is not permitted where subscripting is not permitted.

3. 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 associated with index-names as data in a form specified by the implementor. Such data items are called index data items.

4. Literal-1, literal-3, literal-5, in the above format must be positive numeric integers. Literal-2, literal-4, literal-6 must be unsigned numeric integers.

Condition-Name

Each condition-name must be unique, or be made unique through qualification and/or indexing, or subscripting. If qualification is used to make a condition-name unique, the associated conditional variable may be used as the first qualifier. If qualification is used, the hierarchy of names associated with the conditional variable or the conditional variable itself must be used to make the condition-name unique.

If references to a conditional variable require indexing or subscripting, then references to any of its condition-names also require the same combination of indexing or subscripting.

The format and restrictions on the combined use of qualification, subscripting, and indexing of condition-names is exactly that of 'identifier' except that data-name-1 is replaced by condition-name-1.

In the general formats, 'condition-name' refers to a condition-name qualified, indexed or subscripted, as necessary.

EXPLICIT AND IMPLICIT SPECIFICATIONS

There are three types of explicit and implicit specifications that occur in COBOL source programs:

1. Explicit and implicit Procedure Division references
2. Explicit and implicit transfers of control
3. Explicit and implicit attributes.

Explicit and Implicit Procedure Division References

A COBOL source program can reference data items either explicitly or implicitly in Procedure Division statements. An explicit reference occurs when the name of the referenced item is written in a Procedure Division statement or when the name of the referenced item is copied into the Procedure Division by the processing of a COPY statement. An implicit reference occurs when the item is referenced by a Procedure Division statement without the name of the referenced item being written in the source statement. An implicit reference also occurs, during the execution of a PERFORM statement, when the index or data item referenced by the index-name or identifier specified in the VARYING, AFTER or UNTIL phrase is initialized, modified, or evaluated by the control mechanism associated with that PERFORM statement. Such an implicit reference occurs if and only if the data item contributes to the execution of the statement.

Explicit and Implicit Transfers of Control

The mechanism that controls program flow transfers control from statement to statement in the sequence in which they were written in the source program unless an explicit transfer of control overrides this sequence or there is no next executable statement to which control can be passed. The transfer of control from statement to statement occurs without the writing of an explicit Procedure Division statement, and therefore, is an implicit transfer of control.

COBOL provides both explicit and implicit means of altering the implicit control transfer mechanism.

In addition to the implicit transfer of control between consecutive statements, implicit transfer of control also occurs when the normal flow is altered without the execution of a procedure branching statement. COBOL provides the following types of implicit control flow alterations which override the statement-to-statement transfers of control:

1. If a paragraph is being executed under control of another COBOL statement (for example, PERFORM, USE, SORT and MERGE) and the paragraph is the last paragraph in the range of the controlling statement, then an implied transfer of control occurs from the last statement in the paragraph to the control mechanism of the last executed controlling statement. Further, if a paragraph is being executed under the control of a PERFORM statement which causes iterative execution and that paragraph is the first paragraph in the range of that PERFORM statement, an implicit transfer of control occurs between the control mechanism associated with that PERFORM statement and the first statement in that paragraph for each iterative execution of the paragraph.

2. When a SORT or MERGE statement is executed, an implicit transfer of control occurs to any associated input or output procedures.

3. When any COBOL statement is executed which results in the execution of a declarative section, an implicit transfer of control to the declarative section occurs. Note that another implicit transfer of control occurs after execution of the declarative section, as described in (1) above.

An explicit transfer of control consists of an alteration of the implicit control transfer mechanism by the execution of a procedure branching or conditional statement. (See STATEMENTS AND SENTENCES later in this Section.) An explicit transfer of control can be caused only by the execution of a procedure branching or conditional statement. The execution of the procedure branching statement ALTER does not in itself constitute an explicit transfer of control, but affects the explicit transfer of control that occurs when the associated GO TO statement is executed. The procedure branching statement EXIT PROGRAM causes an explicit transfer of control when the statement is executed in a called program.

In this document, the term 'next executable statement' is used to refer to the next COBOL statement to which control is transferred according to the rules above and the rules associated with each language element in the Procedure Division.

There is no next executable statement following:

1. The last statement in a declarative section when the paragraph in which it appears is not being executed under the control of some other COBOL statement.

2. The last statement in a program when the paragraph in which it appears is not being executed under the control of some other COBOL statement.

Explicit and Implicit Attributes

Attributes may be implicitly or explicitly specified. Any attribute which has been explicitly specified is called an explicit attribute. If an attribute has not been specified explicitly, then the attribute takes on the default specification. Such an attribute is known as an implicit attribute.

For example, the usage of a data item need not be specified, in which case a data item's usage is DISPLAY.

PROGRAM STRUCTURE

A L/II COBOL program consists of four divisions:

1. IDENTIFICATION DIVISION - An identification of the program.
2. ENVIRONMENT DIVISION - A description of the equipment to be used to compile and run the program.
3. DATA DIVISION - A description of the data to be processed.
4. PROCEDURE DIVISION - A set of procedures to specify the operations to be performed on the data.

Each division, is divided into sections which are further divided into paragraphs, which in turn are made up of sentences.

IDENTIFICATION DIVISION

GENERAL DESCRIPTION

The Identification Division is included in a COBOL source program at the user's discretion. This division identifies both the source program and the resultant output listing. In addition, the user may include the date the program is written, the date the compilation of the source program is accomplished and such other information as desired under the paragraphs in the general format shown below.

ORGANIZATION

Paragraph headers identify the type of information contained in the paragraph. All paragraphs are optional and may be included in this division at the user's choice, in order of presentation shown by the format below.

STRUCTURE

The following is the general format of the paragraphs in the Identification Division and it defines the order of presentation in the source program.

General Format

```
[IDENTIFICATION DIVISION.]

[PROGRAM-ID. program-name.]

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

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

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

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

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

ENVIRONMENT DIVISION

GENERAL DESCRIPTION

The Environment Division specifies a standard method of expressing those aspects of a data processing problem that are dependent upon the physical characteristics of a specific computer. This division allows specification of the configuration of the compiling computer and the object computer. In addition, information relating to input-out control, special hardware characteristics and control techniques can be given.

The Environment Division is included in a COBOL source program at the user's discretion.

ORGANIZATION

Two sections make up the Environment Division: the Configuration Section and the Input-Output Section.

The Configuration Section deals with the characteristics of the source computer and the object computer. This section is divided into three paragraphs: the SOURCE-COMPUTER paragraph, which describes the computer configuration on which the source program is compiled; the OBJECT-COMPUTER paragraph, which describes the computer configuration on which the object program produced by the compiler is to be run; and the SPECIAL-NAMES paragraph, which relates the implementation-names used by the compiler to the mnemonic-names used in the source program.

The Input-Output Section deals with the information needed to control transmission and handling of data between external media and the object program. This section is divided into two paragraphs: the FILE-CONTROL paragraph which names and associates the files with external media; and the I-O-CONTROL paragraph which defines special control techniques to be used in the object program.

STRUCTURE

The following is the general format of the sections and paragraphs in the Environment Division, and defines the order of presentation in the source program.

General Format

```
[ENVIRONMENT DIVISION.]
[CONFIGURATION SECTION.]
[SOURCE-COMPUTER. source-computer-entry ]
[OBJECT-COMPUTER. object-computer-entry ]
[SPECIAL-NAMES. special-names-entry ]
[INPUT-OUTPUT SECTION.]
FILE-CONTROL. file-control-entry ...
[I-O-CONTROL. input-output-control-entry ]
```

DATA DIVISION

OVERALL APPROACH

The Data Division describes the data that the object program is to accept as input, to manipulate, to create, or to produce as output. Data to be processed falls into three categories:

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

2. That which is developed internally and placed into intermediate or working storage, or places into specific format for output reporting purposes.

3. Constants which are defined by the user.

PHYSICAL AND LOGICAL ASPECTS OF DATA DESCRIPTION

Data Division Organization

The DATA DIVISION is subdivided into sections.

The FILE SECTION defines the structure of data files. Each file is defined by a file description entry and one or more record descriptions, or by a file description entry and one or more report description entries. Record descriptions are written immediately following the file description entry. The WORKING-STORAGE SECTION describes records and noncontiguous data items which are not part of external data files but are developed and processed internally. It also describes data items whose values are assigned in the source program and do not change during the execution of the object program. The LINKAGE SECTION appears in the called program and describes data items that are to be referred to by the calling program and the called program. Its structure is the same as the WORKING-STORAGE SECTION.

General Format

The following gives the general format of the sections in the Data Division, and defines the order of their presentation in the source program.

```
[ DATA DIVISION. ]

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

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

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

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

PROCEDURE DIVISION

GENERAL DESCRIPTION

The Procedure Division must be included in every COBOL source program. This division may contain declarative procedures.

Declaratives

Declarative sections must be grouped at the beginning of the Procedure Division preceded by the key word DECLARATIVES and followed by the key words END DECLARATIVES. (See Descriptions of the USE Statement in Sections 5, 6 and 7 and the DEBUG Section 11).

Procedures

A procedure is composed of a paragraph, or group of successive paragraphs, or a section, or a group of successive sections within the Procedure Division. If one paragraph is in a section, then all paragraphs must be in sections. A procedure-name is a word used to refer to a paragraph or section in the source program in which it occurs. It consists of a paragraph-name (which may be qualified), or a section-name.

The end of the Procedure Division and 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 zero, one, or more successive paragraphs. A section ends immediately before the next section or 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 a period and a space and by zero, 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 or, in the declaratives portion of the Procedure Division, at the key words END DECLARATIVES.

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

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 indicate some other order.

General Format

Procedure Division Header

The Procedure Division is identified by and must begin with the following header:

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

Procedure Division Body

The body of the Procedure Division must conform to one of the following formats:

```
DECLARATIVES.

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

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

END DECLARATIVES.

section-name SECTION  [segment-number].

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

Format 2:

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

STATEMENTS AND SENTENCES

There are three types of statements:

1. Conditional statements,
2. Compiler directing statements,
3. Imperative statements.

There are three types of sentences:

1. Conditional sentences,
2. Compiler directing sentences,
3. Imperative sentences.

Conditional Statement

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

A conditional statement is one of the following:

* An IF statement.

* A READ statement that specifies the AT END or INVALID KEY phrase.

* A WRITE statement that specifies the INVALID KEY phrase.

* A START, REWRITE or DELETE statement that specifies the INVALID KEY phrase.

* An arithmetic statement (ADD, DIVIDE, MULTIPLY, SUBTRACT) that specifies the SIZE ERROR phrase.

* A CALL statement that specifies the ON OVERFLOW phrase.

Conditional Sentence

A conditional sentence is a conditional statement, optionally preceded by an imperative statement, terminated by a period followed by a space.

Compiler Directing Statement

A compiler directing statement consists of a compiler directing verb and its operands. The compiler directing verbs are COPY, ENTER and USE (See The COPY STATEMENT in Section 9, The ENTER STATEMENT in Section 3, and The USE STATEMENT in Sections 5, 6 and 7). A compiler directing statement causes the compiler to take on specified action during compilation.

Compiler Directing Sentence

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

Imperative Statement

An imperative statement indicates a specific unconditional action to be taken by the object program. 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.

The imperative verbs are:

ACCEPT	GO	SET
ADD[1]	INSPECT	START[2]
ALTER	MOVE	STOP
CALL[3]	MULTIPLY[1]	SUBTRACT[1]
CANCEL	OPEN	WRITE[6]
COMPUTE	PERFORM	
CLOSE	READ[5]	
DELETE[2]	REWRITE[2]	
DISPLAY		
DIVIDE[1]		
EXIT		

1 - Without the optional SIZE ERROR phrase.
2 - Without the optional INVALID KEY phrase.
3 - Without the optional ON OVERFLOW phrase.
5 - Without the optional AT END phrase or INVALID KEY phrase.
6 - Without the optional INVALID KEY phrase or END-OF-PAGE phrase.

When 'imperative-statement' appears in the general format of statements, 'imperative-statement' refers to that sequence of consecutive imperative statements that must be ended by a period or an ELSE phrase associated with a previous IF statement.

Imperative Sentence

An imperative sentence is an imperative statement terminated by a period followed by a space.

REFERENCE FORMAT

GENERAL DESCRIPTION

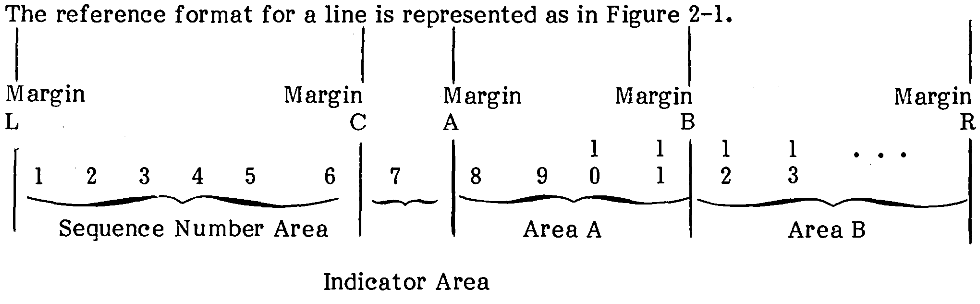
The reference format, which provides a standard method for describing COBOL source programs, is described in terms of character positions in a line on an input-output medium. The L/II COBOL compiler accepts source programs written in reference format and produces an output listing of the source program input in reference format.

The rules for spacing given in the discussion of the reference format take precedence over all other rules for spacing.

The divisions of a source program must be in order as follows: the Identification Division, then the Environment Division, then the Data Division, then the Procedure Division. Each division must be written according to the rules for the reference format.

REFERENCE FORMAT REPRESENTATION

The reference format for a line is represented as in Figure 2-1.



Margin L is immediately to the left of the leftmost character position of a line.

Margin C is between the 6th and 7th character positions of a line.

Margin A is between the 7th and 8th character positions of a line.

Margin B is between the 11th and 12th character positions of a line.

Margin R is immediately to the right of the rightmost character position of a line.

Figure 2 - 1. Reference format for a COBOL Source Line.

The sequence number area occupies six character positions (1-6), and is between Margin L and Margin C.

The indicator area is the 7th character position of a line.

Area A occupies character positions 8, 9, 10 and 11, and is between margin A and margin B.

Area B occupies character positions 12 through 72 inclusive; it begins immediately to the right of Margin B and terminates immediately to the left of Margin R.

Sequence Numbers

A sequence number, consisting of six digits in the sequence area, may be used to label a source program line.

Continuation of Lines

Whenever a sentence, entry, phrase, or clause requires more than one line, it may be continued by starting subsequent line(s) in area B. These subsequent lines are called the continuation line(s). The line being continued is called the continued line. Any word or literal may be broken in such a way that part of it appears on a continuation line.

A hyphen in the indicator area of a line indicates that the first nonblank character in area B of the current line is the successor of the last nonblank character of the preceding line without any intervening space. However, if the continued line contains a nonnumeric literal without closing quotation mark, the first nonblank character in area B on the continuation line must be a quotation mark, and the continuation starts with the character immediately after that quotation mark. All spaces at the end of the continued line are considered part of the literal. Area A of a continuation line must be blank.

If there is no hyphen in the indicator area of a line, it is assumed that the last character in the preceding line is followed by a space.

Blank Lines

A blank line is one that is blank from margin C to margin R, inclusive. A blank line can appear anywhere in the source program, except immediately preceding a continuation line. (See Figure 2-1).

DIVISION, SECTION, PARAGRAPH FORMATS

Division Header

The division header must start in area A. (See Figure 2-1).

Section Header

The section header must start in area A. (See Figure 2-1).

A section consists of paragraphs in the Environment and Procedure Divisions and Data Division entries in the Data Division.

Paragraph Header, Paragraph-Name and Paragraph

A paragraph consists of a paragraph-name followed by a period and a space and by zero, one or more sentences, or a paragraph header followed by one or more entries. Comment entries may be included within a paragraph. The paragraph header or paragraph-name starts in area A of any line following the first line of a division or a section.

The first sentence or entry in a paragraph begins either on the same line as the paragraph header or paragraph-name or in area B of the next nonblank line that is not a comment line. Successive sentences or entries either begin in area B of the same line as the preceding sentence or entry or in area B of the next nonblank line that is not a comment line.

When the sentences or entries of a paragraph require more than one line they may be continued as described in CONTINUATION OF LINES in this Section.

DATA DIVISION ENTRIES

Each Data Division entry begins with a level indicator or a level-number, followed by a space, followed by its associated name (except in the Report Section), followed by a sequence of independent descriptive clauses. Each clause, except the last clause of an entry, may be terminated by either the separator semicolon or the separator comma. The last clause is always terminated by a period followed by a space.

There are two types of Data Division entries: those which begin with a level indicator and those which begin with a level-number.

A level indicator is any of the following: FD (See The FILE DESCRIPTION - COMPLETE ENTRY SKELETON in Sections 5, 6, and 7), SD (See The SORT MERGE FILE DESCRIPTION - COMPLETE ENTRY SKELETON in Section 8).

In those Data Division entries that begin with a level indicator, the level indicator begins in area A followed by a space and followed in area B with its associated name and appropriate descriptive information.

Those Data Division entries that begin with level-numbers are called data description entries.

A level-number has a value taken from the set of values 1 through 49, 66, 77 and 88. Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. At least one space must separate a level-number from the word following the level-number.

In those data description entries that begin with level-number 01 or 66, 77 and 88, the level-number begins in area A followed by a space and followed in area B by its associated record-name or item-name and appropriate descriptive information.

Successive data description entries may have the same format as the first or may be indented according to level-number. The entries in the output listing need be indented only if the input is indented. Indentation does not affect the magnitude of a level-number.

When level-numbers are to be indented, each new level-number may begin any number of spaces to the right of margin A. The extent of indentation to the right is determined only by the width of the physical medium.

DECLARATIVES

The key word DECLARATIVES and the key words END DECLARATIVES that precede and follow, respectively, the declaratives portion of the Procedure Division must appear on a line by itself. Each must begin in area A and be followed by a period and a space (See Figure 2-1).

COMMENT LINES

A comment line is any line with an asterisk in the continuation indicator area of the line. A comment line can appear as any line in a source program after the Identification Division header. Any combination of characters from the computer's character set may be included in area A and area B of that line (See Figure 2-1). The asterisk and the characters in area A and area B will be produced on the listing but serve as documentation only. A special form of comment line represented by a stroke in the indicator area of the line causes page ejection prior to printing the comment.

Successive comment lines are allowed. Continuation of comment lines is permitted, except that each continuation line must contain an '*' in the indicator area.

RESERVED WORDS

A full list of reserved words is given in Appendix A.

SECTION 3

THE NUCLEUS

FUNCTION OF THE NUCLEUS

The Nucleus provides a basic language capability for the internal processing of data within the basic structure of the four divisions of a program.

OVERALL LANGUAGE

NAME CHARACTERISTICS

L/II COBOL data-names need not begin with an alphabetic character; the alphabetic characters may be positioned anywhere within the data-name. Qualification is permitted and all data-names, condition-names, paragraph-names, and text-names need not be unique.

FIGURATIVE CONSTANTS

All the following figurative constants may be used: ZERO, ZEROS, ZEROES, SPACE, SPACES HIGH-VALUE, HIGH-VALUES, LOW-VALUE, LOW-VALUES, QUOTE, QUOTES, and ALL literal.

REFERENCE FORMAT

A word or numeric literal can be broken in such a way that part of it appears on a continuation line.

IDENTIFICATION DIVISION IN THE NUCLEUS

GENERAL DESCRIPTION

The Identification Division is included in a COBOL source program at the user's discretion. This division identifies the source program and the resultant output listing. In addition, the user may include the date the program is written and such other information as desired under the paragraphs in the general format shown below.

ORGANIZATION

Paragraph headers identify the type of information contained in the paragraph. All paragraphs are optional and may be included in this division at the user's choice, in the order of presentation shown by the general format below.

Structure

The general format of the paragraphs in the Identification Division is given below. Paragraphs can be in any order.

General Format

```
IDENTIFICATION DIVISION.

PROGRAM-ID.     program-name.

AUTHOR.         [comment-entry]  ...

INSTALLATION.   [comment-entry]  ...

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

DATE-COMPILED.    [comment-entry]  ..

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

Syntax Rules

1. The Identification Division must begin with the reserved words IDENTIFICATION DIVISION followed by a period and a space.

2. The comment-entry may be any combination of the characters from the computer's character set and may be written in Area B on one or more lines. The continuation of the comment-entry by the use of the hyphen in the indicator area is not permitted.

THE PROGRAM-ID PARAGRAPH

Function

The PROGRAM-ID paragraph gives the name by which a program is identified.

General Format

PROGRAM-ID. program-name.

Syntax Rules

1. The program-name must conform to the rules for formation of a user-defined word.

General Rules

1. The PROGRAM-ID paragraph contains the name of the program.
2. The program-name identifies the source program and all listings pertaining to a particular program.

THE DATE-COMPILED PARAGRAPH

Function

The DATE-COMPILED paragraph provides the compilation date in the Identification Division source program listing.

General Format

DATE-COMPILED. comment-entry ...

Syntax Rule

The comment-entry may be any combination of the characters from the computer's character set. The continuation of the comment entry by use of the hyphen is not permitted; however, the comment entry may be contained on one or more lines.

General Rule

The paragraph-name DATE-COMPILED causes a date entry string to be inserted during program compilation. If a DATE-COMPILED paragraph is present, the comment-entry is replaced in its entirety by the date string.

ENVIRONMENT DIVISION IN THE NUCLEUS

CONFIGURATION SECTION

The SOURCE-COMPUTER Paragraph

Function

The SOURCE-COMPUTER paragraph identifies the computer upon which the program is to be compiled.

General Format

SOURCE COMPUTER. computer-name.

Syntax Rule

Computer-name must be one COBOL word defined by the user.

General Rule

The computer-name provides a means for identifying equipment configuration, in which use the computer-name and its implied configuration are specified by the user.

The OBJECT-COMPUTER Paragraph

Function

The OBJECT-COMPUTER Paragraph identifies the computer on which the program is to be executed.

General Format

$$\underline{\text{OBJECT-COMPUTER}}.\quad \text{computer-name , }\underline{\text{MEMORY}}\text{ SIZE integer}\left\{\begin{array}{l}\underline{\text{WORDS}}\\ \underline{\text{CHARACTERS}}\\ \underline{\text{MODULES}}\end{array}\right\}$$

$$[\text{,PROGRAM COLLATING }\underline{\text{SEQUENCE}}\text{ IS alphabet-name}]\quad .$$

Syntax Rule

Computer-name must be one COBOL word defined by the user.

General Rules

1. The computer-name provides a means for identifying equipment configuration, in which case the computer-name and its implied configurations are specified by the user. The configuration definition contains specific information concerning the memory size.

2. If the PROGRAM COLLATING SEQUENCE Clause is specified, the collating sequence associated with alphabet-name is used to determine the truth value of any nonnumeric comparisons:

    a. Explicitly specified in relation conditions (see Relation Condition later in this Section).

    b. Explicitly specified in condition-name conditions; see Condition Name Condition (Conditional Variable).

3. If the PROGRAM COLLATING SEQUENCE Clause is not specified, the native collating sequence is used. Appendix B lists the full ASCII collating sequence (native) and those characters used in COBOL.

4. If the PROGRAM COLLATING SEQUENCE Clause is specified, the program collating sequence is the collating sequence associated with the alphabet-name specified in that Clause.

5. The PROGRAM COLLATING SEQUENCE Clause is also applied to any nonnumeric merge or sort keys.

The SPECIAL-NAMES Paragraph

Function

The SPECIAL-NAMES paragraph provides a means of relating implementor-names to user-specified mnemonic-names and of relating alphabet-names to character sets and/or collating sequences.

General Format

```
    SPECIAL-NAMES.

    [function-name-1 IS mnemonic-name-1]
            [function-name-2 IS mnemonic-name-2] ...
  _                                                                   _
 |          / 0 \                                                      |
 |  SWITCH {  .  }  IS mnemonic-name [,ON STATUS IS condition-name-1   |
 |         |  .  |                                                     |
 |          \ . /                     [,OFF STATUS IS condition-name-2]]|
 |            7                                                        |
 |_                                                                   _|
                    STANDARD-1
  [, alphabet-name IS / NATIVE                 ...                               _
  |                  |                  {THROUGH}  literal-2                   \ |
  |                  |  literal-1       {THRU   }                               ||
  |                  |             ALSO literal-3 , ALSO literal-4 ...      _    |
  |                  |  _          _                                  _     |   |
  |                   >|            |   {THROUGH}  literal-6           |    | ... >|
  |                  | | literal-5  |   {THRU   }                      |    |   | |
  |                  | |_           |ALSO literal-7[, ALSO literal-8]...|    |   | |
  |_                  \             |_                                _|   _|   / _|
    [, CURRENCY SIGN IS literal-9]
    [, DECIMAL-POINT IS COMMA]
    [, CONSOLE IS CRT]
    [, CURSOR is data-name-1]  .
```

Syntax Rules

1. Mnemonic-names can be any COBOL user-defined word and at least one constituent character must be alphabetic.

2. The literals specified in the literal phrase of the alphabet-name clause:

   a. If numeric, must be unsigned integers and must have a value within the range of one (1) through the maximum number of characters in the native character set.

   b. If nonnumeric and associated with a THROUGH or ALSO phrase, must each be one character in length.

3. If the literal phrase of the alphabet-name clause is specified, a given character must not be specified more than once in an alphabet-name clause.

4. The words THRU and THROUGH are equivalent.

General Rules

1. Function-name-1 specifies system devices or functions used by the compiler.

   The programmer can associate any user-defined COBOL word with a function-name.

   Mnemonic-name-1, -2, etc. can be used in the ACCEPT, DISPLAY or WRITE statements.

   Function-name-1, -2, etc. can be one of the following:

SYSIN	-	System logical input unit: the CRT keyboard
SYSOUT	-	System logical output unit: the CRT screen
TAB	-	Skip to head of form (WRITE, ADVANCING)

2. The SWITCH clause enables a mnemonic name to be set to one of two condition-names depending on the setting of a switch at run time by the operator.

3. The alphabet-name clause provides a means for relating a name to a specified character code set and/or collating sequence. When alphabet-name is referenced in the PROGRAM COLLATING SEQUENCE clause (see the OBJECT-COMPUTER Paragraph in this Section). The alphabet-name clause specifies a collating sequence. When alphabet-name is referenced in a CODE-SET clause in a file description entry (see The File Description - Complete Entry Skeleton in Section 5), the alphabet-name clause specifies a character code set.

   a. If the STANDARD-1 phrase is specified, the character code set or collating sequence identified is that defined in American National Standard Code for Information Interchange, X3.4-1968. Appendix B defines the correspondence between the characters of the standard character set and the characters of the native character set.

   b. If the NATIVE phrase is specified, the native character code set or native collating sequence is used. The native collating sequence is an in ANSI publication X3.4-1968 (see Appendix B).

4. The character that has the highest ordinal position in the program collating sequence specified is associated with the figurative constant HIGH-VALUE. If more than one character has the highest position in the program collating sequence, the last character specified is associated with the figurative constant HIGH-VALUE.

5. The character that has the lowest ordinal position in the program collating sequence specified is associated with the figurative constant LOW-VALUE. If more than one character has the lowest position in the program collating sequence, the first character specified is associated with the figurative constant LOW-VALUE.

6. The literal which appears in the CURRENCY SIGN IS literal clause is used in the PICTURE clause to represent the currency symbol. The literal is limited to a single character and must not be one of the following characters.

   * digits 0 thru 9;

   * alphabetic characters A, B, C, D, L, P, R, S, V, X, Z, or the space;

   * special characters '*', '+', '-', ',', '.', ';', '(', ')', '"', '/'or '='.

   If this clause is not present, only the currency sign is used in the PICTURE clause.

7. The clause DECIMAL-POINT IS COMMA means that the function of comma and period are exchanged in the character-string of the PICTURE clause and in numeric literals.

8. The clause CONSOLE IS changes the defaults in the ACCEPT and DISPLAY statements to the B 20 COBOL interactive extension that enables data to be accepted or displayed at any specified point on the screen. See the ACCEPT Statement and the DISPLAY Statement in this Section.

9. The clause CURSOR IS specifies the data-name to contain the CRT cursor address as used by the ACCEPT statement. If CURSOR IS is not specified the default cursor position on executing an ACCEPT statement is the 'Home' position at top left of the CRT screen. The CURSOR IS clause enables a program to retain the position at the end of execution of the last ACCEPT statement or to specify the initial position at the start of any ACCEPT statement. This is a useful facility when programming menu-type operator prompts. The operator need then only move the cursor to the selected option prompt and press GO or just press GO for the default option.

   Data-name contains the name of a PIC 9999 field in which the most significant 99 represents a line count in the range one to the maximum number of lines on the user screen, and the least significant 99 represents a character position in the range one to the maximum positions allowed by the width of the user screen. If data-name is zero, the effect is as if the CURSOR IS clause was not used, i.e., initial cursor position is top left of screen. (See also the ACCEPT Statement later in this Section).

DATA DIVISION IN THE NUCLEUS

WORKING STORAGE SECTION

The Working-Storage Section is composed of the section header, followed by data description entries for noncontiguous data items and/or record description entries. Each Working-Storage Section record name and noncontiguous item name must be unique since it cannot be qualified. Subordinate data-names need not be unique if they can be made unique by qualification.

Noncontiguous Working-Storage

Items and constants in Working-Storage which bear no hierarchical 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 in a separate data description entry which begins with the special level-number, 77.

The following data clauses are required in each data description entry:

* Level-number 77
* Data-name
* The PICTURE clause or the USAGE IS INDEX clause

Other data description clauses are optional and can be used to complete the description of the item if necessary.

Working-Storage Records

Data elements and constants in Working-Storage which bear a definite hierarchic relationship to one another must be grouped into records according to the rules for formation of record descriptions. All clauses which are used in record descriptions in the File Section can be used in record descriptions in the Working-Storage Section.

Initial Values

The initial value of any item in the Working-Storage Section except an index data item is specified by using the VALUE clause with the data item. The initial value of any index data item is unpredictable.

THE DATA DESCRIPTION - COMPLETE ENTRY SKELETON

Function

A data description entry specifies the characteristics of a particular item of data.

General Format

Format 1:

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

         [; REDEFINES data-name-2]

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

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

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

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

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

         [; BLANK WHEN ZERO]

         [; VALUE IS literal]   .
```

Format 2:

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

Format 3:

```
                     VALUE IS                      THROUGH
  88 condition-name; VALUES ARE    literal-1       THRU        literal-2

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

Syntax Rules

1. The level-number in Format 1 may be any number from 01-49 or 77.

2. The 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.

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

4. The words THRU and THROUGH are equivalent.

General Rules

1. The clauses SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO, must not be specified except for an elementary data item.

2. Format 3 is used for each condition-name. Each condition-name requires a separate entry with level-number 88. Format 3 contains the name of the condition and the value, values, or range of values asosciated with the condition-name. The condition-name entries for a particular conditional variable must follow the entry describing the item with which the condition-name is associated. A condition-name can be associated with any data description entry which contains a level-number except the following:

   a. Another condition-name.

   b. A level 66 item.

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

   d. An index data item (See the USAGE IS INDEX Clause in Section 4).

THE BLANK WHEN ZERO CLAUSE

Function

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

General Format

BLANK WHEN ZERO

Syntax Rule

The BLANK WHEN ZERO clause can be used only for an elementary item whose PICTURE is specified as numeric or numeric edited. (See the PICTURE Clause later in this Section).

General Rules

1. When the BLANK WHEN ZERO clause is used, the item will contain nothing but spaces if the value of the item is zero.
2. When the BLANK WHEN ZERO clause is used for an item whose PICTURE is numeric, the category of the item is considered to be numeric edited.

THE DATA-NAME OR FILLER CLAUSE

Function

A data-name specifies the name of the data being described. The word FILLER specifies an elementary item of the logical record that cannot be referred to explicitly.

General Format

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

Syntax Rule

In the File, Working-Storage, Communication and Linkage Sections, a data-name or the key word FILLER must be the first word following the level-number in each data description entry.

General Rule

The key word FILLER may be used to name an elementary item or group in a record. Under no circumstances can a FILLER item be referred to explicitly. However, the key word FILLER may be used as a conditional variable because such use does not require explicit reference to the FILLER Item but to its value.

THE JUSTIFIED CLAUSE

Function

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

General Format

$$\left\{\begin{array}{l}\underline{\text{JUSTIFIED}}\\\underline{\text{JUST}}\end{array}\right\}\quad\text{RIGHT}$$

Syntax Rules

1. The JUSTIFIED clause can be specified only at the elementary item level.
2. JUST is an abbreviation for JUSTIFIED.
3. The JUSTIFIED clause cannot be specified for any data item described as numeric or for which editing is specified.

General Rules

1. When a receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the leftmost characters are truncated. When the receiving data item is described with the JUSTIFIED clause and it is larger than the sending data item, the data is aligned at the rightmost character position in the data item with space fill for the leftmost character positions.
2. When the JUSTIFIED clause is omitted, the standard rules for aligning data within an elementary item apply. (See STANDARD ALIGNMENT RULES.)

LEVEL NUMBER

Function

The level-number shows the hierarchy of data within a logical record. In addition, it is used to identify entries for working storage items, linkage items, condition-names, and the RENAMES clause.

General Format

level-number

Syntax Rules

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

2. Data description entries subordinate to an FD, CD, or SD entry must have level-numbers with the values 01-49, 66 or 88. (See the FILE DESCRIPTION in Section 5).

3. Data description entries in the Working-Storage Section and Linkage Section must have level-numbers with the values 01-49.

General Rules

1. The level-number 01 identifies the first entry in each record description or a report group.

2. Special level numbers have been assigned to certain entries where there is no real concept of level:

   a. The level-number 77 is assigned to identify noncontiguous working storage data items, noncontiguous linkage data items, and can be used only as described by Format 1 of the data description skeleton. (See the DATA DESCRIPTION - COMPLETE ENTRY SKELETON in this Section).

   b. Level number 66 is assigned to identify RENAMES entries and can be used only as described in Format 2 of the data description skeleton earlier in this Section.

   c. **Level number 88 is assigned to entries which define condition-names associated with a conditional variable and can be used only as described in Format 3 of the data description skeleton earlier in this Section. A maximum number of 23 literals is allowed for level number 88 entries. If more are needed the THRU clause must be used.**

3. Multiple level 01 entries subordinate to any given level indicator, represent implicit redefinitions of the same area.

THE PICTURE CLAUSE

Function

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

General Format

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

Syntax Rules

1. A PICTURE clause can be specified only at the elementary item level.

2. 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.

3. The maximum number of characters allowed in the character-string is 30.

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

5. PIC is an abbreviation for PICTURE.

6. The asterisk when used as the zero suppression symbol and the clause BLANK WHEN ZERO may not appear in the same entry.

General Rules

There are five categories of data that can be described with a PICTURE clause: alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited. General Rules within these categories are given below:

Alphabetic Data Rules

1. Its PICTURE character-string can only contain the symbols 'A', 'B'; and

2. Its contents when represented in standard data format must be any combination of the twenty-six (26) letters of the Roman alphabet and the space from the COBOL character set.

Numeric Data Rules

1. The PICTURE character-string can only contain the symbols '9', 'P', 'S', and 'V'. The number of digit positions that can be described by the PICTURE character-string must range from 1 to 18 inclusive.

2. If unsigned, the data in standard data format must be a combination of the Arabic numerials '0', '1', '2', '3', '4', '5', '6', '7', '8', and '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign. (see the SIGN Clause later in this Section).

Alphanumeric Data Rules

1. The 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. A PICTURE character-string which contains all A's or all 9's does not define an alphanumeric item; and

2. The contents when represented in standard data format can consist of any characters in the computer's character set.

3. The maximum size for any alphanumeric field is:

   PIC X(8191)

Alphanumeric Edited Data Rules

1. Its PICTURE character-string is restricted to certain combinations of the following symbols: 'A', 'X', '9', 'B', '0 ', and '/' as follows:

   a. The character-string must contain at least one 'B' and at least one 'X' or at least one '0' (zero) and at least one 'X' or at least one '/' (stroke) and at least one 'X'; or

   b. The character-string must contain at least one '0' (zero) and at least one 'A' or at least one '/' (stroke) and at least one 'A'; and

2. The contents when represented in standard data format are allowable characters in the computer's set.

Numeric Edited Data Rules

1. Its PICTURE character-string is restricted to certain combinations of the symbols 'B', '/', 'P', 'V', 'Z', '0', '9', ',', '.', '*', '+', '-', 'CR', 'DB', and the currency symbol. The allowable combinations are determined from the order of procedence of symbols and the editing rules as follows:

   a. The number of digit positions that can be represented in the PICTURE character-string must range from 1 to 18 inclusive.

   b. The character-string must contain at least one '0', 'B', '/', 'Z', '*', '+', ',', '.', '-', 'CR', 'DB', or currency symbol.

2. The contents of the character positions of these symbols that are allowed to represent a digit in standard data format must be one of the numerals.

Elementary Item Size

The size of an elementary item, where size means the number of character positions occupied by the elementary item in standard data format, is determined by the number of allowable symbols that represent character positions. An integer which is enclosed in parentheses following the symbols 'A', ',', 'X', '9', 'P', 'Z', '*', 'B', '/', '0', '+', '-', or the currency symbol indicates the number of consecutive occurrences of the symbol. Note that the following symbols may appear only once in a given PICTURE: 'S', 'V', '.', 'CR', and 'DB'.

Symbols Used

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

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

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

P - Each '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 size of the data item. Scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or numeric items. 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's if 'P's are leftmost PICTURE characters and to the right if 'P's are right most 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 '.' (period) cannot both occur in the same PICTURE character-string. If, in any operation involving conversion of data from one form of internal representation to another, the data item being converted is described with the PICTURE character 'P', each digit position described by a 'P' is considered to contain the value zero, and the size of the data item is considered to include the digit positions so described.

S - The letter 'S' is used in a character-string to indicate the presence, but neither the representation nor, necessarily, the position of an operational sign; it must be written as the leftmost character in the PICTURE. The 'S' is not counted in determining the size (in terms of standard data format characters) of the elementary item unless the entry is subject to a SIGN clause which specifies the optional SEPARATE CHARACTER phrase. (See the SIGN Clause in this Section.)

V - The '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 size 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 '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 'Z' in a character-string may only be used to represent the leftmost numeric character positions which will be replaced by a space character when the contents of that character position is zero. Each 'Z' is counted in the size of the item.

9 - Each '9' in the character-string represents a character position which contains a numeral and is counted in the size of the item.

0 - Each '0' (zero) in the character-string represents a character position into which the numeral zero will be inserted. The '0' is counted in the size of the item.

/ - Each '/' (stroke) in the character-string represents a character position into which the stroke character will be inserted. The '/' is counted in the size of the item.

, - Each ',' (comma) in the character-string represents a character position into which the character ',' will be inserted. This character position is counted in the size of the item. The insertion character ',' must not be the last character in the PICTURE character-string.

. - When the character '.' (period) appears in the character-string it is an editing symbol which represents the decimal point for alignment purposes and in addition, represents a character position into which the character '.' will be inserted. The character '.' is counted in the size 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 wherever they appear in a PICTURE clause. The insertion character '.' must not be the last character in the PICTURE character-string.

+, -, CR, DB -
These symbols are used as editing sign control symbols. When used, they represent the character position 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 size of the data item.

* - '*' (asterisk) 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 size of the item.

cs- 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 currency sign or by the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the size of the item.

Editing Rules

There are two general methods of performing editing in the PICTURE clause, either by insertion or by suppression and replacement. There are four types of insertion editing available. They are:

* Simple insertion
* Special insertion
* Fixed insertion
* Floating insertion

There are two types of suppression and replacement editing:

* Zero suppression and replacement with spaces
* Zero suppression and replacement with asterisks

The type of editing which may be performed upon an item is dependent upon the category to which the item belongs. Table 3-1 specifies which type of editing may be performed upon a given category.

Table 3-1. Editing Types for Data Categories.

CATEGORY	TYPE OF EDITING
Alphabetic	Simple insertion 'B' only
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple insertion '0', 'B' and '/'
Numeric Edited	All, but see NOTE below

NOTE

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.

Simple Insertion Editing

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

Special Insertion Editing

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 size 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 disallowed. 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

Fixed Insertion Editing. The currency symbol and the editing sign control symbols, '+', '-', 'CR', '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 size of the item and they must represent the right most character positions that are counted in the size of the item. The symbol '+' or '-', when used, must be either the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character.

Table 3-2. Editing Symbols in PICTURE Character-Strings.

EDITING SYMBOL IN PICTURE CHARACTER-STRING	RESULT	
	DATA ITEM POSITIVE OR ZERO	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 floating insertion characters and as such are mutually exclusive 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 floating insertion characters. This string of floating insertion characters may contain any of the fixed insertion symbols or have fixed insertion characters immediately to the right of this string. These simple insertion characters are part of the floating string.

The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Non-zero numeric data may replace all the characters at or to the right of this limit.

In a 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 on 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.

If the insertion characters are only to the left of the decimal point in the PICTURE character-string, the result is that a single floating insertion character will be placed into the character position immediately preceding either the decimal point or the first non-zero digit in the data represented by the insertion symbol string, whichever is farther to the left in the PICTURE character-string. The character positions preceding the insertion character are replaced with spaces.

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 non-floating insertion characters being edited into the receiving data item, plus one for the floating insertion character.

Zero Suppression Editing

The suppression of leading zeros in numeric character positions is indicated by the use of the alphabetic 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 size of the item. If 'Z' is used, the replacement character will be the space and if the asterisk is used, the replacement character will be '*'.

Zero suppression and replacement is 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 any or all of the leading numeric character positions to the left of the decimal point by suppression symbols. 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 non-zero 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 and the suppression symbol is 'Z', the entire data item will be spaces. If the value is zero and the suppression symbol is '*', the data item will be all '*' except for the actual decimal point.

The symbols '+', '-', '*', 'Z', and the currency symbol, when used as floating replacement characters, are mutually exclusive within a given character-string.

Precedence Rules

Table 3-3 shows the order of precedence when using characters 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.

Table 3-3. PICTURE Character Precedence Chart.

Second Symbol	First Symbol →	Non-Floating Insertion Symbols									Floating Insertion Symbols						Other Symbols					
		B	0	/	,	.	{+ -}	{+ -}	{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		x
	0	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	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	x
	cs						x															
Floating Insertion Symbols	{Z *}	x	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	x						x		x
	cs	x	x	x	x		x								x							
	cs	x	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		x
	A X	x	x	x													x	x				
	S																					
	V	x	x	x	x		x			x	x		x		x		x		x		x	
	P	x	x	x	x		x			x	x		x		x		x		x		x	
	P						x			x									x	x		x

In Table 3-3, 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 represents its use to the left of the decimal point position. The second appearance of symbol in the row and column represents its use to the right of the decimal point position.

THE REDEFINES CLAUSE

Function

The REDEFINES clause allows the same computer storage area to be described by different data description entries.

General Format

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

> NOTE
>
> Level-number, data-name-1 and the semi-colon are shown in the above format to improve clarity. Level-number and data-name-1 are not part of the REDEFINES clause.

Syntax Rules

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

2. The level-numbers of data-name-1 and data-name-2 must be identical but must not be 66 or 88.

3. This clause must not be used in level 01 entries in the File Section. (See General Rule 2 of the DATA RECORDS Clause in Section 5.)

4. This clause must not be used in level 01 entries in the Communication Section.

5. Data-name-2 may be subordinate to an entry which contains a DEDEFINES clause. However data-name-2 may be subordinate to an item whose data description entry contains an OCCURS clause. In this case, the reference to data-name-2 in the REDEFINES clause may not be subscripted or indexed. Neither the original definition nor the redefinition can include an item whose size is variable as defined in the OCCURS clause. (See the OCCURS Clause in Section 4.)

6. No entry having a level-number numerically lower than the level-number of data-name-2 and data-name-1 may occur between the data description entries of data-name-2 and data-name-1.

General Rules

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

2. When the level-number of data-name-1 is other than 01, it must specify the same number of character positions that the data item referenced by data-name-2 contains. It is important to observe that the REDEFINES clause specifies the redefinition of a storage area, not of the data items occupying the area.

3. Multiple redefinitions of the same character positions are permitted. The entries giving the new descriptions of the character positions must follow the entries defining the area being redefined, without intervening entries that define new character positions. Multiple redefinitions of the same character positions must all use the data-name of the entry that originally defined the area.

4. The entries giving the new description of the character positions must not contain any VALUE clauses except in condition-name entries.

5. Multiple level 01 entries subordinate to any given level indicator represent implicit redefinitions of the same area.

THE RENAMES CLAUSE

Function

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

General Format

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

NOTE

Level-number 66, data-name-1 and the semicolon are shown in the above format to improve clarity. Level-number and data-name-1 are not part of the RENAMES clause.

Syntax Rules

1. All RENAMES entries referring to data items within a given logical record must immediately follow the last data description entry of the associated record description entry.

2. Data-name-2 and data-name-3 must be names of elementary items or groups of elementary items in 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 entry.

3. Data-name-1 cannot be used as a qualifier, and can only be qualified by the names of the associated level 01, FD, CD or SD entries. Neither data-name-2 nor data-name-3 may have an OCCURS clause in its data description entry nor be subordinate to an item that has an OCCURS clause in its data description entry. (See the OCCURS Clause in Section 4.)

4. 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 be the right of the end of the area described by data-name-2. Data-name-3, therefore, cannot be subordinate to data-name-2.

5. Data-name-2 and data-name-3 may be qualified.

6. The words THRU and THROUGH are equivalent.

7. None of the items within the range, including data-name-2 and data-name-3, if specified, can be an item whose size is variable as defined in the OCCURS Clause in Section 4.

General Rules

1. One or more RENAMES entries can be written for a logical record.

2. 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 an elementary item or the last elementary item in data-name-3 (if data-name-3 is a group item).

3. 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 SIGN CLAUSE

Function

The SIGN clause specifies the position and the mode of representation of the operational sign when it is necessary to describe these properties explicitly.

General Format

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

Syntax Rules

1. The SIGN clause may be specified only for a numeric data description entry whose PICTURE contains the character 'S', or a group item containing at least one such numeric data description entry.

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

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

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

General Rules

1. The optional SIGN clause, if present, specifies the position and the mode of representation of the operational sign for the numeric data description entry to which it applies, or for each numeric data description entry subordinate to the group to which it applies. The SIGN clause applies only to numeric data description entries whose PICTURE contains the character 'S'; the 'S' indicates the presence of, but neither the representation nor, necessarily, the position of the operational sign.

2. A numeric data description entry whose picture contains the character 'S', but to which no optional SIGN clause applies, has an operational sign, but neither the representation nor, necessarily, the position of the operational sign is specified by the character 'S'. In this (default) case, General Rules 3 through 5 do not apply to such signed numeric data items.

3. If the optional SEPARATE CHARACTER phrase is not present, then:

   a. The operational sign will be presumed to be associated with the leading (or, respectively, trailing) digit position of the elementary numeric data item.

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

4. If the optional SEPARATE CHARACTER phrase is present, then:

   a. The operational sign will be presumed to be the leading (or, respectively, trailing) character position of the elementary numeric data item; this character position is not a digit position.

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

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

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

THE SYNCHRONIZED CLAUSE

Function

The SYNCHRONIZED clause specifies the alignment of an elementary item on the natural boundaries of the computer memory.

General Format

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

Syntax Rules

1. This clause may only appear with an elementary item.

2. SYNC is an abbreviation for SYNCHRONIZED.

General Rules

1. The SYNCHRONIZED clause is accepted for documentation purposes only.

2. This clause specifies that the subject data item is to be aligned in the computer such that no other data item occupies any of the character positions between the leftmost and rightmost natural boundaries delimiting this data item. If the number of character positions required to store this data item is less than the number of character positions between those natural boundaries, the unused character positions (or portions thereof) must not be used for any other data item. Such unused character positions, however, are included in:

   a. The size of any group item(s) to which the elementary item belongs; and

   b. The character positions redefined when this data item is the object of a REDEFINES clause.

3. SYNCHRONIZED not followed by either RIGHT or LEFT specifies that the elementary item is to be positioned between natural boundaries in such a way as to effect efficient utilization of the elementary data item.

4. SYNCHRONIZED LEFT specifies that the elementary item is to be positioned such that it will begin at the left character position of the natural boundary in which the elementary item is placed.

5. SYNCHRONIZED RIGHT specifies that the elementary item is to be positioned such that it will terminate on the right character position of the natural boundary in which the elementary item is placed.

6. Whenever a SYNCHRONIZED item is referenced in the source program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size, such as justification, truncation or overflow.

7. If the data description of an item contains the SYNCHRONIZED clause and an operational sign, the sign of the item appears in the normal operational sign position, regardless of whether the item is SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT.

8. When the SYNCHRONIZED clause is specified in a data description entry of a data item that also contains an OCCURS clause, or in a data description entry of a data item subordinate to a data description entry that contains an OCCURS clause, then:

   a. Each occurrence of the data item is SYNCHRONIZED.

   b. Any implicit FILLER generated for other data items within that same table are generated for each occurence of those data items.

9. This clause is hardware dependent.

THE USAGE CLAUSE

Function

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

General Format

$$[\text{USAGE IS}] \left\{ \begin{array}{l} \underline{\text{COMPUTATIONAL}} \\ \underline{\text{COMP}} \\ \underline{\text{DISPLAY}} \\ \underline{\text{COMPUTATIONAL-3}} \\ \underline{\text{COMP-3}} \end{array} \right\}$$

Syntax Rules

1. The PICTURE character-string of a COMPUTATIONAL or COMPUTATIONAL-3 item can contain only '9's, the operational sign character 'S', the implied decimal point character 'V', one or more 'P's. (See the PICTURE Clause earlier in this Section).

2. COMP is an abbreviation for COMPUTATIONAL.

General Rules

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

2. This clause specifies the manner in which a data item is represented in the storage of a computer. It does not affect the use of the data item, although the specifications for some statements in the Procedure Division may restrict the USAGE clause of the operands referred to. The USAGE clause may affect the radix or type of character representation of the item.

3. A COMPUTATIONAL or COMPUTATIONAL-3 item is capable of representing a value to be used in computations and must be numeric. If a group item is described as COMPUTATIONAL(-3), the elementary items in the group are COMPUTATIONAL(-3). The group item itself is not COMPUTATIONAL(-3) and cannot be used in computations.

4. The USAGE IS DISPLAY clause indicates that the format of the data is a standard data format.

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

6. Space requirements for the various USAGE storage options are given under SELECTION OF CHARACTER REPRESENTATION AND RADIX in Section 2.

THE VALUE CLAUSE

Function

The VALUE clause defines the value of constants, the initial value of working storage items, and the values associated with a conditional name.

General Format

Format 1:

VALUE is literal

Format 2:

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

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

Syntax Rules

1. The VALUE clause cannot be stated for any items whose size is variable. (See the OCCURS Clause in Section 4.)

2. A signed numeric literal must have associated with it a signed numeric PICTURE character-string.

3. All numeric literals in a VALUE clause of an item must have values which are within the range of values indicated by the PICTURE clause, and must not have a value which would require truncation of nonzero digits. Nonnumeric literals in a VALUE clause of an item must not exceed the size indicated by the PICTURE clause.

4. The words THRU and THROUGH are equivalent.

General Rules

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

   a. If the category of the item is numeric, all literals in the VALUE clause must be numeric. If the literal defines the value of a working storage item, the literal is aligned in the data item according to the standard alignment rules. (See STANDARD ALIGNMENT RULES in Section 2.)

b. If the category of the item is alphabetic, alphanumeric, alphanumeric edited or numeric edited, all literals in the VALUE clause must be nonnumeric literals. The literal is aligned in the data item as if the data item had been described as alphanumeric. (See STANDARD ALIGNMENT RULES in Section 2.) Editing characters in the PICTURE clauses are included in determining the size of the data item (see the PICTURE Clause earlier in this Section) but have no effect on initialization of the data item. Therefore, the VALUE for an edited item is presented in an edited form.

c. Initialization takes place independent of any BLANK WHEN ZERO or JUSTIFIED clause that may be specified.

Condition-Name Rules

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

2. Format 2 can be used only in connection with condition-names. Wherever the THRU phrase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.

Data Description Entries Other Than Condition-Names

Rules governing the use of the VALUE clause differ with the respective sections of the Data Division:

1. In the File Section, the VALUE clause may be used only in condition-name entries.

2. In the Working-Storage Section, the value clause must be used in condition-name entries. The VALUE clause may also be used to specify the initial value of a data item; in which case the clause causes the item to assume the specified value at the start of the object program. If the VALUE clause is not used in an item's description, the initial value is undefined.

3. In the Linkage Section, the VALUE clause may be used only in condition-name entries.

4. The VALUE clause must not be stated in a data description entry that contains an OCCURS clause, but not in an entry that is subordinate to an entry containing an OCCURS clause. (See the OCCURS Clause in Section 4.)

5. The VALUE clause may be stated in a data description entry that contains a REDEFINES clause, or in an entry that is subordinate to an entry containing a REDEFINES clause.

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

7. The VALUE clause must not be written for a group containing items with descriptions, including JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE IS DISPLAY).

ARITHMETIC EXPRESSIONS

Definition of an Arithmetic Expression

An arithmetic expression can be an identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses. Any arithmetic expression may be preceded by a unary operator. The permissible combinations of variables, numeric literals, arithmetic operator and parentheses are given in Table 3-4, Combination of Symbols in Arithmetic Expressions.

Those identifiers and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic may be performed.

Arithmetic Operators

There are five binary arithmetic operators and two unary arithmetic operators that may be used in arithmetic expressions. They are represented by specific characters that may be preceded by a space and followed by a space.

Binary Arithmetic Operators	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Unary Arithmetic Operators	Meaning
+	The effect of multiplication by numeric literal +1
-	The effect of multiplication by numeric literal -1.

Formation and Evaluation Rules

1. Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first, and within nested 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 execution is implied:

1st - Unary plus and minus
2nd - Exponentiation
3rd - Multiplication and division
4th - Addition and subtraction

2. Parentheses are used either to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear or to modify the normal hierarchical sequence of execution in expressions 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.

3. The ways in which operators, variables, and parentheses may be combined in an arithmetic expression are summarized in Table 3-4, where:

   a. The letter 'P' indicates a permissible pair of symbols.

   b. The character '-' indicates an invalid pair.

   c. 'Variable' indicates an identifier or literal.

FIRST SYMBOL	SECOND SYMBOL				
	Variable	* / ** - +	Unary + or -	(	)
Variable	-	P	-	-	P
* / ** + -	P	-	P	P	-
Unary + or -	P	-	-	P	-
(	P	-	P	P	-
)	-	P	-	-	P

Table 3-4. Combination of Symbols in Arithmetic Expressions.

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

5. Arithmetic expressions allow the user to combine arithmetic operations without the restrictions on composite of operands and/or receiving data items. See, for example, Syntax Rule 3 of the ADD Statement in this Section.

CONDITIONAL EXPRESSIONS

Conditional expressions identify conditions that are tested to enable the object program to select between alternate paths of control depending upon the truth value of the condition. Conditional expressions are specified in the IF, PERFORM and SEARCH statements. There are two categories of conditions associated with conditional expressions: simple conditions and complex conditions. Each may be enclosed with any number of paired parentheses, in which case its category is not changed.

Simple Conditions

The simple conditions are the relation, class, condition-name, switch-status, and sign conditions. A simple condition has a truth value of 'true' or 'false'. The inclusion in parentheses of simple conditions does not change the simple truth value.

Relation Condition

A relation condition causes a comparison of two operands, each of which may be the data item referenced by an identifier, a literal or the value resulting from an arithmetic expression. A relation condition has a truth value of 'true' if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses. However, for all other comparisons the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply.

The general format of a relation condition is as follows:

```
                           IS [NOT]   GREATER THAN
 identifier-1              IS [NOT]   LESS THAN      identifier-2
 literal-1                 IS [NOT]   EQUAL TO       literal-2
 arithmetic-expression-1   IS [NOT]   >              arithmetic-
                           IS [NOT]   <              expression-2
                           IS [NOT]   =
```

> NOTE
>
> The required relational characters '< ', ' >', and '=' are not underlined to avoid confusion with other symbols such as '$\geq$' (Greater than or equal to.)

The first operand (identifier-1, literal-1 or arithmetic-expression-1) is called the subject of the condition; the second operand (identifier-2 or literal-2 or arithmetic-expression-2) is called the object of the condition. The relation condition must contain at least one reference to a variable.

The relational operator specifies the type of comparison to be made in a relation condition. A space must precede and follow each reserved word comprising the relational operator. When used, 'NOT' and the next key word or relation character are one relational operator that defines the comparison to be executed for truth value; e.g., 'NOT EQUAL' is a truth test for an 'unequal'. Comparison; 'NOT GREATER' is a truth test for an 'equal' or 'less' comparison. The meaning of the relational operators is as shown in Table 3-5.

Table 3-5. Relational Operators.

Meaning	Relational Operator
Greater than or not greater than	IS NOT GREATER THAN IS NOT >
Less than or not less than	IS NOT LESS THAN IS NOT <
Equal to or not equal to	IS NOT EQUAL TO IS NOT =
The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols such as '$\geq$' (Greater than or equal to).	

Comparison of Numeric Operands: For operands whose class is numeric a comparison is made with respect to the algebraic value of the operands. The length of the literal or arithmetic expression operands in terms of number of digits represented, 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 comparison.

Comparison of Nonnumeric Operands: For nonnumeric operands, or one numeric and one nonnumeric operand, a comparison is made with respect to a specified collating sequence of characters (see the OBJECT-COMPUTER Paragraph in this Section). If one of the operands is specified as numeric, it must be an integer data item or an integer literal and:

1. If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it were moved to an elementary alphanumeric data item of the same size as the numeric data item (in terms of standard data format characters), and the contents of this alphanumeric data item were then compared to the nonnumeric operand. (See the MOVE Statement in this Section, and the PICTURE Character 'P' under the heading Symbols Used earlier in this Section.)

2. If the non-numeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item (in terms of standard data format characters), and the contents of this group item were then compared to the nonnumeric operand. (See the MOVE Statement in this Section, and the PICTURE character 'P' under the Heading Symbols Used earlier in this Section.)

3. A non-integer numeric operand cannot be compared to a nonnumeric operand.

The size of an operand is the total number of standard data format characters in the operand. Numeric and nonnumeric operands may be compared only when their usage is the same.

There are two cases to consider:

1. Operands of equal size - If the operands are of equal size, comparison effectively proceeds by comparing characters in corresponding character positions starting from the high order end and continuing until either a pair of unequal characters is encountered or the low order end of the operand is reached, whichever comes first. The operands are determined to be equal if all pairs of characters compare equally through the last pair, when the low order end is reached.

   The first encountered pair of unequal characters is compared to determine their relative position in the collating sequence. The operand that contains the character that is positioned higher in the collating sequence is considered to be the greater operand.

2. Operands of unequal size - If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

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 the operational sign, or alphabetic, that is, consists entirely of the characters 'A', 'B', 'C', ..., 'Z', space. The general format for the class condition is as follows:

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

The usage of the operand being tested must be described as display. When used, 'NOT' and the next key word specify one class condition that defines the class test to be executed for truth value; e.g. 'NOT NUMERIC' is a truth test for determining that an operand is nonnumeric.

The NUMERIC test cannot be used with an item whose data description describes the item as alphabetic or as a group item composed of elementary items whose data description indicates the presence of operational sign(s). If the data description of the item being tested does not indicate the presence of 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. If the data description of the item does indicate the presence of an operational sign, the items described with the SIGN IS SEPARATE clause are the standard data format characters, '+' and '-'.

The ALPHABETIC test cannot be used with an item whose data 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' through 'Z' and the space.

Condition-Name Condition (Conditional Variable)

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 format for the condition-name condition is as follows:

```
          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.

Switch-Status Condition

A switch-status condition determines the 'on' or 'off' status of an implementor-defined switch. The implementor-name and the 'on' or 'off' value associated with the condition must be named in the SPECIAL-NAMES paragraph of the Environment Division. The genaral format for the switch-status condition is as follows:

```
          condition-name
```

The result of the test is true if the switch is set to the specified position corresponding to the condition-name.

Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than or equal to zero. The general format for a sign condition is as follows:

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

When used, 'NOT' and the next key word specify one sign condition that defines that algebraic test to be executed for truth value; e.g., 'NOT ZERO' is a truth test for a nonzero (positive or negative) value. 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. The arithmetic expression must contain at least one reference to a variable.

Complex Conditions

A complex condition is formed by combining simple conditions, combined conditions and/or complex conditions with logical connectors (logical operators 'AND' and 'OR') or negating these conditions with logical negation (the logical operator 'NOT'). The truth value of a complex condition, whether parenthesized or not, is that truth value which results from the interaction of all the stated logical operators on the individual truth values of simple conditions, or the intermediate truth values of conditions logically connected or logically negated.

The logical operators and their meanings are:

Logical Operator	Meaning
AND	Logical conjunction; the truth value is 'true' if both of the conjoined conditions are true; 'false' if one or both of the conjoined conditions is false.
OR	Logical inclusive OR; the truth value if 'true' if one or both of the included conditions is true; 'false' if both included conditions are false.
NOT	Logical negation or reversal of truth value; the truth value is 'true' if the condition is false; 'false' if the condition is true.

The logical operators must be preceded by a space and followed by a space.

Negated Simple Conditions: A simple condition is negated through the use of the logical operator 'NOT'. The negated simple condition effects the opposite truth value for a simple condition. Thus the truth value of a negated simple condition is 'true' if and only if the truth value of the simple condition is 'false', the truth value of a negated simple condition is 'false' if and only if the truth value of the simple condition is 'true'. The inclusion in parentheses of a negated simple condition does not change the truth value.

The general format for a negated simple condition is:

NOT simple-condition

Combined and Negated Combined Conditions: A combined condition results from connecting conditions with one of the logical operators 'AND'or 'OR'. The general format of a combined condition is:

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

where 'condition' may be:

a. A simple condition, or

b. A negated simple condition, or

c. A combined condition,

d. A negated combined condition; i.e, the 'NOT' logical operator followed by a combined condition enclosed within parentheses, or

e. **Combinations of the above, specified according to the rules summarized in Table 2 on page 3-6, Combinations of Conditions, Logical Operators, and Parentheses, located on the next page.**

Although parentheses need never be used when either 'AND' or 'OR' (but not both) is used exclusively in a combined condition, parentheses may be used to effect a final truth value when a mixture of 'AND', 'OR' and 'NOT' is used. (See Table 3-6, Combinations of Conditions, Logical Operators, and Parentheses below, and Condition Evaluation Rules earlier in this Section.)

Table 3-6 on the next page indicates the ways in which conditions and logical operators may be combined and parenthesized. There must be a one-to-one correspondence between left and right parentheses such that each left parentheses is matched by a corresponding right parentheses. The table assumes a left to right sequence of elements.

Table 3-6. Combinations of Conditions, Logical Operators, and Parentheses

Element	Permitted Location in conditional expression	Element can be preceded by only:	Element can be followed by only:
simple-condition	Any	OR, NOT, AND, (	OR, AND,)
OR, or AND	Not first or last	simple-condition,)	simple-condition NOT, (
NOT	Not last	OR, AND, (	simple-condition, (
(	Not last	OR, NOT, AND, (	simple-condition, NOT, (
)	Not first	simple-condition,)	OR, AND,)

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

Abbreviated Combined Relation Conditions

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

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

The format for an abbreviated combined relation condition is:

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

Within a sequence of relation conditions both of the above forms of abbreviation may be used. The effect of using such abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. The result of such implied insertion must comply with the rules of Table 3-6, Combinations of Conditions, Logical Operators, and Parentheses. This insertion of an omitted subject and/or relational operator terminates once a complete simple condition is encountered within a complex condition.

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

1. If the word immediately following 'NOT' is 'GREATER', ' > ', 'LESS', ' < ', 'EQUAL', '=', then the 'NOT' participates as part of the relational operator; otherwise,

2. The 'NOT' is interpreted as a logical operator and, therefore, the implied insertion of subject or relational operator results in a negated relation condition.

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

Abbreviated Combined Relation Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a >b) AND (a NOT< c)) OR (a NOT< d)
a NOT EQUAL b OR c	(a NOT EQUAL b) OR (a NO EQUAL c)
NOT a = b OR c	(NOT (a = b)) OR (a = c)
NOT (a GREATER b OR c)	NOT ((a GREATER b) OR (a< c))
NOT (a GREATER b OR< c)	NOT ((a GREATER b) OR (a >c))
NOT (a NOT >b AND c AND NOT d)	NOT ((((a NOT b) AND (a NOT c)) AND (NOT (a NOT >d))))

Condition Evaluation Rules

Parentheses may be used to specify the order in which individual conditions of complex conditions are to be evaluated when it is necessary to depart from the implied evaluation precedence. Conditions within parentheses are evaluated first, and within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When parentheses are not used, or parenthesized conditions are at the same level of inclusiveness, the following hierarchical order of logical evaluation is implied until the final truth value is determined:

1. Values are established for arithmetic expressions. (See FORMATION AND EVALUATION RULES under ARITHMETIC EXPRESSIONS in this Section.)

2. Truth values for simple conditions are established in the following order:

   relation (following the expansion of any abbreviated relation condition)

      class
      condition-name
      switch-status
      sign

3. Truth values for negated simple conditions are established.

4. Truth values for combined conditions are established:

   'AND' logical operators, followed by
   'OR' logical operators.

5. Truth values for negated combined conditions are established.

6. When the sequence of evaluation is not completely specified by parenthese the order of evaluation of consecutive operations of the same hierarchic level is from left to right.

COMMON PHRASES AND GENERAL RULES FOR STATEMENT FORMATS

In the statement descriptions that follow, several phrases appear frequently: the ROUNDED phrase, the SIZE ERROR phrase and the CORRESPONDING phrase.

These are described below; a resultant-identifier is that identifier associated with a result of an arithmetic operation.

The ROUNDED Phrase

If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When rounding is requested, the absolute value of the resultant-identifier is increased by one whenever the most significant digit of the excess is greater than or equal to five.

When the low-order integer positions in a resultant-identifier are represented by the character 'P' in the PICTURE for the resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

The SIZE ERROR Phrase

If, after decimal point alignment, the absolute value of a result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. Division by zero always causes a size error condition. The size error condition applies only to the final results, except in MULTIPLY and DIVIDE statements, in which case the size error condition applies to the intermediate results as well. If the ROUNDED phrase is specified, rounding takes place before checking for size error. When such a size error condition occurs, the subsequent action depends on whether or not the SIZE ERROR phrase is specified as follows:

SIZE ERROR Phrase Not Specified

When a size error condition occurs, the value of those resultant-identifier(s) affected is undefined. Values of resultant-identifier(s) for which no size error condition occurs are unaffected by size errors that occur for other resultant-identifier(s) during execution of this operation.

SIZE ERROR Phrase Specified

When a size error condition occurs, then the values of resultant-identifier(s) affected by the size errors are not altered. Values of resultant-identifier(s) for which no size error condition occurs are unaffected by size errors that occur for other resultant-identifier(s) during execution of this operation. After completion of the execution of this operation, the imperative statement in the SIZE ERROR phrase is executed.

For the ADD statement with the CORRESPONDING phrase and the SUBTRACT statement with the CORRESPONDING phrase, if any of the individual operations produces a size error condition, the imperative statement in the SIZE ERROR phrase is not executed until all of the individual additions or subtractions are completed.

The CORRESPONDING Phrase

In the text that follows d_1 and d_2 must each be identifiers that refer to group items. A pair of data items, one from d_1 and one from d_2 correspond if the following conditions exist:

1. A data item in d_1 and a data item in d_2 are not designated by the key word FILLER and have the same data-name and the same qualifiers up to, but not including, d_1 and d_2.

2. At least one of the data items is an elementary data item in the case of a MOVE statement with the CORRESPONDING phrase; and both of the data items are elementary numeric data items in the case of the ADD statement with the CORRESPONDING phrase or the SUBTRACT statement with the CORRESPONDING phrase.

3. The description of d_1 and d_2 must not contain level-number 66, 77, or 88 or the USAGE IS INDEX clause.

4. A data item that is subordinate to d_1 or d_2 and contains a REDEFINES, RENAMES, OCCURS or USAGE IS INDEX clause is ignored, as well as those data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause. However, d_1 and d_2 may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

Arithmetic Statements

The arithmetic statements are the ADD, DIVIDE, MULTIPLY, and SUBTRACT statements. Common features are as follows:

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

2. The maximum size of each operand is 18 decimal digits. The composite of operands, which is a hypothetical data item resulting from the superimposition of specified operands in a statement aligned on their decimal points (See the ADD Statement, the DIVIDE Statement, the MULTIPLY Statement and the SUBTRACT Statement later in this Section) must not contain more than 18 decimal digits.

Overlapping Operands

When a sending and a receiving item in an arithmetic statement or an INSPECT, MOVE, SET statement share a part of their storage areas, the result of the execution of such a statement is undefined.

Multiple Results in Arithmetic Statements

The ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements may have multiple results. Such statements behave as though they had been written in the following way:

1. A statement which performs all arithmetic necessary to arrive at the result to be stored in the receiving items, and stores that result in a temporary storage location.

2. A sequence of statements transferring or combining the value of this temporary location with a single result. These statements are considered to be written in the same left-to-right sequence that the multiple results are listed.

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.

Incompatible Data

Except for the class condition (See Class Condition in this Section), when the contents of a data item are referenced in the Procedure Division and the contents of that data item are not compatible with the class specified for that data item by its PICTURE clause, then the result of such a reference is undefined.

THE ACCEPT STATEMENT

Function

The ACCEPT statement causes data keyed at the keyboard to be made available to the program in a specified data item.

General Formats

Format 1

```
ACCEPT identifier     [ FROM { mnemonic-name } ]
                      [      { CONSOLE       } ]
```

Format 2

```
ACCEPT data-name-1    [ AT   { data-name-2 } ]     FROM CRT
                      [      { literal-1   } ]
```

Format 3

```
                          { DATE }
ACCEPT identifier FROM    { DAY  }
                          { TIME }
```

General Rules

1. Format 1 is the standard ANSI ACCEPT statement.

   Format 2 is the extended ACCEPT format.

The two formats are distinguished by their FROM phrases and the default assumes FROM CONSOLE. A user-defined mnemonic-name can be used if this is associated to a system device in the SPECIAL-NAMES paragraph (see the SPECIAL-NAMES Paragraph earlier in this Section). The default can, however, be changed by specifying CONSOLE IS CRT in the SPECIAL-NAMES clause so that FROM CRT becomes the default. This changed default is not shown in the syntax above.

Format 1

2. The ACCEPT statement reads one line of input data from the keyboard. This input data replaces the contents of the data item named by the identifier.

3. The line of input is echoed as it is typed. The line is terminated by pressing RETURN or by exceeding 80 or 132 characters in length, depending upon the video display width.

4. If the input line is of the same size as the receiving data item, the transferred data is stored in the receiving data item.

5. If the input line is not of the same size as the receiving data item, then:

   a. If the size of the receiving data item exceeds the size of the input line, the transferred data is stored aligned to the left in the receiving data item and the data item is filled with trailing spaces.

   b. If the size of the transferred data exceeds 120 bytes, only the first 120 characters of the input line are stored in the receiving data item. The remaining characters of the input line which do not fit into the receiving data item are ignored.

Format 2

6. The ACCEPT statement causes the transfer of data from the CRT to data-name-1. The contents of data-name-1 are replaced by this data.

7. Data-name-1 is taken as a definition of the screen area in which elementary data items correspond to areas on the screen into which the operator can key. FILLER fields correspond to areas on the screen which are inaccessible to the operator.

8. Elementary data items within data-name-1 may be alphanumeric, integer numeric, numeric or edited. Numeric items are treated as two separate integer numeric fields, and edited fields are treated as alphanumeric fields except as described in Rule 16.

9. AT data-name-2 or literal-1 defines the position on the screen of the leftmost character of the data. Either form must refer to a PIC 9999 field. The most significant 99 is taken as a line count in the range one to the maximum lines on the user screen. The least significant 99 is taken as a character position in the range one to the maximum positions allowed by the screen width of the user CRT.

10. Data-name-1 may refer to a record, group or elementary item, but it may not be subscripted. REDEFINES may be used within data-name-1, in which case the first description of the data is used and subsequent descriptions are ignored. OCCURS and nested OCCURS may also be used with the effect that the repeated data-item is expanded into the full number of times it occurs and one definition is thus automatically repeated for many fields.

11. Immediately upon execution of the ACCEPT statement a cursor is displayed in the CRT location corresponding to the leftmost non-FILLER character position in data-name-1. Alternatively, when CURSOR is specified in the SPECIAL-NAMES paragraph, the cursor displays at the position held in the CURSOR data-name. The CURSOR position is stored in CURSOR data-name in

the same format as the screen position is held in data-name-2. If the data-name-2 has the value SPACE or ZERO, the effect is as if CURSOR was not specified; if a valid screen position is specified that is not within a non-FILLER item, the cursor is positioned to the next non-FILLER character position. The CURSOR data-name holds the last cursor position at the end of execution of an ACCEPT statement.

12. If FROM CRT is not specified, the default is FROM CONSOLE (see Rule 1 above).

13. As the operator keys characters, the cursor moves to the right one character position at a time in locations corresponding to data fields. The operator always keys into the current cursor position. At the end of a line the cursor moves down one line and to the leftmost non-FILLER character position.

14. If the data item is integer numeric, only numeric character (0-9) will be accepted into that item. Keying the decimal point character (. or , as specified in the DECIMAL POINT phrase) when accepting a numeric item causes the item to be right justified and zero-filled from the left.

15. When the cursor location reaches a position corresponding to a FILLER item in a data-name, it immediately skips to the next non-FILLER character position, or if there is no such position remaining in the portion of the CRT specified by the data-name, it remains in its current position.

16. The operator can terminate input by pressing the GO key at which time control is passed to the next statement after ACCEPT. Before control is passed to the next statement the following takes place:

    a. The numeric value of each numeric-edited data-field is formed internally from only the keyed characters 0 to 9, +, -, . or , and then moved back to the numeric-edited field with the ANSI PICTURE editing applied. The field may thus be different to that shown on the CRT just before the Carriage Return key was pressed.

    b. When CURSOR IS is specified in the SPECIAL-NAMES paragraph, the cursor position when the Carriage Return key is pressed is returned in the data-name specified by the CURSOR IS clause, except when its value at the start of the ACCEPT function caused it to be treated as unspecified.

17. Before keying GO, the operator can reposition the cursor to overwrite data already keyed or to skip character positions by use of the cursor positioning keys as shown in Table 3-7.

NOTE

The actual key identification and functions shown in this table varies according to the CRT used and the way it is configured (see Appendix J).

Table 3-7. Cursor Repositioning Keys.

KEY:	FUNCTION:
←	Backs up the cursor one position.
↑	Backs up the cursor to the start of the non-FILLER field prior to the current cursor position.
↓	Moves the cursor on to the start of the next non-FILLER field in advance of the current cursor position.
→	Moves the cursor on one position without overwriting existing contents.
PREV PAGE	Moves the cursor back to the start of the first non-FILLER field in the CRT area corresponding to data-name-1.

Format 3

18. The ACCEPT statement causes the information requested to be transferred to the data item specified by identifier according to the rules of the MOVE statement. DATE, DAY, and TIME are conceptual data items and, therefore, are not described in the COBOL program.

19. DATE is composed of the data elements year of century, month of year, and day of month. The sequence of the data element codes shall be from high to low order (left to right), year of century, month of year, and day of month accessed by a COBOL program, behaves as if it had been described in the COBOL program as an unsigned elementary numeric integer data item six digits in length.

20. DAY is composed of the data elements year of century and day of year. The sequence of the data element codes shall be from high order to low order (left to right) year of century, day of year. Therefore, July 1, 1968 would be expressed as 68183. DAY, when accessed by a COBOL program, behaves as if it had been described in a COBOL program as an unsigned elementary numeric integer data item five digits in length.

21. TIME is composed of the data elements hours, minutes, seconds, and hundredths of a second. TIME is based on elapsed time after midnight on a 24-hour clock basis--thus, 2:41 p.m. would be expressed as 14410000. TIME, when accessed by a COBOL program, behaves as if it had been described in a COBOL program as an unsigned elementary numeric integer data item eight digits in length. The minimum value of TIME is 00000000; the maximum value of TIME is 23595900. Hundredths of seconds always have the value zero.

THE ADD STATEMENT

Function

The ADD statement causes two or more numeric operands to be summed and the result to be stored.

General Format

Format 1

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

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

Format 2

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

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

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

Format 3

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

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

Syntax Rules

1. In Formats 1 and 2, each identifier must refer to an elementary numeric item, except that in Format 2 each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item. In Format 3, each identifier must refer to a group item.

2. Each literal must be a numeric literal.

3. The composite of operands must not contain more than 18 digits (see The Arithmetic Statements in this Section).

   a. In Format 1 the composite of operands is determined by using all of the operands in a given statement.

b. In Format 2 the composite of operands is determined by using all of the operands in a given statement excluding the data items that follow the word GIVING.

c. In format 3 the composite of operands is determined separately for each corresponding pair of data items.

General Rules

1. See the ROUNDED PHRASE, the SIZE ERROR PHRASE, the CORRESPONDING PHRASE, ARITHMETIC Statements, OVERLAPPING OPERANDS and MULTIPLE RESULTS IN ARITHMETIC Statements in this Section.

2. If Format 1 is used, the values of the operands preceding the word TO are added together, then the sum is added to the current value of identifier-m storing the result immediately into identifier-m, and repeating this process respectively for each operand following the word TO. Each of the operands following the word TO must be initialized.

3. If Format 2 is used, the value of the operands preceding the word GIVING are added together, then the sum is stored as the new value of each identifier-m, identifier-n, ..., the resultant identifiers.

4. If Format 3 is used, data items in identifier-1 are added to and stored in corresponding data items in identifier-2. These data items in identifier-2 must be initialized.

5. The compiler ensures that enough places are carried so as not to lose any significant digits during execution.

THE ALTER STATEMENT

Function

The ALTER statement modifies a predetermined sequence of operations.

General Format

```
ALTER procedure-name-1 [TO PROCEED TO] procedure-name-2

  [ , procedure-name-3 TO  [PROCEED TO]  procedure-name-4 ] ...
```

Syntax Rules

1. Each procedure-name-1, procedure-name-3, ..., is the name of a paragraph that contains a single sentence consisting of a GO TO statement without the DEPENDING phrase.

2. Each procedure-name-2, procedure-name-4, ..., is the name of a paragraph or section in the Procedure Division.

General Rules

1. Execution of the ALTER statement modifies the GO TO statement in the paragraph named procedure-name-1, procedure-name-3, ..., so that subsequent executions of the modified GO TO statements cause transfer of control to procedure-name-2, procedure-name-4, ..., respectively. Modified GO TO statements in independent segments may, under some circumstances, be returned to their initial states (see INDEPENDENT SEGMENTS in Section 8).

2. A GO TO statement in a section whose segment-number is greater than or equal to 50 must not be referred to by an ALTER statement in a section with a different segment-number.

   All other uses of the ALTER statement are valid and are performed even if procedure-name-1, procedure-name-3 is in an overlayable fixed segment.

THE COMPUTE STATEMENT

Function

The COMPUTE statement assigns to one or more data items the value of an arithmetic expression.

General Format

```
COMPUTE identifier-1        [ROUNDED] [, identifier-2        [ROUNDED]]
-------                      -------                          -------
= arithmetic-expression        [; ON SIZE ERROR imperative-statement]
                                     ---- -----
```

Syntax Rules

Identifiers that appear only to the left of = must refer to either an elementary numeric item or an elementary numeric edited item.

General Rules

1. See the ROUNDED PHRASE, the SIZE ERROR PHRASE, the ARITHMETIC Statements, OVERLAPPING OPERANDS and MULTIPLE RESULTS IN ARITHMETIC Statements.

2. An arithmetic expression consisting of a single identifier or literal provides a method of setting the values of identifier-1, identifier-2, etc., equal to the value of the single identifier or literal.

3. If more than one identifier is specified for the result of the operation, that is preceding = , the value of the arithmetic expression is computed, and then this value is stored as the new value of each of identifier-1, identifier-2, etc., in turn.

4. The COMPUTE statement allows the user to combine arithmetic operations without the restrictions on composite of operands and/or receiving data items imposed by the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE.

THE DISPLAY STATEMENT

Function

The DISPLAY statement causes data to be transferred from specified data items to the CRT screen.

General Formats

Format 1

```
           { identifier-1 }  [   { identifier-2 } ]
DISPLAY    {              }  [ , {              } ] ...  [UPON   { mnemonic-name }
           { literal-1    }  [   { literal-2    } ]              { CONSOLE       }
```

Format 2

```
          { data-name-1 }  [    { data-name-2 } ]
DISPLAY   {             }  [ AT {             } ]     UPON    { CRT       }
          { literal-3   }  [    { literal-4   } ]             { CRT-UNDER }
```

Syntax Rules

1. Each literal may be any figurative constant, except ALL.
2. If the literal is numeric, it must be an unsigned integer.
3. literal-3 must be alphanumeric. Literal-4 must be numeric.
4. Data-name-1 may refer to a record, group or elementary item, but it must not be subscripted.

General Rules

1. Format 1 is the standard ANSI DISPLAY statement.

   Format 2 is the extended DISPLAY format.

   The two formats are distinguished by their UPON phrases and the default assumes UPON CONSOLE. A user-defined mnemonic-name can be used if this is associated with a system device in the SPECIAL-NAMES paragraph. (See the SPECIAL-NAMES paragraph in this Section.) The default can, however, be changed by specifying CONSOLE IS CRT in the SPECIAL-NAMES clause so that UPON CRT becomes the default. This changed default is not shown in the syntax above.

Format 1

2. The DISPLAY statement causes the contents of each operand to be transferred to the CRT in the order listed as one line of output data.

3. The size of the data transfer can be up to 132 bytes.

4. If a figurative constant is specified as one of the operands, only a single occurrence of the figurative constant is displayed.

5. If the CRT is capable of displaying data of the same size as the data item being output, the data item is transferred.

6. If the CRT is not capable of displaying data of the same size as the data item being transferred, one of the following applies.

   a. If the size of the data item being displayed exceeds the size of the data that the CRT is capable of receiving in a single transfer, the data beginning with the leftmost character is stored aligned to the left in the receiving CRT.

   b. If the size of the data item that the CRT is capable of receiving exceeds the size of the data being transferred, the transferred data is stored aligned to the left in the receiving CRT.

7. When a DISPLAY statement contains more than one operand, the size of the sending item is the sum of the sizes associated with the operands, and the values of the operands are transferred in the sequence in which the operands are encountered.

Format 2

8. The DISPLAY statement is used to ouput data to the CRT in the screen positions specified.

9. Data-name-1 is taken as a definition of the screen area into which data items that correspond to areas on the screen are moved. FILLER fields correspond to areas on the screen into which data is not moved.

10. Elementary data items within data-name-1 may be alphanumeric, integer numeric, numeric or edited.

11. AT data-name-2 or literal-4 defines the position on the screen of the leftmost character of the data. Either form must refer to a PIC 9999 field. The most significant 99 is taken as a line count in the range one to the maximum number of lines on the user screen. The least significant 99 is taken as a character position in the range one to the maximum of characters per line on the user screen.

12. Data-name-1 may refer to a record, group or elementary item, but it may not be subscripted. REDEFINES may be used, in which case the first description of the data is used and subsequent descriptions are ignored. OCCURS and nested OCCURS may also be used with the effect that the repeated data-item is expanded into the full number of times it occurs and one definition is thus automatically repeated for many fields.

13. DISPLAY SPACE has the effect of clearing the screen at run time (i.e. filling the whole screen with spaces). DISPLAY " " (one space character), however, displays only one space character.

14. The CRT-UNDER phrase causes the elementary items moved to the CRT to be displayed with the underline feature present.

THE DIVIDE STATEMENT

Function

The DIVIDE statement divides one numeric data item into others and sets the values of data items equal to the quotient.

General Format

Format 1

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

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

Format 2

```
DIVIDE    {identifier-1 }      INTO     {identifier-2}
          {literal-1    }               {literal-2   }

     GIVING     identifier-3     [ROUNDED]  [, identifier-4 [ROUNDED]] ...
     [; ON SIZE ERROR imperative-statement]
```

Format 3

```
DIVIDE    {identifier-1}        BY     {identifier-2}
          {literal-1   }               {literal-2   }

     GIVING     identifier-3    [ROUNDED] [, identifier-4      [ROUNDED] ] ...

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

Format 4

```
DIVIDE    {identifier-1}       INTO     {identifier-2}
          {literal-1   }                {literal-2   }

     GIVING     identifier-3    [ROUNDED]

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

Format 5

```
DIVIDE  {identifier-1}      BY      {identifier-2}
        {literal-1   }              {literal-2   }

     GIVING      identifier-3     [ROUNDED]

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

<u>Syntax Rules</u>

1. Each identifier must refer to an elementary numeric item, except that any identifier associated with the GIVING or REMAINDER phrase must refer to either an elementary numeric item or elementary numeric edited item.

2. Each literal must be a numeric literal.

3. The composite of operands, which is the hypothetical data item resulting from the superimposition of all receiving data items (except the REMAINDER data item) of a given statement aligned on their decimal points, must not contain more than eighteen digits.

<u>General Rules</u>

1. See the ROUNDED PHRASE, the SIZE ERROR PHRASE, the ARITHMETIC Statements, OVERLAPPING OPERANDS and MULTIPLE RESULTS IN ARITHMETIC Statements in this Section for a description of these functions.

2. When Format 1 is used, the value of identifier-1 or literal-1 is divided into the value of identifier-2. The value of the dividend (identifier-2) is replaced by this quotient; similarly for identifier-3, etc.

3. When Format 2 is used, the value of identifier-1 or literal-1 is divided into identifier-2 or literal-2 and the result is stored in identifier-3, identifier-4, etc.

4. When Format 3 is used, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2 and the result is stored in identifier-3, identifier-4, etc.

5. Formats 4 and 5 are used when a remainder from the division operation is desired, namely identifier-4. The remainder in COBOL is defined as the result of subtracting the product of the quotient (identifier-3) and the divisor from the dividend. If identifier-3 is defined as a numeric edited item, the quotient used to calculate the remainder is an intermediate field which contains the unedited quotient. If ROUNDED is used, the quotient used to calculate the remainder is an intermediate field which contains the quotient of the DIVIDE statement, truncated rather than rounded.

6. In Formats 4 and 5, the accuracy of the REMAINDER data item (identifier-4) is defined by the calculation described above. Appropriate decimal alignment truncation (not rounding) will be performed for the content of the data item referenced by identifier-4, as needed.

7. When the ON SIZE ERROR phrase is used in Formats 4 and 5, the following rules pertain:

   a. If the size error occurs on the quotient, no remainder calculation is meaningful. Thus, the contents of the data items referenced by both identifier-3 and identifier-4 will remain unchanged.

   b. If the size error occurs on the remainder, the contents of the data item referenced by identifier-4 remains unchanged. However, as with other instances of multiple results of arithmetic statements, the user will have to do his own analysis to recognize which situation has actually occurred.

THE ENTER STATEMENT

Function

The ENTER statement provides a means of allowing the use of more than one language in the same program.

General Format

```
ENTER    language-name    [routine-name]
```

Syntax Rule

This statement is treated as if for documentation purposes only.

General Rule

Access to other languages can be achieved by means of CALL.

THE EXIT STATEMENT

Function

The EXIT statement provides a common end point for a series of procedures.

General Format

EXIT

Syntax Rules

1. The EXIT statement must appear in a sentence by itself.

2. The EXIT sentence must be the only sentence in the paragraph.

General Rule

An EXIT statement serves only to enable the user to assign a procedure-name to a given point in a program. Such an EXIT statement has no other effect on the compilation or execution of the program.

THE GO TO STATEMENT

Function

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

General Format

Format 1

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

Format 2

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

        DEPENDING ON identifier
```

Syntax Rules

1. Identifier is the name of a numeric elementary item described without any positions to the right of the assumed decimal point.

2. When a paragraph is referenced by an ALTER statement, that paragraph can consist only of a paragraph header followed by a Format 1 GO TO statement.

3. A Format 1 GO TO statement, without procedure-name-1, can only appear in a single statement paragraph.

4. If a GO TO statement represented by Format 1 appears in a consecutive sequence of imperative statements within a sentence, it appears as the last statement in that sequence.

General Rules

1. When a GO TO statement, represented by Format 1 is executed, control is transferred to procedure-name-1 or to another procedure-name if the GO TO statement has been modified by an ALTER statement.

2. If procedure-name-1 is not specified in Format 1, an ALTER statement, referring to this GO TO statement, must be executed prior to the execution of this GO TO statement.

3. When a GO TO statement represented by Format 2 is executed, control is transferred to procedure-name-1, procedure-name-2, etc., depending on the value of the identifier being 1, 2, ..., 99. If the value of the identifier is anything other than the positive or unsigned integers 1, 2, ..., 99, then no transfer occurs and control passes to the next statement in the normal sequence for execution.

THE IF STATEMENT

Function

The IF statement causes a condition to be evaluated (see CONDITIONAL EXPRESSIONS in this Section). The subsequent action of the object program depends on whether the value of the condition is true or false.

General Format

$$\underline{\text{IF}}\ \text{condition;}\quad \left\{\begin{array}{l}\text{statement-1}\\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}}\end{array}\right\}\quad \left\{\begin{array}{l}\text{; }\underline{\text{ELSE}}\ \text{statement-2}\\ \text{; }\underline{\text{ELSE}}\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}}\end{array}\right\}$$

Syntax Rules

1. Statement-1 and statement-2 represent either an imperative statement or a conditional statement, and either may be followed by a conditional statement.

2. The ELSE NEXT SENTENCE phrase may be omitted if it immediately precedes the terminal period of the sentence.

General Rules

1. When an IF statement is executed, the following transfers of control occur:

   a. If the condition is true, statement-1 is executed if specified. If statement-1 contains a procedure branching or conditional statement, control is explicitly transferred in accordance with the rules of that statement. If statement-1 does not contain a procedure branching or conditional statement, the ELSE phrase, if specified, is ignored and control passes to the next executable sentence.

   b. If the condition is true and the NEXT SENTENCE phrase is specified instead of statement-1, the ELSE phrase, if specified, is ignored and control passes to the next executable sentence.

   c. If the condition is false, statement-1 or its surrogate NEXT SENTENCE is ignored, and statement-2, if specified, is executed. If statement-2 contains a procedure branching or conditional statement, control is explicitly transferred in accordance with the rules of that statement. If statement-2 does not contain a procedure branching or conditional statement, control passes to the next executable sentence. If the ELSE statement-2 phrase is not specified, statement-1 is ignored and control passes to the next executable sentence.

d. If the condition is false, and the ELSE NEXT SENTENCE phrase is specified, statement-1 is ignored, if specified, and control passes to the next executable sentence.

2. Statement-1 and/or statement-2 may contain an IF statement. In this case the IF statement is said to be nested.

   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 been already paired with an ELSE.

Function

The INSPECT statement provides the ability to tally (Format 1), replace (Format 2), or tally and replace (Format 3) occurrences of single characters in a data item.

General Format

Format 1

```
 { INSPECT   identifier-1     TALLYING
 {
 {      ,   identifier-2 FOR  { , {{ ALL      }      { identifier-3 }
 {                            {   {{ LEADING  }      { literal-1    }
 {                            {   {  CHARACTERS
        { BEFORE }        INITIAL      { identifier-4 }  } } ... } ...
        { AFTER  }                     { literal-2    }  } }     }
```

Format 2

```
   INSPECT   identifier-1    REPLACING
 {     CHARACTERS BY { identifier-6 } [ { BEFORE }    INITIAL    { identifier-7 } ] }
 {                   { literal-4    } [ { AFTER  }               { literal-5    } ] }
 { { , { ALL     } } , { identifier-5 }   BY  { identifier-6 }                      }
 { {   { LEADING } }   { literal-3    }       { literal-4    }                      }
 { {   { FIRST   } }                                                                }
 { [{ BEFORE }    INITIAL    { identifier-7 }] } ...  }   ...                        }
 { [{ AFTER  }               { literal-5    }] }      }                              }
```

Format 3

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

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

<u>Syntax Rules</u>

All Formats

1. Identifier-1 must reference either a group item or any category of elementary item, described (either implicitly or explicitly) as usage is DISPLAY.

2. Identifier-3 ... identifier-n must reference either an elementary alphabetic, alphanumeric or numeric item described (either implicitly or explicitly) as usage is DISPLAY.

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

4. Literal-1, literal-2, literal-3, literal-4, and literal-5, and the data items referenced by identifier-3, identifier-4, identifier-5, identifier-6, an identifier-7 can be any number of characters in length.

Formats 1 and 3 Only

5. Identifier-2 must reference an elementary numeric data item.

6. If either literal-1 or literal-2 is a figurative constant, the figurative constant refers to an implicit one character data item.

Formats 2 and 3 Only

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

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

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

General Rules

All Formats

1. Inspection (which includes the comparison cycle, the establishment of boundaries for the BEFORE or AFTER phrase, and the mechanism for tallying and/or replacing) begins at the leftmost character position of the data item referenced by identifier-1, regardless of its class, and proceeds from left to right to the rightmost character position as described in General Rules 4 through 6.

2. For use in the INSPECT statement, the contents of the data item referenced by identifier-1, identifier-3, identifier-4, identifier-5, identifier-6 or identifier-7 will be treated as follows:

   a. If any of identifier-1, identifier-3, identifier-4, identifier-5, identifier-6 or identifier-7 are described as alphanumeric, the INSPECT statement treats the contents of each such identifier as a character-string.

   b. If any of identifier-1, identifier-3, identifier-4, identifier-5, identifier-6 or identifier-7 are described as alphanumeric edited, numeric edited or unsigned numeric, the data item is inspected as though it had been redefined as alphanumeric (see General Rule 2a) and the INSPECT statement had been written to reference the redefined data item.

   c. If any of the identifier-1, identifier-3, identifier-4, identifier-5, identifier-6 or identifier-7 are described as signed numeric, the data item is inspected as though it had been moved to an unsigned numeric data item of the same length and then the rules in General Rule 2b had been applied. (See the MOVE Statement later in this Section.)

3. In General Rules 4 through 11 all reference to literal-1, literal-2, literal-3, literal-4 and literal-5 apply equally to the contents of the data item referenced by identifier-3, identifier-4, identifier-5, identifier-6, and identifier-7, respectively.

4. During inspection of the contents of the data item referenced by identifier-1, each properly matched occurrence of literal-1 is tallied (Formats 1 and 3) and/or each properly matched occurrence of literal-3 is replaced by literal-4 (Formats 2 and 3). Data items to be referenced by the INSPECT verb should be placed such that they lie within the first 10,000 bytes of intermediate code.

5. The comparison operation to determine the occurrences of literal-1 to be tallied and/or occurrences of literal-3 to be replaced, occurs as follows:

   a. The operands of the TALLYING and REPLACING phrases are considered in the order they are specified in the INSPECT statement from left to right. The first literal-1, literal-3 is compared to an equal number of contiguous characters, starting with the leftmost character position in the data item referenced by identifier-1. Literal-1, literal-3 and that portion of the contents of the data item referenced by identifier-1 match if, and only if, they are equal, character for character.

   b. If no match occurs in the comparison of the first literal-1, literal-3, the comparison is repeated with each successive literal-1, literal-3, if any, until either a match is found or there is no next successive literal-1, literal-3. When there is no next successive literal-1, literal-3, the character position in the data item referenced by identifier-1 immediately to the right of the leftmost character position considered in the last comparison cycle is considered as the leftmost character position, and the comparison cycle begins again with the first literal-1, literal-3.

   c. Whenever a match occurs, tallying and/or replacing takes place as described in General Rules 8 through 10. The character position in the data item referenced by identifier-1 immediately to the right of the rightmost character position that participated in the match is now conidered to be the leftmost character position of the data item referenced by identifier-1, and the comparison cycle starts again with the first literal-1, literal-3.

   d. The comparison operation continues until the rightmost character position of the data item referenced by identifier-1 has participated in a match or has been considered as the leftmost character position. When this occurs, inspection is terminated.

   e. If the CHARACTERS phrase is specified, an implied one character operand participates in the cycle described in paragraphs 5a through 5d above, except that no comparison to the contents of the data item referenced by identifier-1 takes place. This implied character is considered always to match the leftmost character of the contents of the data item referenced by identifier-1 participating in the current comparison cycle.

6. The comparison operation defined in General Rule 5 is affected by the BEFORE and AFTER phrases as follows:

   a. If the BEFORE or AFTER phrase is not specified, literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates in the comparison operation as described in General Rule 5.

   b. If the BEFORE phrase is specified, the associated, literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates only in those comparison cycles which involve that portion of the contents of the data item referenced by identifier-1 from its leftmost character position up to, but not including, the first occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1. The position of this first occurrence is determined before the first cycle of the comparison operation described in General Rule 5 is begun. If, on any comparison cycle, literal-1, literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of the data item referenced by identifier-1. If there is no occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1, its associated literal-1, literal-3, or the implied operand of the CHARACTERS phrase participates in the comparison operation as though the BEFORE phrase had not been specified.

   c. If the AFTER phrase is specified, the associated literal-1, literal-3 or the implied operand of the CHARACTERS phrase may participate only in those comparison cycles which involve that portion of the contents of the data item referenced by identifier-1 from the character position immediately to the right of the rightmost character position of the first occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1 and the rightmost character position of the data item referenced by identifier-1. The position of this first occurrence is determined before the first cycle of the comparison operation described in General Rule 5 is begun. If, on any comparison cycle, literal-1, literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of the data item referenced by identifier-1. If there is no occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1, its associated literal-1, literal-3, or the implied operand of the CHARACTERS phrase is never eligible to participate in the comparison operation.

Format 1

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

8. The rules for tallying are as follows:

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

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

   c. If the CHARACTERS phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one for each character matched, in the sense of General Rule 5e, within the contents of the data item referenced by identifier-1.

Format 2

9. The required words ALL, LEADING, and FIRST are adjectives that apply to each succeeding BY phrase until the next adjective appears.

10. The rules for replacement are as follows:

    a. When the CHARACTERS phrase is specified, each character matched in the sense of General Rule 5e, in the contents of the data item referenced by identifier-1 is replaced by literal-4.

    b. When the adjective ALL is specified, each occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.

    c. When the adjective LEADING is specified, each contiguous occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which literal-3 was eligible to participate.

    d. When the adjective FIRST is specified, the leftmost occurrence of literal-3 matched within the contents of the data item referenced by identifier-1 is replaced by literal-4.

Format 3

11. A Format 3 INSPECT statement is interpreted and executed as though two successive INSPECT statements specifying the same identifier-1 had been written with one statement being a Format 1 statement with TALLYING phrases identical to those specified in the Format 3 statement, and the other statement being a Format 2 statement with REPLACING phrases identical to those specified in the Format 3 statement. The general rules given for matching and counting apply to the Format 1 statement and the general rules given for matching and replacing apply to the Format 2 statement.

EXAMPLES

Six examples of the use of the INSPECT statement follow:

INSPECT word TALLYING count FOR LEADING "L" BEFORE INITIAL "A", count-1 FOR LEADING "A" BEFORE INITIAL "L".

Where word = LARGE, count = 1, count-1 = 0.
Where word = ANALYST, count = 0, count-1 = 1.

INSPECT word TALLYING count FOR ALL "L", REPLACING LEADING "A" BY "E" AFTER INITIAL "L".

Where word = CALLAR, count = 2, word = CALLAR.
Where word = SALAMI, count = 1, word = SALEMI.
Where word = LATTER, count = 1, word = LETTER.

INSPECT word REPLACING ALL "A" BY "G" BEFORE INITIAL "X".

Where word = ARXAX, word = GRXAX.
Where word = HANDAX, word = HGNDGX.

INSPECT word TALLYING count FOR CHARACTERS AFTER INITIAL "J" REPLACING ALL "A" BY "B".

Where word = ADJECTIVE, count = 6, word = BDJECTIVE.
Where word = JACK, count = 3, word = JBCK.
Where word = JUJMAB, count = 5, word = JUJMBB.

INSPECT word REPLACING ALL "X" BY "Y", "B" BY "Z", "W" BY "Q" AFTER INITIAL "R".

Where word = RXXBQWY, word = RYYZQQY.
Where word = YZACDWBR, word = YZACDWBR.
Where word = RAWRXEB, word = RAQRYEZ.

INSPECT word REPLACING CHARACTERS BY "B" BEFORE INITIAL "A".

word before:	12XZABCD
word after:	BBBBBABCD

THE MOVE STATEMENT

Function

The MOVE statement transfers data, in accordance with the rules of editing, to one or more data areas.

General Format

Format 1

```
MOVE   { identifier-1 }    TO    identifier-2        [, identifier-3] ...
       { literal      }
```

Format 2

```
MOVE   { CORRESPONDING }  identifier-1    TO     identifier-2
       { CORR          }
```

Syntax Rules

1. Identifier-1 and literal represent the sending area; identifier-2, identifier-3, ..., represent the receiving area.

2. CORR is an abbreviation for CORRESPONDING.

3. When the CORRESPONDING phrase is used, both identifiers must be group items.

4. An index data item cannot appear as an operand of a MOVE statement. (See the USAGE Clause in this Section).

General Rules

1. If the CORREPONDING phrase is used, selected items within identifier-1 are moved to selected items within identifier-2, according to the rules given in the CORRESPONDING PHRASE in this Section. The results are the same as if the user had referred to each pair of corresponding identifiers in separate MOVE statements.

2. The data designated by the literal or identifier-1 is moved first to identifier-2, then to identifier-3, ... The rules referring to identifier-2 also apply to the other receiving areas. Any subscripting or indexing associated with identifier-2, ..., is evaluated immediately before the data is moved to the respective data item.

   Any subscripting or indexing associated with identifier-1 is evaluated only once, immediately before data is moved to the first of the receiving operands. The result of the statement:

   MOVE a (b) TO b, c (b)

is equivalent to:

   MOVE a (b) TO temp
   MOVE temp TO b
   MOVE temp TO c (b)

where 'temp' is an intermediate result item provided by the implementor.

3. Any MOVE in which the sending and receiving items are both elementary items is an elementary move. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric edited, alphanumeric edited. These categories are described in the PICTURE clause. Numeric literals belong to the category numeric, and nonnumeric literals belongs to the category alphanumeric. The figurative constant ZERO belongs to the category numeric. The figurative constant SPACE belongs to the category alphabetic. All other figurative constants belong to the category alphanumeric.

The following rules apply to an elementary move between these categories.

a. **The figurative constant SPACE, numeric edited, alphanumeric edited, or alphabetic data item must not be moved to a numeric or numeric edited data item.**

b. A numeric literal, the figurative constant ZERO, a numeric data item or a numeric edited data item must not be moved to an alphabetic data item.

c. A non-integer numeric literal or a non-integer numeric data item must not be moved to an alphanumeric or alphanumeric edited data item.

d. The result of moving a blank numeric-edited field to a numeric field is unpredictable and therefore not permitted.

e. All other elementary moves are legal and are performed according to the rules given in General Rule 4.

4. Any necessary conversion of data from one form of internal representation to another takes place during legal elementary moves, along with any editing specified for the receiving data item:

   a. When an alphanumeric edited or alphanumeric item is a receiving item, alignment and any necessary space filling takes place as defined under STANDARD ALIGNMENT RULES in Section 2. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled. If the sending item is described as being signed numeric, the operational sign will not be moved; if the operational sign occupies a separate character position (see the SIGN Clause in this Section), that character will not be moved and the size of the sending item will be considered to be one less than its actual size (in terms of standard data format characters).

   b. When a numeric or numeric edited item is the receiving item, alignment by decimal point and any necessary zero-filling takes place as defined under the STANDARD ALIGNMENT RULES in Section 2, except where zeroes are replaced because of editing requirements.

      When a signed numeric item is the receiving item, the sign of the sending item is placed in the receiving item. (See the SIGN Clause in this Section). Conversion of the representation of the sign takes place as necessary. If the sending item is unsigned, a positive sign is generated for the receiving item.

      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 a data item described as alphanumeric is the sending item, data is moved as if the sending item were described as an unsigned numeric integer.

   c. When a receiving field is described as alphabetic, justification and any necessary space-filling takes place as defined under the STANDARD ALIGNMENT RULES in Section 2. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled.

5. Any move that is not an elementary move is treated exactly as if it were an alphanumeric to alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In such a move, the receiving area will be filled without consideration for the individual elementary or group items contained within either the sending or receiving area, except as noted in General Rule 4 of the OCCURS clause.

6. Data in Table 3-8 summarizes the legality of the various types of MOVE statements. The general rule reference indicates the rule that prohibits the move or the behavior of a legal move.

Table 3-8. MOVE Statement Data Categories.

Category of Sending Data Item	Category of Receiving Data Item[1]		
	Alphabetic	Alphanumeric Edited Alphanumeric	Numeric Integer Numeric Non-Integer Numeric Edited
ALPHABETIC	Yes/3c	Yes/3a	No/2a
ALPHANUMERIC	Yes/3c	Yes/3a	Yes/3b
ALPHANUMERIC EDITED	Yes/3c	Yes/3a	No/2a
INTEGER NUMERIC NON-INTEGER	No/2b No/2b	Yes/3a No/2c	Yes/3b Yes/3b
NUMERIC EDITED	No/2b	Yes/3a	Yes/2a

1 - The relevant rules number is quoted in these columns

THE MULTIPLY STATEMENT

Function

The MULTIPLY statement causes numeric data items to be multiplied and sets the values of data items equal to the results.

General Format

Format 1

```
MULTIPLY  {identifier-1}  BY identifier-2     [ROUNDED]
          {literal-1   }

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

Format 2

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

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

Syntax Rules

1. Each identifier must refer to a numeric elementary item, except that in Format 2 each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item.

2. Each literal must be numeric literal.

3. The composite of operands, which is that hypothetical data item resulting from the superimposition of all receiving data items aligned on their decimal points must not contain more than 18 digits.

General Rules

1. See the ROUNDED PHRASE, the SIZE ERROR PHRASE, the ARITHMETIC Statements, OVERLAPPING OPERANDS and MULTIPLE RESULTS IN ARITHMETIC Statements in this Section.

2. When Format 1 is used, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2. The value of the multiplier (identifier-2) is replaced by this product; similarly for identifier-1 or literal-1 and identifier-3, etc.

3. When Format 2 is used, the value of identifier-1 or literal-1 is multiplied by identifier-2 or literal-2 and the result is stored in identifier-3, identifier-4, etc.

THE PERFORM STATEMENT

Function

The PERFORM statement is used to transfer control explicitly to one or more procedures and to return control implicitly whenever execution of the specified procedure is complete.

General Format

Format 1

```
PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ]
                         [ { THRU    }                   ]
```

Format 2

```
PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ] { identifier-1 }   TIMES
                         [ { THRU    }                   ] { integer-1    }
```

Format 3

```
PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ] UNTIL condition-1
                         [ { THRU    }                   ]
```

Format 4

```
PERFORM procedure-name-1 [ { THROUGH }  procedure-name-2 ]
                         [ { THRU    }                   ]

     VARYING  { identifier-2 }  FROM  { identifier-3 }
              { index-name-1 }        { index-name-2 }
                                      { literal-1    }

              BY  { identifier-4 }    UNTIL       condition-1
                  { literal-2    }

   [  AFTER   { identifier-5 }  FROM  { identifier-6 }
   [          { index-name-3 }        { index-name-4 }
   [                                  { literal-3    }
   [
   [          BY  { identifier-7 }    UNTIL       condition-2
                  { literal-4    }

  [   AFTER   { identifier-8 }  FROM  { identifier-9 }
  [           { index-name-5 }        { index-name-6 }       ]
                                      { literal-5    }       ]
                                                             ]
              BY  { identifier-10 }   UNTIL       condition-3 ]]
                  { literal-6     }
```

Keywords underlined in the original: PERFORM, THROUGH, THRU, TIMES, UNTIL, VARYING, FROM, BY, AFTER.

Syntax Rules

1. Each identifier represents a numeric elementary item described in the Data Division. In Format 2, identifier-1 must be described as a numeric integer.
2. Each literal represents a numeric literal.
3. The words THRU and THROUGH are equivalent.
4. If an index-name is specified in the VARYING or AFTER phrase, then:
   a. The identifier in the associated FROM and BY phrases must be an integer data item.
   b. The literal in the associated FROM phrase must be a positive integer.
   c. The literal in the associated BY phrase must be a non-zero integer.
5. If an index-name is specified in the FROM phrase, then:
   a. The identifier in the associated VARYING or AFTER phrase must be an integer data item.
   b. The identifier in the associated BY phrase must be an integer data item.
   c. The literal in the associated BY phrase must be an integer.
6. Literal in the BY phrase must not be zero.
7. Condition-1, condition-2, condition-3 may be any conditional expression as described under CONDITIONAL EXPRESSIONS in this Section.
8. Where procedure-name-1 and procedure-name-2 are both specified and either is the name of a procedure in the declarative section of the program then both must be procedure-names in the same declarative section.

General Rules

1. The data items referenced by identifier-4, identifier-7, and identifier-10 must not have a zero value.
2. If an index-name is specified in the VARYING or AFTER phrase, and an identifier is specified in the associated FROM phrase, then the data item referenced by the identifier must have a positive value.

3. When the PERFORM statement is executed, control is transferred to the first statement of the procedure named procedure-name-1 (except as indicated in General Rule 6). This transfer of control occurs only once for each execution of a PERFORM statement. For those cases where a transfer of control to the named procedure does take place, an implicit transfer of control to the next executable 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 is after the last statement of procedure-name-1.

   b. If procedure-name-1 is a section-name and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.

   c. If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.

   d. If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.

4. 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 logical 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.

5. If control passes to these procedures other than via a PERFORM statement the procedures are executed right through to the next executable statement in the main program as if they were just part of the main program.

6. The PERFORM statements operate as follows with Rule 5 above applying to all formats:

   a. Format 1 is the basic PERFORM statement. A procedure referenced by this type of PERFORM statement is executed once and then control passes to the next executable statement following the PERFORM statement.

   b. Format 2 is the PERFORM ... TIMES. The procedures are performed the number of times specified by integer-1 or by the initial value of the data item referenced by identifier-1 for that execution. If, at the time of execution of a PERFORM statement, the value of the data item referenced by identifier-1 is equal to zero or is negative, control passes to the next executable statement following the PERFORM statement. Following the execution of the procedures the specified number of times, control is transferred to the next executable statement following the PERFORM statement.

During execution of the PERFORM statement, references to identifier-1 cannot alter the number of times the procedures are to be executed from that which was indicated by the initial value of identifier-1.

c. Format 3 is the PERFORM ... UNTIL. The specified procedures are performed until the condition specified by the UNTIL phrase is true. When the condition is true, control is transferred to the next executable statement after the PERFORM statement. If the condition is true when the PERFORM statement is entered, no transfer to procedure-name-1 takes place, and control is passed to the next executable statement following the PERFORM statement.

d. Format 4 is the PERFORM ... VARYING. This variation of the PERFORM statement is used to augment the values referenced by one or more identifiers or index-names in an orderly fashion during the execution of a PERFORM statement. In the following discussion, every reference to identifier is the object of the VARYING, AFTER and FROM (current value) phrases also refers to index-names. When index-name appears in a VARYING and/or AFTER phrase, it is initialized and subsequently augmented (as described below) according to the rules of the SET statement. When index-name appears in the FROM phrase and identifier appears in an associated VARYING or AFTER phrase, identifier is initialized according to the rules of the SET statement; subsequent augmentation is as described below.

   In Format 4, when one identifier is varied, identifier-2 is set to the value of literal-1 or the current value of identifier-3 at the point of initial execution of the PERFORM statement; then, if the condition of the UNTIL phrase is false, the sequence of procedures, procedure-name-1 through procedure-name-2, is executed once. The value of identifier-2 is augmented by the specified increment or decrement value (the value of identifier-4 or literal-2) and condition-1 is evaluated again. The cycle continues until this condition is true; at which point, control is transferred to the next executable statement following the PERFORM statement. If condition-1 is true at the beginning of execution of the PERFORM statement, control is transferred to the next executable statement following the PERFORM statement.

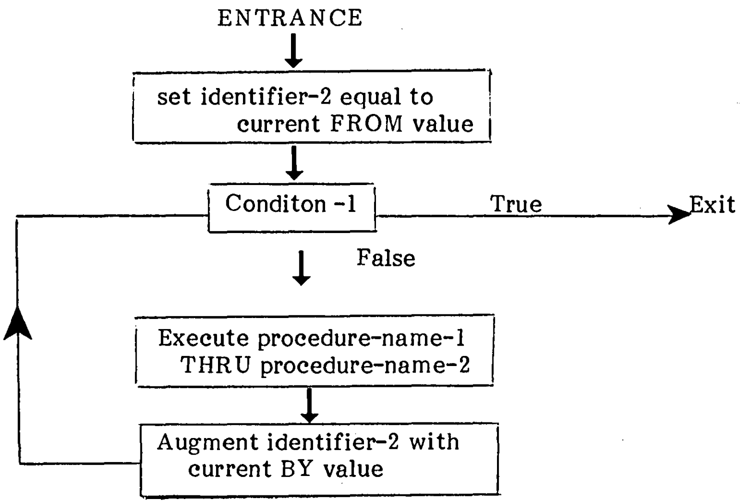

ENTRANCE
set identifier-2 equal to current FROM value
Conditon -1
True
Exit
False
Execute procedure-name-1 THRU procedure-name-2
Augment identifier-2 with current BY value

Figure 3-1. Flowchart for VARYING Phrase of a PERFORM Statement Having One Condition.

In Format 4, when two identifiers are varied, identifier-2 and identifier-5 are set to the current value of identifier-3 and identifier-6, respectively. After the identifiers have been set, condition-1 is evaluated; if true, control is transferred to the next executable statement; if false, condition-2 is evaluated. If condition-2 is false, procedure-name-1 through procedure-name-2 is executed once, then identifier-5 is augmented by identifier-7 or literal-4 and condition-2 is evaluated again. This cycle of evaluation and augmentation continues until this condition is true. When condition-2 is true, identifier-5 is set to the value of literal-3 or the current value of identifier-6, identifier-2 is augmented by identifier-4 and condition-1 is re-evaluated. The PERFORM statement is completed if condition-1 is true; if not, the cycles continue until condition-1 is true.

During the execution of the procedures associated with the PERFORM statement, any change to the VARYING variable (identifier-2 and index-name-1), the BY variable (identifier-4), the AFTER variable (identifier-5 and index-name-3), or the FROM variable (identifier-3 and index-name-2) will be taken into consideration and will affect the operation of the PERFORM statement.

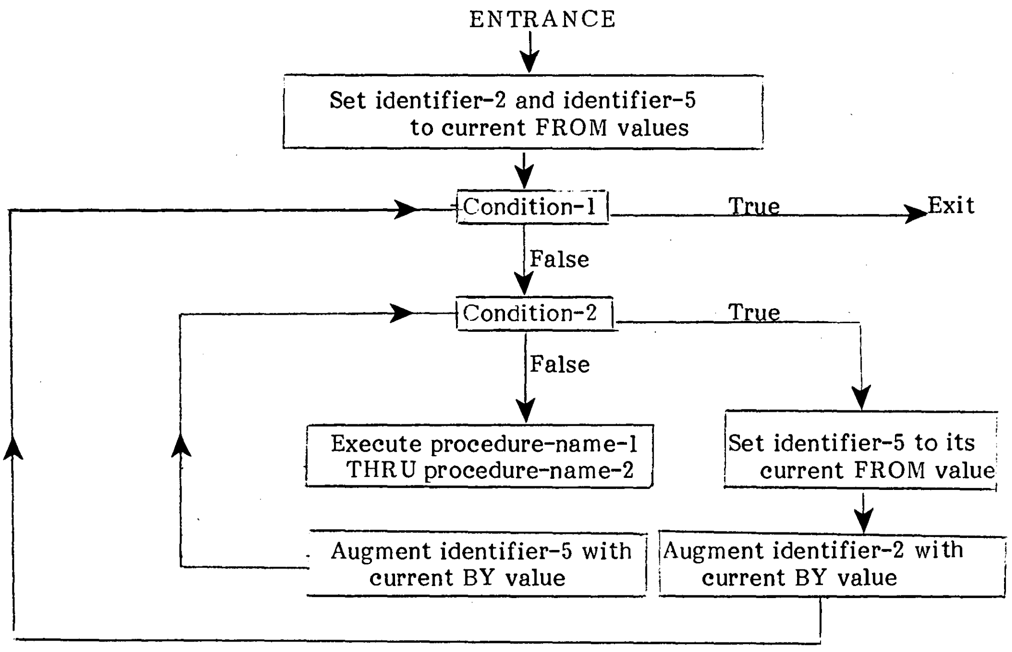

ENTRANCE
Set identifier-2 and identifier-5 to current FROM values
Condition-1
True
Exit
False
Condition-2
True
False
Execute procedure-name-1 THRU procedure-name-2
Set identifier-5 to its current FROM value
Augment identifier-5 with current BY value
Augment identifier-2 with current BY value

Figure 3-2. Flowchart for VARYING Phrase of PERFORM Statement with Two Conditions.

At the termination of the PERFORM statement, identifier-5 contains the current value of identifier-6. Identifier-2 has a value that exceeds the last used setting by an increment or decrement value, unless condition-1 was true when the PERFORM statement was entered, in which case identifier-2 contains the current value of identifier-3.

When two identifiers are varied, identifier-5 goes through a complete cycle (FROM, BY, UNTIL) each time identifier-2 is varied.

For three identifiers the mechanism is the same as for two identifiers except that identifier-8 goes through a complete cycle each time that identifier-5 is augmented by identifier-7 or literal-4, which in turn goes through a complete cycle each time identifier-2 is varied.

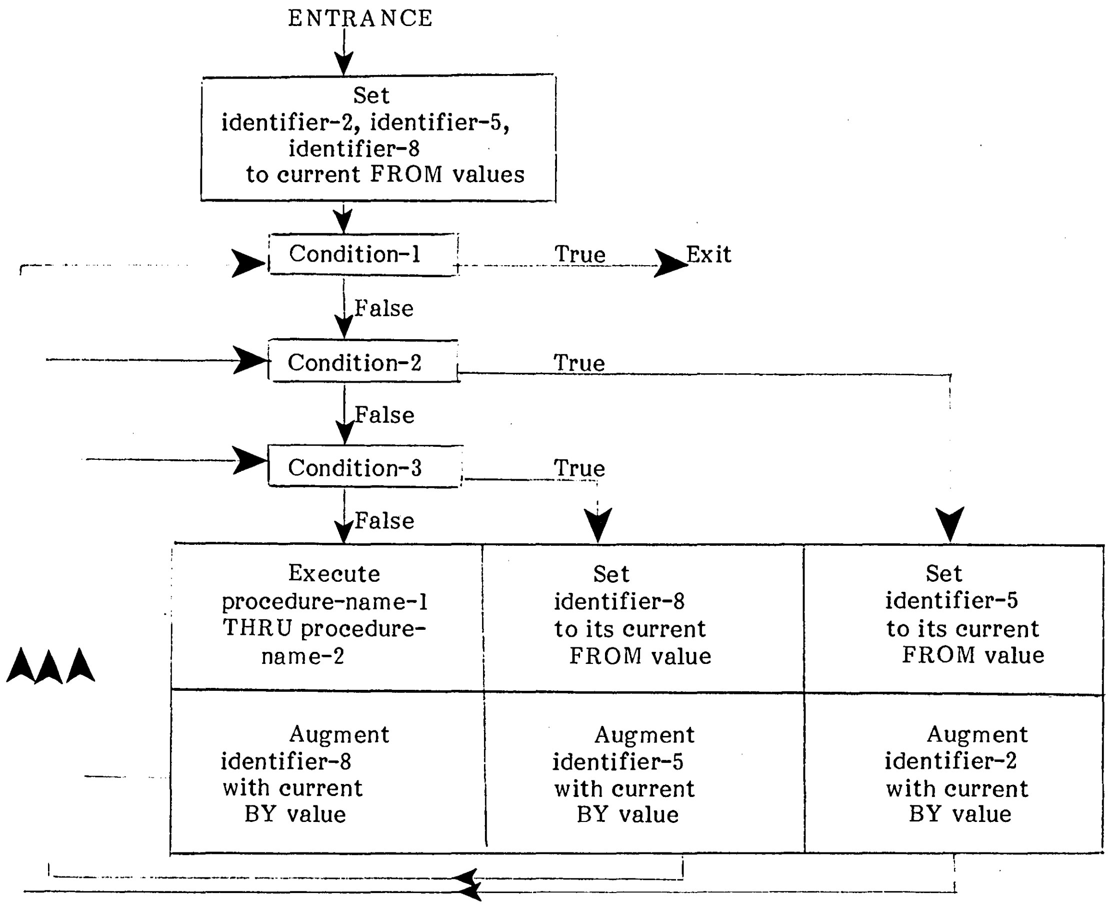

ENTRANCE
Set identifier-2, identifier-5, identifier-8 to current FROM values
Condition-1
True
Exit
False
Condition-2
True
False
Condition-3
True
False
Execute procedure-name-1 THRU procedure-name-2
Set identifier-8 to its current FROM value
Set identifier-5 to its current FROM value
Augment identifier-8 with current BY value
Augment identifier-5 with current BY value
Augment identifier-2 with current BY value

Figure 3-3. Flowchart for VARYING Phrase of PERFORM Statement with Three Conditions.

After the completion of a Format 4 PERFORM statement, identifier-5 and identifier-8 contain the current value of identifier-6 and identifier-9 respectively. Identifier-2 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-2 contains the current value of identifier-3.

7. If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements may not have a common exit. See Figure 3-4.

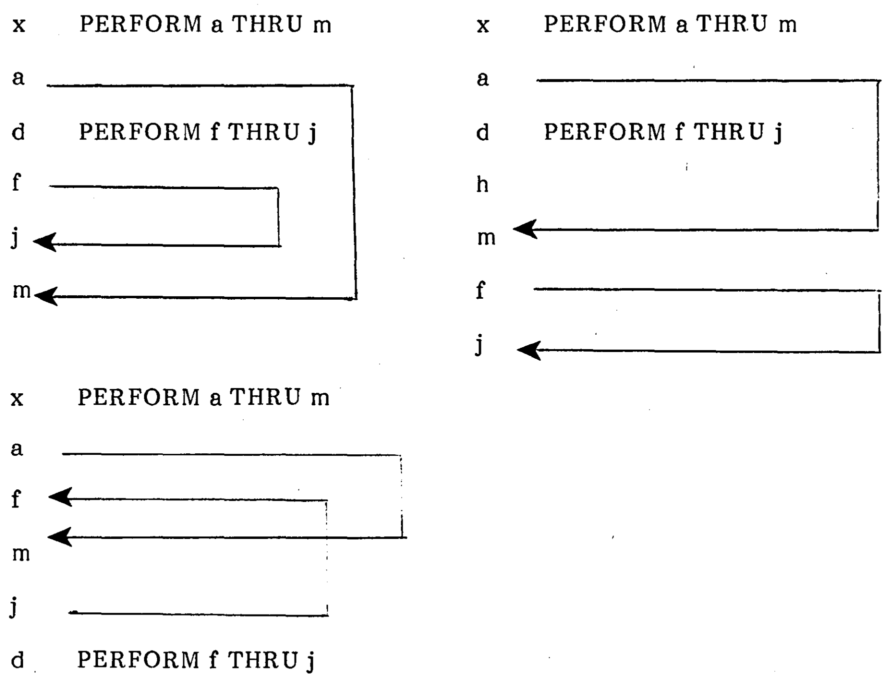


Fig. 3-4. PERFORM Statement in Sequence.

8. A PERFORM statement that appears in a section that is not an independent segment can have within its range, in addition to any declarative sections whose execution is cause within that range, only one of the following:

   a. Sections and/or paragraphs wholly contained in one or more non-independent segments.

   b. Sections and/or paragraphs wholly contained in a single independent segment.

9. A PERFORM statement that appears in an independent segment can have within its range, in addition to any declarative sections whose execution is caused within that range, only one of the following:

   a. Sections and/or paragraphs wholly contained in one or more non-independent segments.

   b. Sections and/or paragraphs wholly contained in the same independent segment as that PERFORM statement.

THE STRING STATEMENT

Function

The STRING statement provides juxtaposition of the partial or complete contents of two or more data items into a single data item.

General Format

```
                                                                       ( identifier-3 )
STRING    {identifier-1}   [, identifier-2]   ... DELIMITED BY         { literal-3    }
          {literal-1   }   [, literal-2   ]                            ( SIZE         )

      [[identifier-4]   [, identifier-5]...
      [[literal-4   ]   [, literal-5   ]

                                    ( identifier-6 )   ]
                  DELIMITED BY      { literal-6    }   ]   ...
                                    ( SIZE         )   ]

        INTO     identifier-7     [WITH POINTER identifier-8]

        [; ON OVERFLOW imperative-statement]
```

Syntax Rules

1. Each literal may be any figurative constant without the optional word ALL.

2. All literals must be described as nonnumeric literals, and all identifiers, except identifier-8, must be described implicitly or explicitly as usage is DISPLAY.

3. Identifier-8 must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size plus 1 of the area referenced by identifier-7. The symbol 'P' may not be used in the PICTURE character-string of identifier-8.

4. Where identifier-1, identifier-2, ..., or identifier-3 is an elementary numeric data item, it must be described as an integer without the symbol 'P' in its PICTURE character-string.

General Rules

1. All references to identifier-1, identifier-2, identifier-3, literal-1, literal-2, literal-3 apply equally to identifier-4, identifier-5, identifier-6, literal-4, literal-5 and literal-6, respectively, and all recursions thereof.

2. Identifier-1, literal-1, identifier-2, literal-2, represent the sending items. Identifier-7 represents the receiving item.

3. Literal-3, identifier-3, indicate the character(s) delimiting the move. If the SIZE phrase is used, the complete data item defined by identifier-1, literal-1, identifier-2, literal-2, is moved. When a figurative constant is used as the delimiter, it stands for a single character nonnumeric literal.

4. When a figurative constant is specified as literal-1, literal-2, literal-3, it refers to an implicit one-character data item where usage is DISPLAY.

5. When the STRING statement is executed, the transfer of data is governed by the following rules:

   a. Those characters from literal-1, literal-2, or from the contents of the data item referenced by identifier-1, identifier-2, are transferred to the contents of identifier-7 in accordance with the rules for alphanumeric to alphanumeric moves, except that no space-filling will be provided. (See the MOVE Statement.)

   b. If the DELIMITED phrase is specified without the SIZE phrase, the contents of the data item referenced by identifier-1, identifier-2, or the value of literal-1, literal-2, are transferred to the receiving data item in the sequence specified in the STRING statement beginning with the leftmost character and continuing from left to right until the end of the data item is reached, or until the character(s) specified by literal-3, or by the contents of identifier-3 are encountered. The character(s) specified by literal-3, or by the data item referenced by identifier-3 are not transferred.

   c. If the DELIMITED phrase is specified with the SIZE phrase, the entire contents of literal-1, literal-2, or the contents of the data item referenced by identifier-1, identifier-2, are transferred, in the sequence specified in the STRING statement, to the data item referenced by identifier-7 until all data has been transferred or the end of the data item referenced by identifier-7 has been reached.

6. If the POINTER phrase is specified, identifier-8 is explicitly available to the programmer, who is then responsible for setting its initial value. The initial value must not be less than one.

7. If the POINTER phrase is not specified, the following general rules apply as if the user had specified identifier-8 with an initial value of 1.

8. When characters are transferred to the data item referenced by identifier-7, the moves behave as though the characters were moved one at a time from the source into the character position of the data item referenced by identifier-7 designated by the value associated with identifier-8, and then identifier-8 was increased by one prior to the move of the next character. The value associated with identifier-8 is changed during execution of the STRING statement only by the behavior specified above.

9. At the end of execution of the STRING statement, only the portion of the data item referenced by identifier-7 that was referenced during the execution of the STRING statement is changed. All other portions of the data item referenced by identifier-7 will contain data that was present before this execution of the STRING statement.

10. If at any point at or after initialization of the STRING statement, but before execution of the STRING statement is completed, the value associated with identifier-8 is either less than one or exceeds the number of character positions in the data item referenced by identifier-7, no (further) data is transferred to the data item referenced by identifier-7, and the imperative statement in the ON OVERFLOW phrase is executed, if specified.

11. If the ON OVERFLOW phrase is not specified when the conditions described in General Rule 10 above are encountered, control is transferred to the next executable statement.

THE STOP STATEMENT

Function

The STOP statement causes a permanent or temporary suspension of the execution of the object program.

General Format

$$\underline{\text{STOP}} \quad \left\{ \begin{matrix} \underline{\text{RUN}} \\ \text{literal} \end{matrix} \right\}$$

Syntax Rules

1. The literal may be numeric or nonnumeric or may be any figurative constant, except ALL.

2. If the literal is numeric, then it must be an unsigned integer.

3. If a STOP RUN statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

General Rules

1. If the RUN phrase is used, then the operating system ending procedure is instituted.

2. If STOP literal is specified, the literal is communicated to the operator. Continuation of the object program begins with the execution of the next executable statement, in sequence, after the operator presses RETURN.

THE SUBTRACT STATEMENT

Function

The SUBTRACT statement is used to subtract one, or the sum of two or more, numeric data items from one or more items, and set the values of one or more items equal to the results.

General Format

Format 1

```
SUBTRACT  {identifier-1}  [ ,{identifier-2} ] ... FROM
          {literal-1   }  [  {literal-2   } ]

     identifier-m     [ROUNDED] [ , identifier-n        [ROUNDED] ]

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

Format 2

```
SUBTRACT  {identifier-1}  [ ,{identifier-2} ] ... FROM  {identifier-m}
          {literal-1   }  [  {literal-2   } ]           {literal-m   }

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

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

Format 3

```
SUBTRACT  {CORRESPONDING}  identifier-1 FROM identifier-2 [ROUNDED]
          {CORR         }

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

Syntax Rules

1. Each identifier must refer to a numeric elementary item except that in Format 2, each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item, and in Format 3, each identifier must refer to a group item.

2. Each literal must be a numeric literal.

3. The composite of operands must not contain more than 18 digits. (See the ARITHMETIC Statements in this Section.)

   a. In Format 1 the composite of operands is determined by using all of the operands in a given statement.

   b. In Format 2 the composite of operands is determined by using all of the operands in a given statement excluding the data items that follow the word GIVING.

   c. In Format 3 the composite operands is determined separately for each pair of corresponding data items.

4. CORR is an abbreviation for CORRESPONDING.

<u>General Rules</u>

1. See the ROUNDED PHRASE, the SIZE ERROR PHRASE, the ARITHMETIC Statement, OVERLAPPING OPERANDS and MULTIPLE RESULTS IN ARITHMETIC Statements in this Section.

2. In Format 1, all literals or identifiers preceding the word FROM are added together and this total is subtracted from the current value of identifier-m storing the result immediately into identifier-m, and repeating this process respectively for each operand following the word FROM.

3. In Format 2, all literals or identifiers preceding the word FROM are added together, the sum is subtracted from literal-m or identifier-m and the result of the subtraction is stored as the new value of identifier-n, identifier-o, etc.

4. If Format 3 is used, data items in identifier-1 are subtracted from and stored into corresponding data items in identifier-2.

5. The compiler ensures enough places are carried so as not to lose significant digits during execution.

THE UNSTRING STATEMENT

Function

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

General Format

```
UNSTRING     identifier-1

[ DELIMITED BY [ALL] {identifier-2} [ , OR [ALL] {identifier-3} ] ... ]
                     {literal-1   }              {literal-2   }

    INTO  identifier-4     [, DELIMITER IN identifier-5]

    [, COUNT IN identifier-6]

[ , identifier-7  [, DELIMITER IN identifier-8]

   [, COUNT IN identifier-9] ] ...

  [WITH POINTER identifier-10]      [TALLYING IN identifier-11]

  [; ON OVERFLOW imperative-statement]
```

Syntax Rules

1. Each literal must be nonnumeric literal. In addition, each literal may be any figurative constant without the optional word ALL.

2. Identifier-1, identifier-2, identifier-3, identifier-5, and identifier-8 must be described, implicitly or explicitly, as an alphanumeric data item.

3. Identifier-4 and identifier-7 may be described as either alphabetic (except that the symbol 'B' may not be used in the PICTURE character-string), alphanumeric, or numeric (except that the symbol 'P' may not be used in the PICTURE character-string), and must be described as usage is DISPLAY.

4. Identifier-6, identifier-9, identifier-10, and identifier-11 must be described as elementary numeric integer data items (except the symbol 'P' may not be used in the PICTURE character-string).

5. No identifier may name a level 88 entry.

6. The DELIMITER IN phrase and the COUNT IN phrase may be specified only if the DELIMITED BY phrase is specified.

General Rules

1. All references to identifier-2, literal-1, identifier-4, identifier-5 and identifier-6, apply equally to identifier-3, literal-2, identifier-7, identifier-8 and identifier-9, respectively, and all recursions thereof.

2. Identifier-1 represents the sending area.

3. Identifier-4 represents the data receiving area. Identifier-5 represents the receiving area for delimiters.

4. Literal-1 or the data item referenced by identifier-2 specifies a delimiter.

5. Identifier-6 represents the count of the number of characters within the data item referenced by identifier-1 isolated by the delimiters for the move to identifier-4. This value does not include a count of the delimiter character(s).

6. The data item referenced by identifier-10 contains a value that indicates a relative character position within the area defined by identifier-1.

7. The data item referenced by identifier-11 is a counter that records the number of data items acted upon during the execution of an UNSTRING statement.

8. When a figurative constant is used as the delimiter, it stands for single character nonnumeric literal.

   When the ALL phrase is specified, one occurrence or two or more contiguous occurrences of literal-1 (figurative constant or not) or the contents of the data item referenced by identifier-2 are treated as if it were only one occurrence, and this occurrence is moved to the receiving data item according to the rules in General Rule 13d.

9. When any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled according to the description of the receiving area.

10. Literal-1 or the contents of the data item referenced by identifier-2 can contain any character in the computer's character set.

11. Each literal-1 or the data item referenced by identifier-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions of the sending item, and in the order given to be recognized as a delimiter.

12. When two or more delimiters are specified in the DELIMITED BY phrase, an 'OR' condition exists between them. Each delimiter is compared to the sending field. If a match occurs, the character(s) in the sending field is considered to be a single delimiter. No character(s) in the sending field is considered to be a single delimiter. No character(s) in the sending field can be considered a part of more than one delimiter.

    Each delimiter is applied to the sending field in the sequence specified in the UNSTRING statement.

13. When the UNSTRING statement is initiated, the current receiving area is the data item referenced by identifier-4. Data is transferred from the data item referenced by identifier-1 to the data item referenced by identifier-4 according to the following rules:

    a. If the POINTER phrase is specified, the string of characters referenced by identifier-1 is examined beginning with the relative character position indicated by the contents of the data item referenced by identifier-10. If the POINTER phrase is not specified, the string of characters is examined beginning with the leftmost character position.

    b. If the DELIMITED BY phrase is specified, the examination proceeds left to right until either a delimiter specified by the value of literal-1 or the data item referenced by identifier-2 is encountered. (See General Rule 11.) If the DELIMITED BY phrase is not specified, the number of characters examined is equal to the size of the current receiving area. However, if the sign of the receiving item is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area.

       If the end of the data item referenced by identifier-1 is encountered before the delimiting condition is met, the examination terminates with the last character examined.

    c. The characters thus examined (excluding the delimiting character(s), if any) are treated as an elementary alphanumeric data item, and are moved into the current receiving area according to the rules for the MOVE statement. (See the MOVE Statement.)

    d. If the DELIMITER IN Phrase is specified, the delimiting character(s) are treated as an elementary alphanumeric data item and are moved into the data item referenced by identifier-5 according to the rules for the MOVE statement. (See the MOVE Statement.) If the delimiting condition is the end of the data item referenced by identifier-1, then the data item referenced by identifier-5 is space-filled.

    e. If the COUNT IN phrase is specified, a value equal to the number of characters thus examined (excluding the delimiter character(s) if any) is moved into the area referenced by identifier-1 according to the rules for an elementary move.

f. If the DELIMITED BY phrase is specified, the string of characters is further examined beginning with the first character to the right of the delimiter. If the DELIMITED BY phrase is not specified, the string of characters is further examined beginning with the character to the right of the last character transferred.

g. After data is transferred to the data item referenced by identifier-4, the current receiving area is the data item referenced by identifier-7. The behavior described in paragraph 13b through 13f is repeated until either all the characters are exhausted in the data item referenced by identifier-1, or until there are no more receiving areas.

14. The initialization of the contents of the data items associated with the POINTER phrase or the TALLYING phrase is the responsibility of the user.

15. The contents of the data item referenced by identifier-10 will be incremented by one for each character examined in the data item referenced by identifier-1. When the execution of an UNSTRING statement with a POINTER phrase is complete, the contents of the data item referenced by identifier-10 will contain a value equal to the initial value plus the number of characters examined in the data item referenced by identifier-1.

16. When the execution of an UNSTRING statement with a TALLYING phrase is completed, the contents of the data item referenced by identifier-11 contains a value equal to its initial value plus the number of data receiving items acted upon.

17. Either of the following situations causes an overflow condition:

a. An UNSTRING is initiated, and the value in the data item referenced by identifier-10 is less than 1 or greater than the size of the data item referenced by identifier-1.

b. If, during execution of an UNSTRING statement, all data receiving areas have been acted upon, and the data item referenced by identifier-1 contains characters that have not been examined.

18. When an overflow condition exists, the UNSTRING operation is terminated. If an ON OVERFLOW phrase has been specified, the imperative statement included in the ON OVERFLOW phrase is executed. If the ON OVERFLOW phrase is not specified, control is transferred to the next executable statement.

19. The evaluation of subscripting and indexing for the identifiers is as follows:

    a. Any subscripting or indexing associated with identifier-1, identifier-10, identifier-11 is evaluated only once, immediately before any data is transferred as the result of the execution of the UNSTRING statement.

    b. Any subscripting or indexing associated with identifier-2, identifier-3, identifier-4, identifier-5, identifier-6 is evaluated immediately before the transfer of data into the respective data item.

SECTION 4

TABLE HANDLING

INTRODUCTION TO THE TABLE HANDLING MODULE

The Table Handling module provides a capability for defining tables of contiguous data items and accessing an item relative to its position in the table. Language facilities are provided for specifying how many times an item is to be repeated. Each item may be identified through use of a subscript or an index (see Section 2).

Table Handling provides a capability for accessing items in fixed length tables of multiple dimensions.

DATA DIVISION IN THE TABLE HANDLING MODULE

THE OCCURS CLAUSE

Function

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

General Format

Format 1

```
    OCCURS integer-2 TIMES
      ┌ ┌ASCENDING ┐                                   ┐
      │ └DESCENDING┘       KEY IS data-name-2  [, data-name-3   ...  │ ...
      └                                                   ┘
       ┌                                                 ┐
       │ [INDEXED BY index-name-1  [, index-name-2]   ...  │ ...
       └                                                 ┘
```

Format 2

```
    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]   ...  │
      └                                                ┘
```

Syntax Rules

1. Where both integer-1 and integer-2 are used, the value of integer-1 must be less than the value of integer-2.

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

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

4. 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.

5. Data-name-3, etc., must be the name of an entry subordinate to the group item which is the subject of this entry.

6. An INDEXED BY phrase is required if the subject of this entry, or an entry subordinate to this entry, is to be referred to by indexing. The index-name identified by this clause is not defined elsewhere since its allocation and format are dependent on the hardware, and not being data, cannot be associated with any data hierarchy.

7. A data description entry that contains Format 2 of the OCCURS clause may only be followed, within that record description, by data description entries which are subordinate to it.

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

   Describes an item whose size is variable. The size of an item is variable if the data description of any subordinate item contains Format 2 of the OCCURS clause.

9. In Format 2, the data item defined by data-name-1 must not occupy a character position within the range of the first character position defined by the data description entry containing the OCCURS clause and the last character position defined by the record description entry containing that OCCURS clause.

10. If data-name-2 is not the subject of this entry, then:

    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 this entry.

    b. Items identified by the data-name in the KEY IS phrase must not contain an OCCURS clause.

    c. There must not be any entry that contains an OCCURS clause between the items identified by the data-names in the KEY IS phrase and the subject of this entry.

11. Index-name-1, index-name-2, . . . must be unique words within the program.

General Rules

1. The OCCURS clause is used in defining tables and other homogenous sets of repeated data items. 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 or USE FOR DEBUGGING. Further, if the subject of this entry 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 as the object of a REDEFINES clause. (See under headings Subscripting, Indexing and Identifier in Section 2.)

2. Except for the OCCURS clause itself, all data description clauses associated with an item whose description includes an OCCURS clause apply to each occurrence of the item described. (See restriction in General Rule 2 under The Data Description-Complete Entry Skeleton in Section 3.)

3. The number of occurrences of the subject entry is defined as follows:

   a. In Format 1, the value of integer-2 representing the exact number of occurrences.

   b. In Format 2, the current value of the data item referenced by data-name-1 represents the number of occurrences.

      This format specifies that the subject of this entry has a variable number of occurrences. The value of integer-2 represents the maximum number of occurrences and the value of integer-1 represents the minimum number of occurrences. This does not imply that the length of the subject of the entry is variable, but that the number of occurrences is variable.

      The value of the data item referenced by data-name-1 must fall within the range of integer-1 through integer-2. Reducing the value of the data item referenced by data-name-1 makes the contents of data items, whose occurrence numbers now exceed the value of the data item referenced by data-name-1, unpredictable.

4. When a group item, having subordinate to it an entry that specifies Format 2 of the OCCURS clause, is referenced, only that part of the table area that is specified by the value of data-name-1 will be used in the operation.

5. The KEY IS phrase 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, etc. The ascending or descending order is determined according to the rules for comparison of operands (see Comparison of Numeric Operands, Comparison of Nonnumeric Operands in Section 3.) The data-names are listed in their descending order of significance.

THE USAGE CLAUSE

Function

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

General Format

[USAGE IS]   INDEX

Syntax Rules

1. An index data item can be referenced explicitly only in a SEARCH or SET statement, a relation condition, the USING phrase of a Procedure Division header, or the USING phrase of a CALL statement.

2. The SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items described with the USAGE IS INDEX clause.

General Rules

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

2. An elementary item described with the USAGE IS INDEX clause is called an index data item and contains a value which must correspond to an occurrence number of a table element. The elementary item cannot be a conditional variable. The compiler will allocate a 2 byte binary field. If a group item is described with the USAGE IS INDEX clause, the elementary items in the group are all index data items. The group itself is not an index data item and cannot be used in the SEARCH or SET statement or in a relation condition.

3. An index data item can be part of a group which is referred to in a MOVE or input-output statement, in which case no conversion will take place.

RELATION CONDITION

Comparisons Involving Index-Names And/Or Index Data Items

Relation tests may be made between the following data items:

* Two index-names. The result is the same as if the corresponding occurrence numbers were compared.
* An index-name and a data item (other than an index data item) or literal. The occurrence number that corresponds to the value of the index-name compared to the data item or literal.
* An index data item and an index-name or another index data item. The actual values are compared without conversion.
* The result of the comparison of an index data item with any data item or literal not specified above is undefined.

OVERLAPPING OPERANDS

When a sending and a receiving item in a SET statement share a part of their storage areas, the result of the execution of such a statement is undefined.

THE SEARCH STATEMENT

Function

The SEARCH statement is used to search a table for a table element that satisfies the specified condition and to adjust the associated index name to indicate that table element.

General Format

Format 1

```
SEARCH identifier-1 [ VARYING { identifier-2 } ]
                              { identifier-1 }

     [; AT END imperative-statement-1]

     ; WHEN condition-1 { imperative-statement-2 }
                         { NEXT SENTENCE          }

   [ ; WHEN condition-2 { imperative-statement-3 } ]  ...
                        { NEXT SENTENCE          }
```

Format 2

```
SEARCH ALL identifier-1 [; AT END imperative-statement-1]

                                                   (identifier-3            ) )
                                   (IS EQUAL TO)   (literal-1               ) )
    ; WHEN        ( data-name-1    (IS =       )   (arithmetic-expression-1 ) )
                  (
                  ( condition-name-1

  [                                                (identifier-4            ) )  ]
  [                                (IS EQUAL TO)   (literal-2               ) )  ]
  [ AND           ( data-name-2    (IS =       )   (arithmetic-expression-2 ) )  ]
  [               (                                                              ]
  [               ( condition-name-2                                             ]
  (imperative-statement-2)
  (NEXT SENTENCE         )
```

NOTE: The required relational character '=' is not underlined to avoid confusion with other symbols.

Syntax Rules

1. In both Formats 1 and 2, identifier-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause. The description of identifier-1 in Format 2 must also contain the KEY IS phrase in its OCCURS clause.

2. Identifier-2, when specified, must be described as USAGE IS INDEX or as a numeric elementary item without any positions to the right of the assumed decimal point.

3. In Format 1, condition-1, condition-2, etc., may be any condition as described in CONDITION EXPRESSIONS in Section 3.

4. In Format 2, all referenced condition-names must be defined as having only a single value. The data-name associated with a condition-name must appear in the KEY clause of identifier-1. Each data-name-1, data-name-2 may be qualified. Each data-name-1, data-name-2 must be indexed by the first index-name associated with identifier-1 along with other indices or literals as required, and must be referenced in the KEY clause of identifier-1. Identifier-3, identifier-4, or identifiers specified in arithmetic-expression-1, arithmetic-expression-2 must not be referenced in the KEY clause of identifier-1 or be indexed by the first index-name associated with identifier-1.

   In Format 2, when a data-name in the KEY clause of identifier-1 is referenced, or when a condition-name associated with a data-name in the KEY clause of identifier-1 is referenced then their associated index must also be referenced.

General Rules

1. If Format 1 of the SEARCH is used, a serial type of search operation takes place, starting with the current index setting.

   a. 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 greater than the highest permissible occurrence number for identifier-1, the SEARCH is terminated immediately. The number of occurrences of identifier-1, the last of which is the highest permissible, is discussed in the OCCURS clause. (See The OCCURS Clause in Section 4.) Then, if the AT END phrase is specified, imperative-statement-1 is executed; if the AT END phrase is not specified, control passes to the next executable sentence.

   b. 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 number of occurrences of identifier-1, the last of which is the highest permissible is discussed in the OCCURS clause) the SEARCH statement operates by evaluating the conditions in the order that they are written, making use of the index settings, wherever specified, to determine the occurrence of those items to be tested. If none of the conditions are satisfied, the index-name for identifier-1 is incremented to obtain reference to the next occurrence. The process is then repeated using the new index-name settings unless the new value of the index-name settings for identifier-1 corresponds to a table element outside the permissible range of occurrence values, in which case the search terminates as indicated in 1a above. 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.

2. In a Format 2 SEARCH, the results of the SEARCH ALL operation are predictable only when:

   a. The data in the table is ordered in the same manner as described in the ASCENDING/DESCENDING KEY clause associated with the description of identifier-1, and

   b. The contents of the key(s) referenced in the WHEN clause are sufficient to identify a unique table element.

3. If Format 2 of the SEARCH is used, a nonserial type of search operation may take place; the initial setting of the index-name for identifier-1 is ignored and its setting is varied during the search operation with the restriction that at no time is it set to a value that exceeds the value which corresponds to the last element of the table, or that is less than the value that corresponds to the first element of the table. The length of the table is discussed in the OCCURS clause.

If any of the conditions specified in the WHEN clause cannot be satisfied for any setting of the index within the permitted range, control is passed to imperative-statement-1 of the AT END phrase, when specified, or to the next executable sentence when this phrase is not specified; in either case, the final setting of the index is not predictable. If all conditions can be satisfied, the index indicates an occurrence that allows the conditions to be satisfied, and control passes to imperative-statement-2.

4. After execution of imperative-statement-1, imperative-statement-2, or imperative-statement-3, that does not terminate with a GO TO statement, control passes to the next executable sentence.

5. In Format 2, the index-name that is used for the search operation is the first (or only) index-name that appears in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

6. In Format 1, if the VARYING phrase is not used, the index-name that is used for the search operation is the first (or only) index-name that appears in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

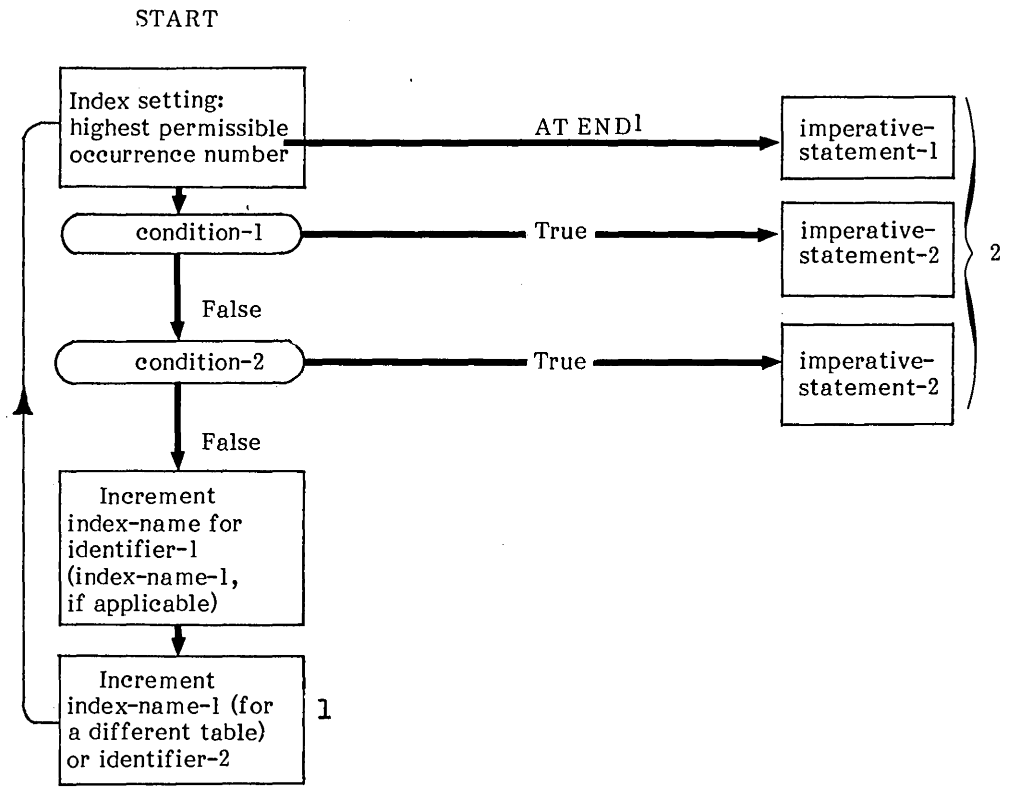
7. In Format 1, if the VARYING index-name-1 phrase is specified, and if index-name-1 appears in the INDEXED BY phrase of identifier-1, that index-name is used for this search. If this is not the case, or if the VARYING identifier-2 phrase is specified, the first (or only) index-name given in the INDEXED BY phrase of identifier-1 is used for the search. In addition, the following operations will occur:

   a. If the VARYING index-name-1 phrase is used, and if index-name-1 appears in the INDEXED BY phrase of another table entry, the occurrence number represented by the index-name associated with identifier-1 is incremented.

   b. If the VARYING identifier-2 phrase is specified, and identifier-2 is an index data item, then the data item referenced by identifier-2 is incremented by the same amount as, and at the same time as, the index associated with identifier-1 is incremented. If identifier-2 is not an index data item, the data item referenced by identifier-2 is incremented by the value (1) at the same time as the index referenced by the index-name associated with identifier-1 is incremented.

8. If identifier-1 is a data item subordinate to a data item that contains an OCCURS clause (providing for a two or three dimensional table), an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. Only the setting of the index-name associated wih identifier-1 (and the data item identifier-2 or index-name-1, if present) is modified by the execution of the SEARCH statement. To search an entire two or three dimensional table is not necessary to execute a SEARCH statement several times. Prior to each execution of a SEARCH statement, SET statements must be executed whenever index-names must be adjusted to appropriate settings.

Figure 4-1 shows a flowchart of the Format 1 SEARCH operation containing two WHEN phrases.


START
Index setting: highest permissible occurrence number
AT END1
imperative-statement-1
condition-1
True
imperative-statement-2
False
condition-2
True
imperative-statement-2
2
False
Increment index-name for identifier-1 (index-name-1, if applicable)
Increment index-name-1 (for a different table) or identifier-2
1

1 - These operations are options included only when specified in the SEARCH statement.

2 - Each of these control transfers is to the next executable sentence unless the imperative-statement ends with a GO TO statement.

Figure 4-1. Flowchart of SEARCH Operation with TWO WHEN Phrases.

THE SET STATEMENT

Function

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements.

General Format

Format 1

```
SET  { identifier-1     [, identifier-2]    ... }  TO  { identifier-3   }
     { index-name-1     [, index-name-2]    ... }      { index-name-3   }
                                                       { integer-1      }
```

Format 2

```
SET    index-name-4     [, index-name-5]    ...  { UP BY   }  { identifier-4 }
                                                 { DOWN BY }  { integer-2    }
```

Syntax Rules

1. All reference to index-name-1, identifier-1, and index-name-4 apply equally to index-name-2, identifier-2, and index-name-5, respectively.

2. Identifier-1 and identifier-3 must name either index data items, or elementary items described as an integer.

3. Identifier-4 must be described as an elementary numeric integer.

4. Integer-1 and integer-2 may be signed. Integer-1 must be positive.

General Rules

1. Index-names are considered related to a given table and are defined by being specified in the INDEXED BY clause.

2. 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-4, index-name-5 is specified, the value of the index both before and after 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. (See THE SEARCH STATEMENT and THE PERFORM STATEMENT in Section 3.)

3. In Format 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 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 or 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.

4. In Format 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.

5. Data in Table 4-1 represents the validity of various operand combinations in the SET statement. The general rule reference indicates the applicable general rule.

Table 4-1. SET Statement Valid Operand Combinations.

Sending Item	Receiving Item [1] Integer Data Item	Index-Name	Index Data Item
Integer Literal	No/3c	Valid/3a	No/3b
Integer Data Item	No/3c	Valid/3a	No/3b
Index-Name	Valid/3c	Valid/3a	Valid/3b[2]
Index Data Item	No/3c	Valid/3a[2]	Valid/3b[2]

1 = Rule numbers under General Rules above are referred to.

2 = No conversion takes place.

SECTION 5

SEQUENTIAL INPUT AND OUTPUT

INTRODUCTION TO THE SEQUENTIAL I-O MODULE

The Sequential I-O module provides a capability to access records of a file in established sequence. The sequence is established as a result of writing the records to the file. It also provides for the specification of re-run points and the sharing of memory areas among files.

LANGUAGE CONCEPTS

Organization

Sequential files are organized such that each record in the file except the first has a unique predecessor record, and each record except the last has a unique successor record. These predecessor-successor relationships are established by the order of WRITE statements when the file is created. Once established, the predecessor-successor relationships do not change except in the case where records are added to the end of the file.

Access Mode

In the sequential access mode, the sequence in which records are accessed is the order in which the records were originally written.

Current Record Pointer

The current record pointer is a conceptual entity used in this document to facilitate specification of the next record to be accessed within a given file. The concept of the current record point has no meaning for a file opened in the output mode. The setting of the current record pointer is affected only by the OPEN and READ statements.

I-O Status

If the FILE STATUS clause is specified in a file control entry, a value is placed into the specified two-character data item during the execution of an OPEN, CLOSE, READ, WRITE, or REWRITE statement and before any applicable USE procedure is executed, to indicate to the COBOL program the status of that input-output operation.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as Status Key 1 and is set to indicate one of the following conditions upon completion of the input-output operation.

'0'	- indicates Successful Completion
'1'	- indicates at End
'2'	- indicates an Invalid Key
'3'	- indicates Permanent Error
'9'	- indicates a Run-Time Error Message

The meaning of the above indications are as follows:

0 - Successful Completion. The input-output statement was successfully executed.

1 - At End. The sequential READ statement was unsuccessfully executed either as a result of an attempt to read a record when no next logical record exists in the file or as a result of the first READ statement being executed for a file described with the OPTIONAL clause, and that file was not available to the program at the time its associated OPEN statement was executed.

2 - Invalid Key. The input-output statement was unsuccessfully executed as a result of one of the following:

Duplicate Key
No Record Found
Boundary Violation

3 - Permanent Error. The input-output statement was unsuccessfully executed as the result of a boundary violation for a sequential file or is the result of an input-output error, such as data check parity error, or transmission error.

9 - Run-Time Error Message. The input-output statement was unsuccessfully executed as a result of a condition that is specified by the Run-Time System Error Message. This value is used only to indicate a condition not indicated by other defined values of status key 1, or by specified combinations of the values of status key 1 and status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as Status Key 2 and is used to further describe the results of the input-output operation. This character will contain a value as follows:

* If no further information is available concerning the input-output operation, then status key 2 contains a value of '0'.

* When status key 1 contains a value of '3' an irrecoverable error has occurred. This is treated as a fatal error by the Operating System.

* When status key 1 contains a value of '9', the value of status key 2 is the Run-Time Error Message number. Appendix J contains some details of the status-key-2 representation. Note that it is not possible to extract this number directly.

Status key 2 is a hexadecimal number which is displayed in ASCII. This returned ASCII character must be converted back to its hexadecimal equivalent by the user.

This ASCII character and its hexadecimal equivalent are located in Table B-2 in Appendix B of the BTOS Reference Manual. Find this character in the table and then convert its corresponding character code (in hex) to decimal. The decimal number will be the COBOL Run-Time error.

Valid Combinations of Status Keys 1 and 2

The valid permissible combinations of the values of status key 1 and status key 2 are shown in the following table. An 'X' at an intersection indicates a valid permissible combination.

Status Key 1	Status Key 2 No Further Information (0)
Successful Completion (0)	X
At End (1)	X
Permanent Error (3)	X
Implementor Defined (9)	RT Error Number

The At END Condition

The AT END condition can occur as a result of the execution of a READ statement. For details of the causes of the condition, see The READ STATEMENT later in this Section.

LINAGE - COUNTER

The reserved word LINAGE-COUNTER is a name for a special register generated by the presence of a LINAGE clause in a file description entry. The implicit description is that of an unsigned integer whose size is equal to integer-1 or the data item referenced by data-name-1 in the LINAGE clause. See The LINAGE Clause later in this Section.

ENVIRONMENTAL DIVISION IN THE SEQUENTIAL I-O MODULE

INPUT-OUTPUT SECTION

The FILE-CONTROL Paragraph

Function

The FILE-CONTROL paragraph names each file and allows specification of other file-related information. (See also Appendix I in this manual).

General Format

```
    FILE-CONTROL    [file-control-entry]  ...
```

The FILE-CONTROL Entry

Function

The file control entry names a file and may specify other file-related information.

General Format

```
    SELECT   [OPTIONAL]    file-name
        ASSIGN TO   { external-file-name-literal }     [ , { external-file-name-literal } ]
                    { file-identifier            }     [   { file-identifier            } ]
    [ ; RESERVE integer-1       [ AREA  ] ]
                                [ AREAS ]
    [ ; ORGANIZATION IS         { SEQUENTIAL      } ]
                                { LINE SEQUENTIAL }
        [; ACCESS MODE IS SEQUENTIAL]

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

Syntax Rules

1. The SELECT clause must be specified first in the file control entry. The clauses which follow the SELECT clause may appear in any order.

2. Each file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph. Each file specified in the file control entry must have a file description entry in the Data Division.

3. If the ACCESS MODE clause is not specified, the ACCESS MODE IS SEQUENTIAL clause is implied.

4. Data-name-1 must be defined in the Data Division as a two-character data item of the category alphanumeric and must not be defined in the File Section.

5. Data-name-1 may be qualified.

6. When the ORGANIZATION IS SEQUENTIAL clause is not specified, the ORGANIZATION IS SEQUENTIAL clause is implied.

7. The OPTIONAL phrase may only be specified for input files. Its specification is required for input files that are not necessarily present each time the object program is executed.

General Rules

1. The ASSIGN clause specifies the association of the file referenced by the SELECT clause to a real file on the system. If the file-name is not enclosed in double quotes, then it must be fully qualified as a file-name (that is real) in the Working-Storage Section.

2. The RESERVE clause allows the user to specify the number of input-output areas allocated. If the RESERVE clause is specified, the number of input-output areas allocated is equal to the value of integer-1.

3. The ORGANIZATION clause specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed.

4. When LINE SEQUENTIAL ORGANIZATION is specified, the file is treated as consisting of variable length records, each record containing one line of data. A line of data is terminated with the new line character. Trailing spaces in a record are truncated.

5. Records in the file are accessed in the sequence dictated by the file organization. This sequence is specified by predecessor-successor record relationships established by the execution of WRITE statements when the file is created or extended.

6. When the FILE STATUS is specified, a value will be moved by the operating system into the data item specified by data-name-1 after the execution of every statement that references that file either explicitly or implicitly. This value indicates the status of execution of the statement. (See I-O STATUS in this Section.)

7. When the file-name is ASSIGNed to a file-identifier, and that file-identifier is then declared in WORKING-STORAGE, B 20 COBOL expects the file-identifier to be followed by (to terminate with) a space.

   Example:
   ```
   01   your-file     PIC X(9)  VALUE  "IND.FILE ".
   ```

The I-O-CONTROL Paragraph

Function

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

General Format

```
I-O-CONTROL
                                                   ⎧⎧[END OF]    ⎧REEL⎫ ⎫             ⎫ ⎤
⎡                                                  ⎪⎨            ⎩UNIT⎭ ⎬             ⎪ ⎥
⎢               ⎡                            ⎤     ⎪⎩ integer-1   RECORDS⎭OF file-name-2⎬ ⎥ D
⎢ ; RERUN   ⎢ ON ⎧file-name-1          ⎫ ⎥ EVERY ⎨ integer-2  CLOCK-UNITS             ⎪ ⎥
⎣               ⎣    ⎩implementor-name⎭ ⎦     ⎩ condition-name                      ⎭ ⎦
  [; SAME     ⎡RECORD     ⎤
              ⎢SORT       ⎥  AREA FOR file-name-3 , file-name-4 ...  ]  ...
              ⎣SORT-MERGE ⎦
  [; MULTIPLE FILE TAPE CONTAINS file-name-5  [POSITION integer-3]

             [file-name-6  [POSITION integer-4]]   ...]...
```

Syntax Rules

1. The I-O-CONTROL paragraph is optional.

2. File-name-1 must be a sequentially organized file.

3. The END OF REEL/UNIT clause may only be used if file-name-2 is a sequentially organized file.

4. When either the integer-1 RECORDS clause of the integer-2 CLOCK-UNITS clause is specified, implementor-name must be given in the RERUN clause.

5. More than one RERUN clause may be specified for a given file-name-2 subject to the following restrictions:

   a. When multiple integer-1 RECORD clauses are specified, no two of them can specify the same file-name-2.

   b. When multiple END OF REEL or END OF UNIT clauses are specified, no two of them may specify the same file-name-2.

6. The two forms of the SAME clause (SAME AREA, SAME RECORD AREA) are considered separately in the following:

   More than one SAME clause may be included in a program, however:

   a. A file-name must not appear in more than one SAME AREA clause.

   b. A file-name must not appear in more than one SAME RECORD AREA clause.

   c. If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in that SAME AREA clause must appear in the SAME RECORD AREA clause. However, additional file-names not appearing in that SAME AREA clause may also appear in that SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any given time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any given time.

7. The files referenced in the SAME AREA clause need not all have the same organization or access.

General Rules

1. The RERUN clause is treated as for documentation purposes only.
2. The SAME AREA clause specifies that two or more files are to use the same memory area during processing. The area being stored includes all storage area assigned to the files specified; therefore, it is not valid to have more than one of the files open at the same time. (See Syntax Rule 6c.)
3. The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All of the files may be open at the same time. A logical record in the SAME RECORD AREA is considered as a logical record of each opened output file whose file-name appears in this SAME RECORD AREA clause and of the most recently read input file whose file-name appears in this SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the area, i.e., records are aligned on the leftmost character position.
4. The MULTIPLE FILE clause is treated as for documentation purposes only.

DATA DIVISION IN THE SEQUENTIAL I-O MODULE

FILE SECTION

In a L/II COBOL program the file description entry (FD) represents the highest level of organization in the File Section. The File Section header is followed by a file description entry consisting of a level indicator (FD), a file-name and a series of independent clauses. The FD clauses specify the size of the logical and physical records, the presence or absence of label records, the value of implementor-defined label items, the names of the data records which comprise the file. The entry itself is terminated by a period.

RECORD DESCRIPTION STRUCTURE

A 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 if required, followed by a series of independent clauses as required. A record description has a hierarchical structure and therefore the clauses used with an entry may vary considerably, depending upon whether or not it is followed by subordinate entries. The structure of a record description is defined in CONCEPT OF LEVELS in Section 2, while the elements allows in a record description are shown in the Data Description - Complete Entry Skeleton in Section 3.

THE FILE DESCRIPTION - COMPLETE ENTRY SKELETON

Function

The file description furnishes information concerning the physical structure, identification, and record names pertaining to a given file.

General Format

```
FD   file-name

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

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

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

    [ ; VALUE OF data-name-1 IS          literal-1

           [, data-name-2  IS            literal-2] ... ]

   [ ; DATA      { RECORD IS   }
                 { RECORDS ARE } data-name-3          [, data-name-4] ... ]

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

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

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

Syntax Rules

1. The level indicator FD identifies the beginning of a file description and must precede the file-name.

2. The clauses which follow the name of the file are optional, and their order of appearance is immaterial.

3. One or more record description entries must follow the file description entry.

THE BLOCK CONTAINS CLAUSE

Function

The BLOCK CONTAINS clause specifies the size of a physical record.

General Format

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

General Rule

This clause is required for documentation purposes only.

THE CODE-SET CLAUSE

Function

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

General Format

$\underline{\text{CODE-SET}}$ IS alphabet-name

Syntax Rules

1. When the CODE-SET clause is specified for a file, all data in that file must be described as usage is DISPLAY and any signed numeric data must be described with the SIGN IS SEPARATE clause.

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

3. The CODE-SET clause may only be specified for non-disk files.

General Rule

The CODE-SET clause is specified for documentation purposes only.

THE DATA RECORDS CLAUSE

Function

The DATA RECORDS clause serves only as documentation for the names of data records with their associated file.

General Format

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

Syntax Rule

Data-name-1 and data-name-2 are the names of data records and must have 01 level-number record descriptions, with the same names, associated with them.

General Rule

The DATA RECORDS clause is specified for documentation purposes only.

THE LABEL RECORDS CLAUSE

Function

The LABEL RECORDS clause specifies whether labels are present.

General Format

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

Syntax Rule

This clause is optional in every file description entry.

General Rule

This clause is used for documentation purposes only.

THE LINAGE CLAUSE

Function

The LINAGE CLAUSE provides a means for specifying the depth of a logical page in terms of number of lines. It also provides for specifying the size of the top and bottom margins on the logical page, and the line number, within the page body, at which the footing area begins.

General Format

```
LINAGE IS  { data-name-1 }  LINES   , [ WITH FOOTING AT  { data-name-2 } ]
           { integer-1   }                               { integer-2   }

   [ , LINES AT TOP  { data-name-3 } ] [ ,LINES AT BOTTOM  { data-name-4 } ]
                     { integer-3   } ]                     { integer-4   }
```

Syntax Rules

1. Data-name-1, data-name-2, data-name-3, data-name-4 must reference elementary unsigned numeric integer data items.

2. The value of integer-1 must be greater than zero.

3. The value of integer-2 must not be greater than integer-1.

4. The value of integer-3, integer-4 may be zero.

General Rules

1. The LINAGE clause provides a means for specifying the size of a logical page in terms of number of lines. The logical page size is the sum of the values referenced by each phrase except the FOOTING phrase. If the LINES AT TOP or LINES AT BOTTOM phrases are not specified, the values for these functions are zero. If the FOOTING phrase is not specified, the assumed value is equal to integer-1, or the contents of the data item referenced by data-name-1, whichever is specified.

   There is not necessarily any relationship between the size of the logical page an the size of a physical page.

2. The value of integer-1 on the data item referenced by data-name-1 specifies the number of lines that can be written and/or spaced on the logical page. The value must be greater than zero. That part of the logical page in which these lines can be written and/or spaced is called the page body.

3. The value of integer-3 or the data item referenced by data-name-3 specifies the number of lines that comprise the top margin on the logical page. The value may be zero.

4. The value of integer-4 or the data item referenced by data-name-4 specifies the number of lines that comprise the bottom margin on the logical page. The value may be zero.

5. The value of integer-2 or the data item referenced by data-name-2 specifies the line number within the page body at which the footing area begins. The value must be greater than zero and not greater than the value of integer-1 or the data item referenced by data-name-1.

The footing area comprises the area of the logical page between the line represented by the value of integer-2 or the data item referenced by data-name-2 and the line represented by the value of integer-1 or the data item referenced by data-name-1, inclusive.

6. The value of integer-1, integer-3, and integer-4, if specified, will be used at the time the file is opened by the execution of an OPEN statement with the OUTPUT phrase, to specify the number of lines that comprise each of the indicated sections of a logical page. The value of integer-2, if specified, will be used at that time to define the footing area. These values are used for all logical pages written for the file during a given execution of the program.

7. The values of the data items referenced by data-name-1, data-name-3, and data-name-4, if specified, will be used as follows:

   a. The values of the data items, at the time an OPEN statement with the OUTPUT phrase is executed for the file, will be used to specify the number of lines that are to comprise each of the indicated sections for the first logical page.

   b. The values of the data items, at the time a WRITE statement with the ADVANCING PAGE phrase is executed or page overflow condition occurs (See The WRITE STATEMENT), will be used to specify the number of lines that are to comprise each of the indicated sections for the next logical page.

8. The value of the data item referenced by data-name-2, if specified, at the time an OPEN statement with the OUTPUT phrase is executed for the file, will be used to define the footing area for the first logical page. At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, it will be used to define the footing area for the next logical page.

9. A LINAGE-COUNTER is generated by the presence of a LINAGE clause. The value in the LINAGE-COUNTER at any given time represents the line number at which the device is positioned within the current page body. The rules governing the LINAGE-COUNTER are as follows:

   a. A separate LINAGE-COUNTER is supplied for each file described in the File Section whose file description entry contains a LINAGE clause.

   b. LINAGE-COUNTER may be referenced, but may not be modified, by Procedure Division statements. Since more than one LINAGE-COUNTER may exist in a program, the user must qualify LINAGE-COUNTER by file-name when necessary.

   c. LINAGE-COUNTER is automatically modified, according to the following rules, during the execution of a WRITE statement to an associated file:

      * When the ADVANCING PAGE phrase of the WRITE statement is specified, the LINAGE-COUNTER is automatically reset to one.

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

* When the ADVANCING phrase of the WRITE statement is not specified, the LINAGE-COUNTER is incremented by the value one. (See The WRITE STATEMENT.)

* The value of LINAGE-COUNTER is automatically reset to one when the device is repositioned to the first line that can be written on for each of the succeeding logical pages. (See The WRITE STATEMENT.)

d. The value of LINAGE-COUNTER is automatically set to one at the time an OPEN statement is executed for the associated file.

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

THE RECORD CONTAINS CLAUSE

Function

The RECORD CONTAINS clause specifies the size of data records.

General Format

$$\underline{\text{RECORD}}\ \text{CONTAINS integer-1}\ \underline{\text{TO}}\quad\text{integer-2 CHARACTERS}$$

General Rule

The size of each data record is completely defined within the record description entry, therefore this clause is never required. The RECORD CONTAINS clause is specified for documentation purposes only.

THE VALUE OF CLAUSE

Function

The VALUE OF clause specializes the description of an item in the label records associated with a file.

General Format

$$\underline{\text{VALUE}}\ \underline{\text{OF}}\ \text{data-name-1 IS} \left\{\begin{matrix}\text{data-name-2}\\ \text{literal-1}\end{matrix}\right\}$$
$$\left[\ \text{, data-name-3 IS} \left\{\begin{matrix}\text{data-name-4}\\ \text{literal-2}\end{matrix}\right\}\right]\ \ldots$$

Syntax Rules

1. Data-name-2, data-name-3, etc. should be qualified when necessary but cannot be subscripted or indexed, nor can they be items described with the USAGE IS INDEX clause.

2. Data-name-2, data-name-3, etc. must be in the Working-Storage Section.

General Rules

1. This clause is used for documentation purposes only.

2. On input data-name-1 is checked against data-name-2 or literal-1 as specified and data-name-3 against data-name-4 or literal-2 as specified, etc.

   On output data-name-2 or literal-1 are substituted for data-name-1 as specified and data-name-4 or literal-2 from data-name-3, etc.

3. A figurative constant may be substituted in the format above wherever a literal is specified.

PROCEDURE DIVISION IN THE SEQUENTIAL I-O MODULE

THE CLOSE STATEMENT

Function

The CLOSE statement terminates the processing of files.

General Format

$$\underline{\text{CLOSE}}\ \text{file-name-1}\ \left[\begin{array}{l}\left\{\begin{array}{l}\underline{\text{REEL}}\\ \underline{\text{UNIT}}\end{array}\right\}\ \left[\begin{array}{l}\text{WITH } \underline{\text{NO}}\ \underline{\text{REWIND}}\\ \text{FOR } \underline{\text{REMOVAL}}\end{array}\right]\\ \text{WITH}\ \left\{\begin{array}{l}\underline{\text{NO}}\ \underline{\text{REWIND}}\\ \underline{\text{LOCK}}\end{array}\right\}\end{array}\right]$$

$$\left[\text{, file-name-2}\ \left[\begin{array}{l}\left\{\begin{array}{l}\underline{\text{REEL}}\\ \underline{\text{UNIT}}\end{array}\right\}\ \left[\begin{array}{l}\text{WITH } \underline{\text{NO}}\ \underline{\text{REWIND}}\\ \text{FOR } \underline{\text{REMOVAL}}\end{array}\right]\\ \text{WITH}\ \left\{\begin{array}{l}\underline{\text{NO}}\ \underline{\text{REWIND}}\\ \underline{\text{LOCK}}\end{array}\right\}\end{array}\right]\right]$$

Syntax Rule

The magnetic tape phrases REEL, UNIT, WITH NO REWIND, FOR REMOVAL, etc. must only be used for sequential files. All magnetic tape phrases are for documentation purposes only.

General Rules

1. A CLOSE statement may only be executed for a file in an open mode.

2. The action taken if the file is in the open mode when a STOP RUN statement is executed is to close the file. The action taken for a file that has been opened in a called program and not closed in that program prior to the execution of a CANCEL statement for that program is to close the file.

3. If a CLOSE statement has been executed for a file, no other statement can be executed that references that file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed.

4. Following the successful execution of a CLOSE statement the record area associated with file-name is no longer available. The unsuccessful execution of such a CLOSE statement leaves the availability of the record area undefined.

THE OPEN STATEMENT

Function

The OPEN statement initiates the processing of files. It also performs checking and/or writing of labels and other input-output operations.

General Format

```
       { INPUT file-name-1     [REVERSED          ]                }
       {                       [WITH NO REWIND    ]                }
       {                                                           }
       {      [, file-name-2   [REVERSED          ]] . . .         }
       {      [                [WITH NO REWIND    ]]               }
       {                                                           }
       { OUTPUT file-name-3    [WITH NO REWIND    ]                }
OPEN   {      [, file-name-4   [WITH NO REWIND    ]] . . .         }  . . .
       {                                                           }
       { I-O file-name-5        [, file-name-6]        . . .       }
       {                                                           }
       { EXTEND file-name-7     [, file-name-8]        . . .       }
```

Syntax Rules

1. The REVERSED and NO REWIND phrases can only be used with sequential files and are for documentation purposes only.

2. The I-O phrase can be used only for disk files, except for files in Line Sequential organization.

3. The EXTEND phrase can be used only for seqeuntial files, and Line Sequential files.

4. The EXTEND phrase must not be specified with multiple file reels.

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

<u>General Rules</u>

1. The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode.

2. The successful execution of an OPEN statement makes the associated record area available to the program.

3. Prior to the successful execution of an OPEN statement for a given file, no statement can be executed that references that file, either explicitly or implicitly.

4. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statement. In Table 5-1, 'X' at an intersection indicates that the specified statement, used in the sequential access mode, may be used with the sequential file organization and open mode given at the top of the column.

Table 5-1. Permissible Combinations of Statements and OPEN Modes for Sequential I/O.

Statement	Open Mode			
	Input	Output	Input-Output1	Extend
READ	X		X	
WRITE		X		X
REWRITE			X	
1 - This OPEN mode is not supported for ORGANIZATION line sequential files.				

5. A file may be opened with the INPUT, OUTPUT, EXTEND and I-O phrases in the same program. Following the initial execution of an OPEN statement for a file, each subsequent OPEN statement execution for that same file must be preceded by the execution of a CLOSE statement for that file.

6. Execution of the OPEN statement does not obtain or release the first data record.

7. The ASSIGNed name in the SELECT statement for a file is processed as follows:

   a. When the INPUT phrase is specified, the execution of the OPEN statement causes the ASSIGNed name to be checked in accordance with the operating system conventions for opening files for input.

   b. When the OUTPUT phrase is specified, the execution of the OPEN statement causes the ASSIGNed name to be written in accordance with the operating system conventions for opening files for output.

8. The file description entry for file-name-1, file-name-2, file-name-5, file-name 6, file-name-7, and file-name-8 must be equivalent to that used when this file was created.

9. If an input file is designated with the OPTIONAL phrase in its SELECT clause, the object program causes an interrogation for the presence or absence of this file. If the file is not present, the first READ statement for this file causes the AT END condition to occur.

10. If the storage medium for the file permits rewinding, execution of the OPEN statement causes the file to be positioned at its beginning.

11. For files being opened with the INPUT or I-O phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set such that the next executed READ statement for the file will result in an AT END condition. If the file does not exist, OPEN INPUT will cause an error status.

12. When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statement referencing the file will add records to the file as though the file had been opened with the OUTPUT phrase.

13. The I-O phrase permits the opening of a disk file for both input and output operations except for files in ORGANIZATION LINE SEQUENTIAL. If the file does not exist it will be created.

14. Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At that time the associated file contains no data records. If a file of the same name exists, it will be deleted. If write-protected, an error will occur.

THE READ STATEMENT

Function

The READ statement makes available the next logical record from a file.

General Format

READ file-name RECORD [INTO identifier] [; AT END imperative-statement]

Syntax Rules

1. The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions. The storage area associated with identifier and the record area associated with file-name must not be the same storage area.

2. The AT END phrase must be specified if no applicable USE procedure is specified for file-name.

General Rules

1. The associated file must be open in the INPUT or I-O mode at the time this statement is executed. (See The OPEN STATEMENT in this Section.)

2. The record to be made available by the READ statement is determined as follows:

   a. If the current record pointer was positioned by the execution of the OPEN statement, the record pointed to by the current record pointer is made available.

   b. If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

3. The execution of the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. (See I-O STATUS in this Section.)

4. Regardless of the method used to overlap access time with processing time, the concept of the READ statement is unchanged in that a record is available to the object program prior to the execution of any statement following the READ statement.

5. When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the READ statement.

6. If the INTO phrase is specified, the record being read is moved from the record area to the area specified by identifier according to the rules specified for the MOVE statement. The implied MOVE does not occur if the execution of the READ statement was unsuccessful. Any subscripting or indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

7. When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with identifier.

8. If, at the time of execution of a READ statement, the position of current record pointer for that file is undefined, the execution of that READ statement is unsuccessful.

9. If the end of a reel or unit is recognized during the execution of a READ statement, an end-of-file status condition exists.

   a. The standard ending reel/unit label procedure.

   b. A reel/unit swap.

   c. The standard beginning reel/unit label procedure.

   d. The first data record of the new reel/unit is made available.

10. If a file described with the OPTIONAL clause is not present at the time the file is opened, then at the time of the execution of the first READ statement for the file, the AT END condition occurs and the execution of the READ statement is unsuccessful. The standard end of file procedures are not performed. (See The FILE-CONTROL paragraph and the OPEN and the USE statement descriptions in this Section.) Execution of the program then proceeds as in General Rule 12.

11. If, at the time of the execution of a READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful. (See I-O STATUS.)

12. When the AT END condition is recognized the following actions are taken in the specified order:

    a. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an AT END condition. (See I-O STATUS.)

    b. If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative-statement. Any USE procedure specified for this file is not executed.

    c. If the AT END phrase is not specified, then a USE procedure must be specified, either explicitly or implicitly, for this file and that procedure is executed.

    When the AT END condition occurs, execution of the input-output statement which caused the condition is unsuccessful.

13. Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined.

14. When the AT END condition has been recognized, a READ statement for that file must not be executed without first executing a successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

THE REWRITE STATEMENT

Function

The REWRITE statement logically replaces a record existing in a disk file.

General Format

REWRITE record-name [FROM identifier]

Syntax Rules

1. Record-name and identifier must not refer to the same storage area.

2. Record-name is the name of a logical record in the File Section of the Data Division and may be qualified.

General Rules

1. The file associated with record-name must be a disk file and must be open in the I-O mode at the time of execution of this statement. (See The OPEN STATEMENT in this Section.)

2. The last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a successfully executed READ statement. The operating system logically replaces the record that was accessed by the READ statement.

3. The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

4. The logical record released by a successful execution of the REWRITE statement is no longer available in the record area unless the associated file is saved in a SAME RECORD AREA clause. In this case, not only is the record still available to the program in the record area as a record of this file, but as a record of other files named in the SAME RECORD AREA clause.

5. The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

   MOVE identifier TO record-name

   followed by the execution of the same REWRITE statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of the REWRITE statement.

6. The current record pointer is not affected by the execution of a REWRITE statement.

7. The execution of the REWRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. (See I-O STATUS in this Section.)

8. The REWRITE statement cannot be used with line sequential files.

THE USE STATEMENT

Function

The USE statement specifies procedures for input-output error handling that are in addition to the standard procedures provided by the input-output control system.

General Format

$$\underline{\text{USE}}\ \underline{\text{AFTER}}\ \text{STANDARD} \left\{\begin{array}{l}\underline{\text{EXCEPTION}}\\ \underline{\text{ERROR}}\end{array}\right\}\ \underline{\text{PROCEDURE}}\ \text{ON} \left\{\begin{array}{l}\text{file-name-1 [, file-name-2]...}\\ \underline{\text{INPUT}}\\ \underline{\text{OUTPUT}}\\ \underline{\text{I-O}}\\ \underline{\text{EXTEND}}\end{array}\right\}$$

Syntax Rules

1. A USE statement, when present, must immediately follow a section header in the declarative section and must be followed by a period followed by a space. The remainder of the section must consist of zero, one or more procedural paragraphs that define the procedure to be used.

2. The USE statement itself is never executed; it merely defines the conditions calling for the execution of the USE procedures.

General Rules

1. If the AT END phrase has not been specified in the input-output statement, the designated procedures are executed by the input-output system after completing the standard input-output error routine upon recognition of the AT END condition.

2. After execution of a USE procedure, control is returned to the invoking routine.

3. Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declarative portion, except that PERFORM statements may refer to a USE statement or to the procedures associated with such a USE statement.

4. Within a USE procedure, there must not be the execution of any statement that would cause the execution of a USE procedure that had previously been invoked and had not yet returned control to the invoking routine.

THE WRITE STATEMENT

Function

The WRITE statement releases a logical record for an output file. It can also be used for vertical positioning of lines within a logical page.

General Format

```
WRITE record-name   [FROM identifier-1]
 _                                                              _
|                             /  / integer      \ [ LINE  ]  \   |
|                             |  \ identifier-2 / [ LINES ]  |   |
|  / BEFORE \    ADVANCING    <                              >   |
|  \ AFTER  /                 |  TAB                         |   |
|                             |  mnemonic-name               |   |
|_                            \  PAGE                        /  _|

  [ ; AT  / END-OF-PAGE \      imperative-statement ]
  [       \ EOP         /                           ]
```

Syntax Rules

1. Record-name and identifier-1 must not reference the same storage area.

2. When TAB is specified the result is to cause the paper to throw to the standard vertical tabulation position. A user-defined mnemonic-name can be used instead of TAB if they are associated in the SPECIAL-NAMES paragraph.

3. The record-name is the name of a logical record in the File Section of the Data Division.

4. When identifier-2 is used in the ADVANCING phrase, it must be the name of an elementary integer data item.

5. Integer on the value of the data item referenced by identifier-2 may be zero.

6. If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file description entry for the associated file.

7. The word END-OF-PAGE and EOP are equivalent.

8. The ADVANCING TAB phrase cannot be specified when writing a record to a file whose file description entry contains the LINAGE clause.

General Rules

1. The associated file must be open in the OUTPUT OR EXTEND mode at the time of the execution of this statement. (See The OPEN STATEMENT in this Section.)

2. The logical record released by the execution of the WRITE statement is no longer available in the record area unless the associated file is named in a SAME RECORD AREA clause or the execution of the WRITE statement was unsuccessful due to a boundary violation.

   The logical record is also available to the program as a record of other files referenced in the same SAME RECORD AREA clause as the associated output file, as well as to the file associated with record-name.

3. The results of the execution of the WRITE statement with the FROM phrase is equivalent to the execution of:

   a. The statement:

      MOVE identifier-1 TO record-name

      according to the rules specified for the MOVE statement, followed by:

   b. The same WRITE statement without the FROM phrase.

      The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement.

      After execution of the WRITE statement is complete, the information in the area referenced by identifier-1 is available, even though the information in the area referenced by record-name may not be. (See General Rule 2.)

4. The current record pointer is unaffected by the execution of a WRITE statement.

5. The execution of the WRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. (See I-O STATUS in this Section.)

6. The maximum record size for a file is established at the time the file is created and must not subsequently be changed.

7. The number of character positions on a disk required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

8. The execution of the WRITE statement releases a logical record to the operating system.

9. The ADVANCING phrase allows control of the vertical positioning of each line on a representation of a printed page.

   a. With ORGANIZATION SEQUENTIAL if the ADVANCING phrase is not used, automatic advancing is provided when output is directed to a list-device to act as if the user had specified AFTER ADVANCING 1 LINE. If the ADVANCING phrase is used, advancing is provided as follows:

      i. If integer is specified, the representation of the printed page is advanced the number of lines equal to the value of integer.

      ii. If the BEFORE phrase is used, the line is presented before the representation of the printed page is advanced according to rule a above.

      iii. If the AFTER phrase is used, the line is presented after the representation of the printed page is advanced according to rule a above.

      iv. If PAGE is specified, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page.

   b. With ORGANIZATION LINE SEQUENTIAL, if the ADVANCING phrase is not used, automatic advancing of one line is provided.

      If the ADVANCING phrase is used, advancing is provided according to rules 9a(i) through 9a(iv) above.

   c. If the BEFORE phrase is used, the line is presented before the representation of the printed page is advanced according to rule a above.

   d. If the AFTER phrase is used, the line is presented after the representation of the printed page is advanced according to rule a above.

e. If PAGE is specified, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page. If the record to be written is associated with a file whose description entry contains a LINAGE clause, the repositioning is to the first line that can be written on the next logical page as specified in the LINAGE clause. If the record to be written is associated with a file whose file description entry does not contain a LINAGE clause, the repositioning to the next logical page is accomplished in accordance with an implementor-defined technique. If page has no meaning in conjunction with a specific device, then advancing will be provided by the implementor to act as if the user had specified BEFORE or AFTER (depending on the phrase used) ADVANCING 1 LINE.

10. If the logical end of the representation of the printed page is reached during the execution of a WRITE statement with the END-OF-PAGE phrase, the imperative-statement specified in the END-OF-PAGE phrase is executed. The logical end is specified in the LINAGE clause associated with record-name.

11. An end-of-page condition is reached whenever the execution of a given WRITE statement with the END-OF-PAGE phrase occurs when the execution of such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the value specified by integer-2 or the data item referenced by data-name-2 of the LINAGE clause, if specified. In this case, the WRITE statement is executed and then the imperative statement in the END-OF-PAGE phrase is executed.

    An automatic page overflow condition is reached whenever the execution of a given WRITE statement (with or without an END-OF-PAGE phrase) cannot be fully accommodated within the current page body.

    This occurs when a WRITE statement, if executed, would cause the LINAGE-COUNTER to exceed the value specified by integer-1 or the data item referenced by data-name-1 of the LINAGE clause. In this case, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the first line that can be written on the next logical page as specified in the LINAGE clause. The imperative statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the device has been repositioned.

    If integer-2 or data-name-2 of the LINAGE clause is not specified, no end-of-page condition distinct from the page overflow condition is detected. In this case, the end-of-page condition and page overflow condition occur simultaneously.

    If integer-2 or data-name-2 of the LINAGE clause is specified, but the execution of a given WRITE statement would cause LINAGE-COUNTER to simultaneously exceed the value of both integer-2 or the data item referenced by data-name-2 and integer-1 or the data item referenced by data-name-1, then the operation proceeds as if integer-2 or data-name-2 had not been specified.

12. When an attempt is made to write beyond the externally defined boundaries of a sequential file, an exception condition exists and the contents of the record area are unaffected. The following action takes place:

    a. The value of the FILE STATUS data item, if any, of the associated file is set to a value indicating a boundary violation. (See I-O STATUS in this Section.)

    b. If a USE AFTER STANDARD EXCEPTION declarative is explicitly or implicitly specified for the file, that declarative procedure will then be executed.

    c. If a USE AFTER STANDARD EXCEPTION declarative is not explicitly or implicitly specified for the file, the result is undefined.

SECTION 6

RELATIVE INPUT AND OUTPUT

INTRODUCTION TO THE RELATIVE I-O MODULE

The Relative I-O module provides a capability to access records of a mass storage file in either a random or sequential manner. Each record in a relative file is uniquely identified by an integer value greater than zero which specifies the record's ordinal position in the file.

LANGUAGE CONCEPTS

Organization

Relative file organization is permitted only on disk devices. A relative file consists of records which are identified by relative record numbers. The file may be thought of as composed of a serial string of areas, each capable of holding a logical record. Each of these areas is denominated by a relative record number. Records are stored and retrieved based on this number. For example, the tenth record area, whether or not records have been written in the first through the ninth record areas.

Access Modes

In the sequential access mode, the sequence in which records are accessed is the ascending order of the relative record numbers of all records which currently exist within the file.

In the random access mode, the sequence in which records are accessed is controlled by the programmer. The desired record is accessed by placing its relative record number in a relative key data item.

In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements.

Current Record Pointer

The current record pointer is a conceptual entity used in this document to facilitate specification of the next record to be accessed within a given file. The concept of the current pointer has no meaning for a file opened in the output mode. The setting of the current record pointer is affected only by the OPEN, START and READ statements.

I-O Status

If the FILE STATUS clause is specified in a file control entry, a value is placed into the specified two-character data item during the execution of an OPEN, CLOSE, READ, WRITE, DELETE or START statement and before any applicable USE procedure is executed, to indicate to the COBOL program the status of that input-output operation.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1 and is set to indicate one of the following conditions upon completion of the input-output operation.

'0' -	indicates Successful Completion
'1' -	indicates At End
'2' -	indicates Invalid Key
'3' -	indicates Permanenet Error
'9' -	indicates Run-Time Error Message

The meaning of the above indications are as follows:

'0' - Successful Completion. The input-output statement was successfully executed.

'1' - At End. The Format 1 READ statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

'2' - Invalid Key. The input-output statement was unsuccessfully executed as a result of one of the following:

- * Duplicate Key
- * No Record Found
- * Boundary Violation

'3' - Permanent Error. The input-output statement was unsuccessfully executed as the result of an input-output error, such as data check, parity error or transmission error.

'9' - Run-Time Error Message. The input-output statement was unsuccessfully executed as the result of a condition that is specified by the Run-Time System. This value is used only to indicate a condition not indicated by other defined values of status key 1, or by specified combinations of the values of status key 1 and status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2 and is used to further describe the results of the input-output operation. This character contains a value as follows:

* If no further information is available concerning the input-output operation, then status key 2 contains a value of '0'.

* When status key 1 contains a value of '2' indicating an INVALID KEY condition, status key 2 is used to designate the cause of that condition by the following values:

2 - Indicates a duplicate key value. An attempt has been made to write a record that would create a duplicate key in a relative file.

3 - Indicates no record found. An attempt has been made to access a record, identified by a key, and that record does not exist in the file.

4 - Indicates a boundary violation. An attempt has been made to write beyond the externally-defined boundaries of a relative file. This is normally treated as a fatal error by the Operating System.

* When status key 1 contains a value of '9', the value of status key 2 is the Run-Time Error Message number. Appendix J contains some details of the status-key-2 representation. Note that it is not possible to extract this number directly.

Status key 2 is a hexadecimal number which is displayed in ASCII. This returned ASCII character must be converted back to its hexadecimal equivalent by the user.

This ASCII character and its hexadecimal equivalent are located in Table B-2 in Appendix B of the BTOS Reference Manual. Find this character in the table and then convert its corresponding character code (in hex) to decimal. The decimal number will be the COBOL Run-Time error.

Valid Combinations of Status Keys 1 and 2

The valid permissible combinations of the values of status key 1 and status key 2 are shown in the table. An 'X' at an intersection indicates a valid permissible combination.

Status Key 1	Status Key 2			
	No Further Information (0)	Duplicate Key (2)	No Record Found (3)	Boundary Violation (4)
Successful Completion (0)	X			
At End (1)	X			
Invalid Key (2)		X	X	X
Permanent Error (3)	X			
Implementor Defined (9)	Run-Time System Error Message Number			

The INVALID KEY Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE or DELETE statement. For details of the causes of the condition, see The START Statement, The READ Statement, The WRITE Statement, The REWRITE Statement, and The DELETE Statement later in this Section.

When the INVALID KEY condition is recognized, the Operating System takes these actions in the following order:

1. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an INVALID KEY condition. (See I-O Status in this Section.)

2. If the INVALID KEY phrase is specified in the statement causing the condition, control is transferred to the INVALID KEY imperative statement. Any USE procedure specified for this file is not executed.

3. If the INVALID KEY phrase is not specified, but a USE procedure is specified, either explicitly or implicitly, for this file, that procedure is executed.

When the INVALID KEY condition occurs, execution of the input-output statement which recognized the condition is unsuccessful, and the file is not affected.

The AT END Condition

The AT END condition can occur as a result of the execution of a READ statement. For details of the causes of the condition, see The READ Statement later in this Section.

INPUT-OUTPUT SECTION

The File-Control Paragraph

Function

The FILE-CONTROL paragraph name each file and allows specifications of other file-related information.

General Format

```
FILE CONTROL    [file-control-entry] ...
```

The File Control Entry

Function

The file control entry names a file and may specify other file-related information.

General Format

```
SELECT file-name

     ASSIGN TO  { external-file-name-literal }
                { file-identifier            }
              [    { external-file-name-literal } ]
              [ ,  { file-identifier            } ]

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

     ; ORGANIZATION IS RELATIVE

                            SEQUENTIAL    ,RELATIVE KEY IS data-name-1

     ; ACCESS MODE IS         RANDOM      , RELATIVE KEY IS data-name-1
                              DYNAMIC

     [; FILE STATUS IS data-name-2].
```

Syntax Rules

1. The SELECT clause must be specified first in the file control entry. The clauses which follow the SELECT clause may appear in any order.

2. Each file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph. Each file specified in the file control entry must have a file description entry in the Data Division.

3. If the ACCESS MODE clause is not specified, the ACCESS MODE IS SEQUENTIAL clause is implied.

4. Data-name-2 must be defined in the Data Division as a two-character data item of the category alphanumeric and must not be defined in the File Section, the Report Section, or the Communication Section.

5. Data-name-1 and data-name-2 may be qualified.

6. If a relative file is to be referenced by a START statement, the RELATIVE KEY phrase must be specified for that file.

7. Data-name-1 must not be defined in a record description entry associated with that file-name.

8. The data item referenced by data-name-1 must be defined as an unsigned integer.

General Rules

1. The ASSIGN clause specifies the association of a file referenced by the SELECT clause to a real file on the system. The first assignment takes place. Subsequent assignments within any one ASSIGN clause are for documentation purposes only.

2. The RESERVE clause allows the user to specify the number of input-output areas allocated. If the RESERVE clause is specified, the number of input-output areas allocated is equal to the value of integer-1. The RESERVE clause is treated as for documentation purposes only.

3. The ORGANIZATION clause specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed.

4. When the access mode is sequential, records in the file are accessed in the sequence dictated by the file organization. This sequence is the order of ascending relative record numbers of existing records in the file.

5. When the FILE STATUS clause is specified, a value will be moved by the operating system into the data item specified by data-name-2 after the execution of every statement that references that file either explicitly or implicitly. This value indicates the status of execution of the statement. (See I-O Status in this Section.)

6. If the access mode is random, the value of the RELATIVE KEY data item indicates the record to be accessed.

7. When the access mode is dynamic, records in the file may be accessed sequentially and/or randomly. (See General Rules 4 and 6.)

8. All records stored in a relative file are uniquely identified by relative record numbers. The relative record number of a given record specifies the record's logical ordinal position in the file. The first logical record has a relative record number of 1, and subsequent logical records have relative record numbers of 2, 3, 4, ...

9. The data item specified by data-name-1 is used to communicate a relative record number between the user and the Operating System.

10. When the file-name is ASSIGNed to a file-identifier, and that file-identifier is then declared in WORKING-STORAGE, B 20 COBOL expects the file-identifier to be followed by (to terminate with) a space.

    Example:
    ```
    01   your-file    PIC X(9)  VALUE  "IND.FILE ".
    ```

The I-O-CONTROL Paragraph

Function

The I-O-CONTROL paragraph specifies the points at which rerun is to be established and the memory area which is to be shared by different files.

General Format

```
I-O-CONTROL

[                {file-name-1        }         {integer-1 RECORDS OF file-name-2}  ]
[ ; RERUN ON     {implementor-name   }  EVERY  {integer-2 CLOCK-UNITS           }  ]
[                                              {condition-name                  }  ]

[;SAME [RECORD] AREA FOR file-name-3 [, file-name-4] ....] .
```

Syntax Rules

1. The I-O-CONTROL paragraph is optional.

2. File-name-1 must be a sequentially organized file.

3. When either the integer-1 RECORDS clause or the integer-2 CLOCK-UNITS clause is specified, implementor-name must be given in the RERUN clause.

4. More than one RERUN clause may be specified for a given file-name-2, subject to the following restriction

   When multiple integer-1 RECORDS clauses are specified, no two of them may specify the same file-name-2.

5. Only one RERUN clause containing the CLOCK-UNITS clause may be specified.

6. The two forms of the SAME clause (SAME AREA, SAME RECORD AREA) are considered separately in the following:

   a. A file-name must not appear in more than one SAME AREA clause.

   b. A file-name must not appear in more than one SAME RECORD AREA clause.

   c. If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in that SAME AREA clause must appear in the SAME RECORD AREA clause. However, additional file-names not appearing in that SAME AREA clause may also appear in that SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any given time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any given time.

7. The files referenced in the SAME AREA or SAME RECORD AREA clauses need not all have the same organization or access.

General Rules

1. The RERUN clause is treated as for documentation purposes only.

2. The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All of the files may be open at the same time. A logical record in the SAME RECORD AREA is considered as a logical record of each opened output file whose file-name appears in this SAME RECORD AREA clause and of the most recently read input file whose file-name appears in this SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the area i.e., records are aligned on the leftmost character position.

DATA DIVISION IN THE RELATIVE I-O MODULE

FILE SECTION

In a COBOL program the file description entry (FD) represents the highest level or organization in the File Section. The File Section header is followed by a file description entry consisting of a level indicator (FD), a file-name and a series of independent clauses. The FD clauses specify the size of the logical and physical records, the presence or absence of label records, the value of implementor-defined label items, and the names of the data records which comprise the file. The entry itself is terminated by a period.

RECORD DESCRIPTION STRUCTURE

A 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 if required, followed by a series of independent clauses as required. A record description has a hierarchical structure and therefore the clauses used with an entry may vary considerably, depending upon whether or not it is followed by subordinate entries. The structure of a record description is defined in CONCEPTS OF LEVELS in Section 2 while the elements allowed in a record description are shown in the DATA DESCRIPTION-COMPLETE ENTRY SKELETON in Section 3.

THE FILE DESCRIPTION - COMPLETE ENTRY SKELETON

Function

The file description furnishes information concerning the physical structure, identification, and record names pertaining to a given file.

General Format

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

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

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

   [ ; VALUE OF implementor-name-1 IS       { data-name-1 }
                                            { literal-1   }

            [ , implementor-name-2 IS   { data-name-2 } ] ] ...
                                        { literal -2  }

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

Syntax Rules

1. The level indicator FD identifies the beginning of a file description and must precede the file-name.

2. The clauses which follow the name of the file are optional in many cases, and their order of appearance is immaterial.

3. One or more record description entries must follow the file description entry.

THE BLOCK CONTAINS CLAUSE

Function

The BLOCK CONTAINS clause specifies the size of a physical record.

General Format

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

General Rule

This clause is required for documentation purposes only.

THE DATA RECORDS CLAUSE

Function

The DATA RECORDS clause serves only as documentation for the names of data records with their associated file.

General Format

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

Syntax Rule

Data-name-1 and data-name-2 are the names of data records and must have 01 level-number record descriptions, with the same names, associated with them.

General Rule

The DATA RECORDS clause is specified for documentation purposes only.

THE LABEL RECORDS CLAUSE

Function

The LABEL RECORDS clause specifies whether labels are present.

General Format

$$\underline{\text{LABEL}}\quad \left\{\begin{array}{l}\underline{\text{RECORD}}\ \text{IS}\\ \underline{\text{RECORDS}}\ \text{ARE}\end{array}\right\}\left\{\begin{array}{l}\underline{\text{STANDARD}}\\ \underline{\text{OMITTED}}\end{array}\right\}$$

Syntax Rule

This clause is optional in every file description entry.

General Rule

This clause is used for documentation purposes only.

THE RECORD CONTAINS CLAUSE

Function

The RECORD CONTAINS clause specifies the size of data records.

Format

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

General Rule

The size of each data record is completely defined within the record description entry, therefore, this clause is never required.

The RECORD CONTAINS clause is specified for documentation purposes only.

THE VALUE OF CLAUSE

Function

The VALUE of clause specializes the description of an item in the label records associated with a file.

General Format

$$\underline{\text{VALUE OF}}\ \text{data-name-1 IS}\ \begin{Bmatrix}\text{data-name-2}\\ \text{literal-1}\end{Bmatrix}$$
$$\left[\ \text{,data-name-3 IS}\ \begin{Bmatrix}\text{data-name-4}\\ \text{literal-2}\end{Bmatrix}\right]$$

Syntax Rules

1. Data-names should be qualified when necessary, but cannot be subscripted or indexed, nor can they be items described with the USAGE IS INDEX clause.

2. Data-name-2, data-name-4 etc., must be in the Working-Storage Section.

General Rules

1. This clause is for documentation purposes only.

   The compiler checks that data-name-1 matches in value data-name-2 or literal-1, data-name-3 matches in value data-name-4 or literal-2, etc., for input files. For output files, the value of data-name-2 or literal-1 is substituted for data-name-1, the value of data-name-4 or literal-2 is substituted for data-name-3, etc.

2. A figurative constant may be substituted in the format above wherever a literal is specified.

THE CLOSE STATEMENT

Function

The CLOSE statements terminates the processing of files. The LOCK is for documentary purposes only.

General Format

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

Syntax Rule

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

General Rules

1. A CLOSE statement may only be executed for a file in an open mode.

2. The action taken if a file is in the open mode when a STOP RUN statement is executed is to close the file. The action taken for a file that has been opened in a called program and not closed in that program prior to the execution of a CANCEL statement for the program is to close the file.

3. If a CLOSE statement has been executed for a file, no other statement can be executed that references that file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed.

4. Following the successful execution of a CLOSE statement, the record area associated with file-name is no longer available. The unsuccessful execution of such a CLOSE statement leaves the availability of the record area undefined.

THE DELETE STATEMENT

Function

The DELETE statement logically removes a record from a mass storage file.

General Format

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

Syntax Rules

1. The INVALID KEY phrase must not be specified for a DELETE statement which references a file which is in sequential access mode.

2. The INVALID KEY phrase must be specified for a DELETE statement which references a file which is not in sequential access mode and for which an applicable USE procedure is not specified.

General Rules

1. The associated file must be open in the I-O mode at the time of the execution of this statement. (See THE OPEN STATEMENT later in this Section).

2. For files in the sequential access mode, the last input-output statement must have been a successfully executed READ statement. The Operating System logically removes from the file the record that was accessed by that READ statement.

3. For a file in random or dynamic access mode, the Operating System logically removes from the file that record identified by the contents of the RELATIVE KEY data item associated with file-name. If the file does not contain the record specified by the key, an INVALID key condition exists. (See The INVALID KEY Condition in this Section.)

4. After the successful execution of a DELETE statement, the identified record has been logically removed from the file and can no longer be accessed.

5. The execution of a DELETE statement does not affect the contents of the record area associated with file-name.

6. The current record pointer is not affected by the execution of a DELETE statement.

7. The execution of the DELETE statement causes the value of the specified FILE STATUS data item, if any, associated with the file-name to be updated. (See I-O STATUS in this Section.)

THE OPEN STATEMENT

Function

The OPEN statement initiates the processing of files. It also performs checking and/or writing of labels and other input-output operations.

General Format

```
            ⎧ INPUT   file-name-1   [,file-name-2]... ⎫
OPEN        ⎨ OUTPUT  file-name-3   [,file-name-4]... ⎬ ...
            ⎩ I-O     file-name-5   [,file-name-6]... ⎭
```

Syntax Rule

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

General Rules

1. The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode.

2. The successful execution of the OPEN statement makes the associated record area available to the program.

3. Prior to the successful execution of an OPEN statement for a given file, no statement can be executed that references that file, either explicitly or implicitly.

4. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. In Table 6-1, 'X' at an intersection indicates that the specified statement, used in the access mode given for that row, may be used with the relative file organization and the open mode given at the top of the column.

Table 6-1. Permissible Combinations of Statements and Open Modes for Relative I/O.

File Access Mode	Statement	Open Mode		
		Input	Output	Input/Output
Sequential	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

5. A file may be opened with the INPUT, OUTPUT, AND I-O phrases in the same program. Following the initial execution of an OPEN statement for a file, each subsequent execution for that same file must be preceded by the execution of a CLOSE statement for that file.

6. Execution of the OPEN statement does not obtain or release the first data record.

7. The file description entry for file-name-1, file-name-2, file-name-5 or file-name-6 must be equivalent to that used when this file was created.

8. For files being opened with the INPUT or I-O phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set such that the next executed Format 1 READ statement for the file will result in an AT END condition. If the file does not exist, OPEN INPUT will cause an error status.

9. The I-O phrase permits the opening of a file for both input and output operations. If the file does not exist, it will be created.

10. Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At the time, the associated file contains no data records. If a file of the same number exists it will be deleted. If write protected, an error status occurs.

THE READ STATEMENT

Function

For sequential access, the READ statement makes available the next logical record from a file. For random access, the READ statement makes available a specified record from a disk file.

General Format

Format 1

```
READ    file-name [NEXT] RECORD [INTO identifier]
        [; AT END imperative-statement]
```

Format 2

```
READ file-name RECORD [INTO identifier] [;INVALID KEY imperative-statement]
```

Syntax Rules

1. The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions. The storage area associated with identifier and the record area associated with file-name must not be the same storage area.

2. Format 1 must be used for all files in sequential access mode.

3. The NEXT phrase must be specified for files in dynamic access mode, when records are to be retrieved sequentially.

4. Format 2 is used for files in random access mode or for files in dynamic access mode when records are to be retrieved randomly.

5. The INVALID KEY phrase or the AT END phrase must be specified if no applicable USE procedure is specified for file-name.

General Rules

1. The associated files must be open in the INPUT or I-O mode at the time this statement is executed. (See THE OPEN STATEMENT in this Section.)

2. The record to be made available by a Format 1 READ statement is determined as follows:

   a. The record, pointed to by the current record pointer, is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer; if the record is no longer accessible, which may have been caused by the deletion of the record, the current record pointer is updated to point to the next existing record in the file and that record is then made available.

   b. If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

3. The execution of the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. (See I-O Status in this Section.)

4. Regardless of the method used to overlap access time with processing time, the concept of the READ statement is unchanged in that a record is available to the object program prior to the execution of any statement following the READ statement.

5. When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the READ statement.

6. If the INTO phrase is specified, the record being read is moved from the record area to the area specified by identifier according to the rules specified for the MOVE statement without the CORRESPONDING phrase. The implied MOVE does not occur if the execution of the READ statement was unsuccessful. Any subscripting or indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

7. When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with identifier.

8. If, at the time of execution of a Format 1 READ statement, the position of current record pointer for that file is undefined, the execution of that READ statement is unsuccessful.

9. If, at the time of execution of a Format 1 READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful. (See I-O Status in this Section.)

10. When the AT END condition is recognized the following actions are taken in the specified order:

    a. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an AT END condition. (See I-O Status in this Section.)

    b. If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative-statement. Any USE procedure specified for this file is not executed.

    c. If the AT END phrase is not specified, then a USE procedure must be specified, either explicitly or implicitly, for this file, and that procedure is executed.

11. Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined.

12. When the AT END condition has been recognized, a Format 1 READ statement for that file must not be executed without first executing one of the following:

    a. A successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

    b. A successful START statement for that file.

    c. A successful Format 2 READ statement for that file.

13. For a file for which dynamic access mode is specified, a Format 1 READ statement with the NEXT phrase specified causes the next logical record to be retrieved from the file as described in General Rule 2.

14. If the RELATIVE KEY phrase is specified, the execution of a Format 1 READ statement updates the contents of the RELATIVE KEY data item such that it contains the relative record number of the record made available.

15. The execution of a Format 2 READ statement sets the current record pointer to, and makes available, the record whose relative record number is contained in the data item named in the RELATIVE KEY phrase for the file. If the file does not contain such a record, the INVALID KEY condition exists and execution of the READ statement is unsuccessful. (See The INVALID KEY Condition in this Section).

THE REWRITE STATEMENT

Function

The REWRITE statement logically replaces a record existing in a disk file.

General Format

REWRITE record-name [FROM identifier] [; INVALID KEY imperative-statement]

Syntax Rules

1. Record-name and identifier must not refer to the same storage area.

2. Record-name is the name of a logical record in the File Section of the Data Division.

3. The INVALID KEY phrase must be specified in the REWRITE statement for files in the random or dynamic access mode for which an appropriate USE procedure is not specified.

General Rules

1. The file associated with record-name must be open in the I-O mode at the time of execution of this statement. (See THE OPEN STATEMENT in this Section).

2. For files in the sequential access mode, the last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a successfully executed READ statement. The Operating System logically replaces the record that was accessed by the READ statement.

3. The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

4. The logical record released by a successful execution of the REWRITE statement is no longer available in the record area unless the associated file is named in a SAME RECORD AREA clause, in which case the logical record is avoidable to the program as a record of other files appearing in the same SAME RECORD AREA clause as the associated I-O file, as well as to the file associated with record-name.

5. The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

   MOVE identifier TO record-name

   followed by the execution of the same REWRITE statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of the REWRITE statement.

6. The current record pointer is not affected by the execution of a REWRITE statement.

7. The execution of the REWRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. (See I-O STATUS in this Section).

8. For a file accessed in either random or dynamic access mode, the Operating System logically replaces the record specified by the contents of the RELATIVE KEY data item associated with the file. If the file does not contain the record specified by the key, the INVALID KEY condition exists. (See THE INVALID KEY CONDITION in this Section). The updating operation does not take place and the data in the record area is unaffected.

THE START STATEMENT

Function

The START statement provides a basis for logical positioning within a relative file, for subsequent sequential retrieval of records.

General Format

```
                 ┌        ⎧ IS  EQUAL TO      ⎫                ┐
                 │        ⎪ IS =              ⎪                │
                 │        ⎪ IS  GREATER THAN  ⎪                │
START file-name  │ KEY    ⎨ IS >              ⎬   data-name    │
                 │        ⎪ IS  NOT LESS THAN ⎪                │
                 └        ⎩ IS  NOT <         ⎭                ┘

     [; INVALID KEY imperative-statement]
```

NOTE

The required relational characters '>', and '<' and '=' are not underlined to avoid confusion with other symbols such as '$\underline{>}$' (greater than or equal to).

Syntax Rules

1. File-name must be the name of a file with sequential or dynamic access.

2. Data-name may be qualified.

3. The INVALID KEY phrase must be specified if no applicable USE procedure is specified for file-name.

4. Data-name, if specified, must be the data item specified in the RELATIVE KEY phrase of the associated file control entry.

General Rules

1. File-name must be open in the INPUT or I-O mode at the time that the START statement is executed. (See THE OPEN STATEMENT in this Section).

2. If the KEY phrase is not specified, the relational operator 'IS EQUAL TO' is implied.

3. The type of comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referenced by file-name and a data item as specified in General Rule 5.

   a. The current record pointer is positioned to the first logical record currently existing in the file whose key satisfies the comparison.

   b. If the comparison is not satisfied by any record in the file, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined. (See The INVALID KEY Condition in this Section).

4. The execution of the START statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. (See I-O STATUS in this Section).

5. The comparison described in General Rule 3 uses the data item referenced by the RELATIVE KEY clause associated with file-name.

THE USE STATEMENT

Function

The USE statement specified procedures for input-output error handling that are in addition to the standard procedures provided by the input-output control system.

General Format

```
                          {             }   { file-name-1 [,file-name-2] } . . .
                          { EXCEPTION   }   { INPUT                      }
USE AFTER STANDARD        { PROCEDURE ON}   { OUTPUT                     }
                          { ERROR       }   { I-O                        }
```

Syntax Rules

1. A USE statement, when present, must immediately follow a section header in the declaratives section and must be followed by a period followed by a space. The remainder of the section must consist of zero, one or more procedural paragraphs that define the procedures to be used.

2. The USE statement itself is never executed; it merely defines the conditions calling for the execution of the USE procedures.

3. The same file-name can appear in a different specific arrangement of the format. Appearance of a file-name in a USE statement must not cause the simultaneous request for execution of more than one USE procedure.

4. The words ERROR and EXCEPTION are synonymous and may be used interchangably.

5. The files implicitly or explicitly referred in a USE statement need not all have the same organization or access.

General Rules

1. If the INVALID KEY or AT END phrases have not been specified in the input-output statement, the designated procedures are executed by the input-output system after completing the standard input-output error routine, upon recognition of the INVALID KEY or AT END conditions.

2. After execution of a USE procedure, control is returned to the invoking routine.

3. Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must be no reference to procedure-names in the declarative portion, except that PERFORM statements may refer to a USE statement or to the procedures associated with such a USE statement.

4. Within a USE procedure, there must not be the execution of any statement that would cause the execution of a USE procedure that had previously been invoked and had not yet returned control to the invoking routine.

THE WRITE STATEMENT

Function

The WRITE statement releases a logical record for an output or input-output file.

General Format

WRITE record-name [FROM identifier] [; INVALID KEY imperative-statement]

Syntax Rules

1. Record-name and identifier must not reference the same storage area.

2. The record-name is the name of a logical record in the File Section of the Data Division.

3. The INVALID KEY phrase must be specified if an applicable USE procedure is not specified for the associated file.

General Rules

1. The associated file must be open in the OUTPUT or I-O mode at the time of the execution of this statement. (See THE OPEN STATEMENT Section).

2. The logical record released by the execution of the WRITE statement is no longer available in the record area unless the associated file is named in a SAME RECORD AREA clause or the execution of the WRITE statement is unsuccessful due to an INVALID KEY condition.

   [The logical record is also available to the program as a record of other files referenced in the same SAME RECORD AREA clause as well as the file associated with record-name.]

3. The results of the execution of the WRITE statement with the FROM phrase is equivalent to the execution of

   a. The statement:

      MOVE identifier TO record-name

      according to the rules specified for the MOVE statement, followed by:

b. The same WRITE statement without the FROM phrase.

   The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement.

   After execution of the WRITE statement is complete, the information in the area referenced by identifier is available, even though the information in the area referenced by record-name may not be. (See General Rule 2 above).

4. The current record pointer is unaffected by the execution of a WRITE statement.

5. The execution of the WRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. (See I-O Status in this Section).

6. The maximum record size for a file is established at the time the file is created and must not subsequently be changed.

7. The number of character positions on a mass storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

8. The execution of the WRITE statement releases a logical record to the operating system.

9. When a file is opened in the output mode, records may be placed into the file by one of the following:

   a. If the access mode is sequential, the WRITE statement will cause a record to be released to the Operating System. The first record will have a relative record number of one and subsequent records released will have relative record numbers of 2, 3, 4, ... If the RELATIVE KEY data item has been specified in the file control entry for the associated file, the relative record number of the record just released will be placed into the RELATIVE KEY data item by the Operating System during execution of the WRITE statement.

   b. If the access mode is random or dynamic, prior to the execution of the WRITE statement, the value of the RELATIVE KEY data item must be initialized in the program with the relative record number to be associated with the record in the record area. That record is then released to the Operating System by execution of the WRITE statement.

10. When a file is opened in the I-O mode and the access mode is random or dynamic, records are to be inserted in the associated file. The value of the RELATIVE KEY data item must be initialized by the program with the relative record number to be associated with the record in the record area. Execution of a WRITE statement then causes the contents of the record area to be released to the Operating System.

11. The INVALID KEY condition exists under the following circumstances:

    a. When the access mode is random or dynamic, and the RELATIVE KEY data item specifies a record which already exists in the file, or

    b. When an attempt is made to write beyond the externally defined boundaries of the file.

12. When the INVALID KEY condition is recognized, the execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected, and the FILE STATUS data item, if any, of the associated file is set to a value indicating the cause of the condition. Execution of the program proceeds according to the rules stated in The INVALID KEY Condition in this Section. (See also I-O Status in this Section).

SECTION 7

INDEXED INPUT AND OUTPUT

INTRODUCTION TO THE INDEXED I-O MODULE

The Indexed I-O module provides a capability to access records of a mass storage file in either a random or sequential manner. Each record in an indexed file is uniquely identified by the value of one or more keys within that record.

LANGUAGE CONCEPTS

Organization

A file whose organization is indexed is a mass storage file in which date records may be accessed by the value of a key. A record description may include one or more key data items, each of which is associated with an index. Each index provides a logical path to the data records according to the contents of a data item within each record which is the record key for that index.

The data item named in the RECORD KEY clause of the file control entry for a file is the prime record key for that file. For purposes of inserting, updating and deleting records in a file, each record is identified solely by the value of its prime record key. This value must, therefore, be unique and must not be changed when updating the record. Key lengths must not exceed 64 bytes.

A data item named in the ALTERNATE RECORD KEY clause of the file control entry for a file is an alternative record key for that file. The value of an alternative record key may be non-unique if the DUPLICATES phrase is specified for it. These keys provide alternative access paths for retrieval of records from the file.

Access Modes

In the sequential access mode, the sequence in which records are accessed is the ascending order of the record key values. The order of retrieval of records within a set of records having duplicate record key values is the order in which the records were written into the set.

In the random access mode, the sequence in which records are accessed is controlled by the programmer. The desired record is accessed by placing the value of its record key in the record key data item.

In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements.

Current Record Pointer

The current record pointer is a conceptual entity used in this document to facilitate specification of the next record to be accessed within a given file. The concept of the current record pointer has no meaning for a file opened only in the output mode. The setting of the current record pointer is affected only by the OPEN, START and READ statements.

I-O Status

If the FILE STATUS clause is specified in a file control entry, a value is placed into the specified two-character data item during the execution of an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE or START statement and before any applicable USE procedure is executed, to indicate to the COBOL program the status of that input-output operation.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1 and is set to indicate one of the following conditions upon completion of the input-output operation.

'0' -	Successful Completion
'1' -	At End
'2' -	Invalid Key
'3' -	Permanent Error
'9' -	Run-Time Error Message

The meaning of the above indications are as follows:

'0' - Successful Completion. The input-output statement was successfully executed.

'1' - At End. The Format 1 READ statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

'2' - Invalid Key. The input-output statement was unsuccessfully executed as a result of one of the following:

Sequence Error
Duplicate Key
No Record Found
Boundary Violation

'3' - Permanent Error. The input-output statement was unsuccessful as the result of an input-output error, such as data check, parity error, or transmission error.

'9' - Run-Time Error Message. The input-output statement was unsuccessfully executed as the result of a condition that is specified by the Run-Time System Error Message number. This value is used only to indicate a condition not indicated by other defined values of status key 1, or by specified combinations of the value of status key 1 and status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2 and is used to further describe the results of the input-output operation. This character will contain a value as follows:

If no further information is available concerning the input-output operation, then status key 2 contains a value of '0'.

When status key 1 contains a value of '0' indicating a successful completion, status key 2 may contain a value of '2' indicating a duplicate key. This condition indicates one of two possibilities:

1. For a READ statement, the key value for the current key of reference is equal to the value of that same key in the next record within the current key of reference.

2. For a WRITE or REWRITE statement, the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.

3. When status key 1 contains a value of '2' indicating an INVALID KEY condition, status key 2 contains values to designate the cause of that condition as follows:

1	Indicates a sequence error for a sequentially accessed indexed file. The ascending sequence requirements of successive record key values have been violated (see The WRITE Statement later in this Section), or the prime record key value has been changed by the COBOL program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file.
2	Indicates a duplicate key value. An attempt has been made to write or rewrite a record that would create a duplicate key in an indexed file.

3 Indicates no record found. An attempt has been made to access a record, identified by a key, and that record does not exist in the file.

4 Indicates a boundary violation. An attempt has been made to write beyond the externally defined boundaries of an indexed file. This is usually treated as a fatal error by operating system.

When status key 1 contains a value of '9,' the value of status key 2 is the run-time system error message number. Appendix J contains some details of the status-key-2 representation. Note that it is not possible to extract this number directly.

Status key 2 is a hexadecimal number which is displayed in ASCII. This returned ASCII character must be converted back to its hexadecimal equivalent by the user.

This ASCII character and its hexadecimal equivalent are located in Table B-2 in Appendix B of the BTOS Reference Manual. Find this character in the table and then convert its corresponding character code (in hex) to decimal. The decimal number will be the COBOL Run-Time error.

Valid Combinations of Status Keys 1 and 2

The valid permissible combinations of the values of status key 1 and status key 2 are shown in the following table. An 'X' at an intersection indicates a valid permissible combination.

Status Key 1	Status Key 2				
	No Further Information (0)	Sequence Error (1)	Duplicate Key (2)	No Record Found (3)	Boundary Violation (4)
Successful Completion (0)	X				
At End (1)	X				
Invalid Key (2)		X	X	X	X
Permanent Error (3)	X				
Implementor Defined (9)	Run-Time System Error Message Number				

The INVALID KEY Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE or DELETE statement. For details of the causes of the condition, see The START Statement, The READ Statement, The WRITE Statement, and The DELETE Statement later in this section.

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

1. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an INVALID KEY condition. (See I-O Status).

2. If the INVALID KEY phrase is specified in the statement causing the condition, control is transferred to the INVALID KEY imperative statement. Any USE procedure specified for this file is not executed.

When the INVALID KEY condition occurs, execution of the input-output statement which recognized the condition is unsuccessful and the file is not affected.

The AT END Condition

The AT END condition can occur as a result of the execution of a READ statement. For details of the causes of the condition, see The READ Statement later in this Section.

ENVIRONMENT DIVISION IN THE INDEXED I-O MODULE

INPUT-OUTPUT SECTION

The File-Control Paragraph

Function

The FILE-CONTROL paragraph names each file and allows specification of other file-related information.

General Format

```
FILE-CONTROL.  [file-control-entry] ...
```

The File-Control Entry

Function

The file control entry names a file and may specify other file-related information.

General Format

```
SELECT file-name

     ASSIGN TO   {external-file-name-literal}
                 {file-identifier           }
              [, {external-file-name-literal}]
              [  {file-identifier           }]

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

   ; ORGANIZATION IS INDEXED

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

   ; RECORD KEY IS data-name-1

  [; ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLCIATES]] ...

   [; FILE STATUS IS data-name-3]
```

Syntax Rules

1. The SELECT clause must be specified first in the file control entry. The clauses which follow the SELECT clause may appear in any order.

2. Each file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph. Each file specified in the file control entry must have a file description entry in the Data Division.

3. If the ACCESS MODE clause is not specified, the ACCESS MODE IS SEQUENTIAL clause is implied.

4. Data-name-3 must be defined in the Data Division as a two-character data item of the category alphanumeric and must not be defined in the File Section.

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

6. The data item referenced by data-name-1 must be defined as a data item of the category alphanumeric within a record description entry associated with that file-name.

7. Data-name-1 cannot describe an item whose size is variable. (See The OCCURS Clause in Section 4).

8. Data-name-2 cannot reference an item whose leftmost character position corresponds to the leftmost character position of an item referenced by data-name-1 or by any other data-name-2 associated with this file.

General Rules

1. The ASSIGN clause specifies the association of the file referenced by file-name to a storage medium.

   (The first assignment takes effect. Subsequent assignments within any one ASSIGN clause are for documentation purposes only.)

2. The RESERVE clause allows the user to specify the number of input-output areas allocated. If the RESERVE clause is specified, the number of input-output areas allocated is equal to the value of integer-1.

3. The ORGANIZATION clauses specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed.

4. When the access mode is sequential, records in the file are accessed in the sequence dictated by the file organization. For indexed files this sequence is the order of ascending record key values within a given key of reference.

5. When the FILE STATUS clause is specified, a value will be moved by the operating system into the data item specified by data-name-3 after the execution of every statement that references that file either explicitly or implicitly. This value indicates the status of execution of the statement. (See I-O Status in this Section.)

6. If the access mode is random, the value of the record key data item indicates the record to be accessed.

7. When the access mode is dynamic, records in the file may be accessed sequentially and/or randomly. (See General Rules 4 and 6.)

8. The RECORD KEY clause specifies the record key that is the prime record key for the file. The values of the prime record key must be unique among records of the file. This prime record key provides an access path to records in an indexed file.

   COBOL only supports CHARACTER key types. If another key type is desired, the file must not be created with or by COBOL.

9. An ALTERNATE RECORD KEY clause specifies a record key that is an alternative record key for the file. This alternate record key provides an alternate access path to record in an indexed file.

10. The data description of data-name-1 and data-name-2 as well as relative locations within a record must be the same as that used when the file was created. The number of alternate keys for the file must also be the same as that used when the file was created.

11. The DUPLICATES phrase specifies that the value of the associated alternate record key may be duplicated within any of the records in the file. If the DUPLICATES phrase is not specified, the value of the associated alternate record key must not be duplicated among any of the records in the file.

12. When the file-name is ASSIGNed to a file-identifier, and that file-identifier is then declared in WORKING-STORAGE, B 20 COBOL expects the file-identifier to be followed by (to terminate with) a space.

    Example:
    ```
    01   your-file    PIC X(9)   VALUE  "IND.FILE ".
    ```

The I-O-CONTROL Paragraph

Function

The I-O-CONTROL paragraph specifies the points at which rerun is to be established and the memory area which is to be shared by different files.

General Format

```
 I-O-CONTROL
┌                                                                   ┐
│             {file-name-1       }         integer-1 RECORDS OF file-name-2 │
│ ; RERUN ON  {implementor-name  }  EVERY  integer-2 CLOCK-UNITS          │
└                                          condition-name                 ┘
┌                                                        ┐
│[;SAME [RECORD] AREA FOR file-name-3  ,[file-name-4]  ....│ ...
└                                                        ┘
```

Syntax Rules

1. The I-O-CONTROL paragraph is optional. The whole paragraph is for documentation purposes only when present.

2. File-name-1 must be a sequentially organized file.

3. When either the integer-1 RECORDS clause or the integer-2 CLOCK-UNITS clause is specified, implementor-name must be given in the RERUN clause.

4. When multiple integer-1 RECORDS clauses are specified, no two of them may specify the same file-name-2.

5. Only one RERUN clause containing the CLOCK-UNITS clause may be specified.

6. The two forms of the SAME clause (SAME AREA, SAME RECORD AREA) are considered separately in the following:

   More than one SAME clause may be included in a program, however:

   a. A file-name must not appear in more than one SAME AREA clause.

   b. A file-name must not appear in more than one SAME RECORD AREA clause.

   c. If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in the SAME AREA clause must appear in the SAME RECORD AREA clause. However, additional file-names not appearing in that SAME AREA clause may also appear in that SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any given time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any given time.

7. The files referenced in the SAME AREA or SAME RECORD AREA clauses need not all have the same organization or access.

General Rules

1. The RERUN clause is treated as for documentation purposes only.

2. The SAME AREA clause specifies that two or more files are to use the same memory area during processing. The area shared includes all storage areas assigned to the files specified; therefore, it is not valid to have more than one of the files open at the same time. (See Syntax Rule 6c.)

3. The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All of the files may be open at the same time. A logical record in the SAME RECORD AREA is considered as a logical record of each opened output file whose file-name appears in this SAME RECORD AREA clause and of the most recently read input file whose file-name appears in this SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the area, i.e., records are aligned on the leftmost character position.

FILE SECTION

In a COBOL program the file description entry (FD) represents the highest level or organization in the File Section. The File Section header is followed by a file description entry consisting of a level indicator (FD), a file-name and a series of independent clauses. The FD clauses specify the size of the logical and physical records, the presence or absence of label records, the value of implementor-defined label items, and the names of the data records which comprise the file. The entry itself is terminated by a period.

RECORD DESCRIPTION STRUCTURE

A 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 if required, followed by a series of independent clauses as required. A record description has a hierarchical structure and therefore the clauses used with an entry may vary considerably, depending upon whether or not it is followed by subordinate entries. The structure of a record description is defined in CONCEPTS OF LEVELS in Section 2 while the elements allowed in a record description are shown in the DATA DESCRIPTION-COMPLETE ENTRY SKELETON in Section 3.

THE FILE DESCRIPTION - COMPLETE ENTRY SKELETON

Function

The file description furnishes information concerning the physical structure, identification, and record names pertaining to a given file.

General Format

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

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

      ; LABEL  {RECORD IS   }{STANDARD}
               {RECORDS ARE }{OMITTED }
                                                 {data-name-1}
  [  ; VALUE OF implementor-name-1   IS          {literal-1  }

         [ [, implementor-name-2 IS  {data-name-2}]   ...  ]
                                     {literal -2 }
   [ ; DATA   {RECORD IS   }  data-name-3  [,data-name-4]    ... ] .
              {RECORDS ARE }
```

Syntax Rules

1. The level indicator FD identifies the beginning of a file description and must precede the file-name.

2. The clauses which follow the name of the file are optional in many cases, and their order of appearance is immaterial.

3. One or more record description entries must follow the file description entry.

THE BLOCK CONTAINS CLAUSE

Function

The BLOCK CONTAINS clause specifies the size of a physical record.

General format

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

General Rule

This clause is required for documentation purposes only.

THE DATA RECORDS CLAUSE

Function

The DATA RECORDS clause serves only as documentation for the names of data records with their associated file.

General Format

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

Syntax Rule

Data-name-1 and data-name-2 are the names of data records and must have 01 level-number record descriptions, with the same names, associated with them.

General Rules

1. 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.

2. Conceptually, all data records within a file share the same area. This is in no way altered by the presence of more than one type of data record within the file.

THE LABEL RECORDS CLAUSE

Function

The LABEL RECORDS clause specifies whether labels are present.

General Format

$$\underline{\text{LABEL}} \quad \left\{ \begin{array}{l} \underline{\text{RECORD}}\ \text{IS} \\ \underline{\text{RECORDS}}\ \text{ARE} \end{array} \right\} \quad \left\{ \begin{array}{l} \underline{\text{STANDARD}} \\ \underline{\text{OMITTED}} \end{array} \right\}$$

General Rule

This clause is used for documentation purposes only.

THE RECORD CONTAINS CLAUSE

Function

The RECORD CONTAINS clause specifies the size of data records.

General Format

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

General Rule

The size of each data record is completely defined within the record description entry, therefore, this clause is never required. The RECORD CONTAINS clause is specified for documentation purposes only.

THE VALUE OF CLAUSE

Function

The VALUE of clause specializes the description of an item in the label records associated with a file.

General Format

```
VALUE OF    data-name-1 IS    data-name-2
                              literal-1

           ,data-name-2 IS    data-name-4
                              literal-2       ...
```

Syntax Rules

1. Data-name-1, data-name-2, etc., should be qualified when necessary, but cannot be subscripted or indexed, nor can they be items described with the USAGE IS INDEX clause.

2. Data-name-1, data-name-2, etc., must be in the Working-Storage Section.

General Rules

1. For an input file, the appropriate label routine checks to see if the value of implementor-name-1 is equal to the value of literal-1, or of data-name-1, whichever has been specified.

   For an output file, at the appropriate time the value of implementor-name-1 is made equal to the value of literal-1, or of a data-name-1, whichever has been specified.

2. A figurative constant may be substituted in the format above wherever a literal is specified.

THE CLOSE STATEMENT

Function

The CLOSE statements terminates the processing of files. The LOCK phrase is for documentation purposes only.

General Format

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

Syntax Rule

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

General Rules

1. A CLOSE statement may only be executed for a file in an open mode.

2. The action taken if a file is in the open mode when a STOP RUN statement is executed is to close the file. The action taken for a file that has been opened in a called program and not closed in that program prior to the execution of a CANCEL statement for the program is to close the file.

3. If a CLOSE statement has been executed for a file, no other statement can be executed that references that file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed.

4. Following the successful execution of a CLOSE statement, the record area associated with file-name is no longer available. The unsuccessful execution of such a CLOSE statement leaves the availability of the record area undefined.

THE DELETE STATEMENT

Function

The DELETE statement logically removes a record from a mass storage file.

General Format

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

Syntax Rules

1. The INVALID KEY phrase must not be specified for a DELETE statement which references a file which is in sequential access mode.

2. The INVALID KEY phrase must be specified for a DELETE statement which references a file which is not in sequential access mode and for which an applicable USE procedure is not specified.

General Rules

1. The associated file must be open in the I-O mode at the time of the execution of this statement. (See The OPEN STATEMENT later in this Section).

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

3. For a file in random or dynamic access mode, the operating system logically removes from the file that record identified by the contents of the prime record key data item associated with file-name. If the file does not contain the record specified by the key, an INVALID KEY condition exists. (See The INVALID KEY CONDITION in this Section.)

4. After the successful execution of a DELETE statement, the identified record has been logically removed from the file and can no longer be accessed.

5. The execution of a DELETE statement does not affect the contents of the record area associated with file-name.

6. The current record pointer is not affected by the execution of a DELETE statement.

7. The execution of the DELETE statement causes the value of the specified FILE STATUS data item, if any, associated with file-name to be updated. (See I-O STATUS in this Section).

THE OPEN STATEMENT

Function

The OPEN statement initiates the processing of files. It also performs checking and/or writing of labels and other input-output operations.

General Format

$$\underline{\text{OPEN}} \quad \left\{ \begin{array}{lll} \underline{\text{INPUT}} & \text{file-name-1} & [,\text{file-name-2}]\ldots \\ \underline{\text{OUTPUT}} & \text{file-name-3} & [,\text{file-name-4}]\ldots \\ \underline{\text{I-O}} & \text{file-name-5} & [,\text{file-name-6}]\ldots \end{array} \right\} \ldots$$

Syntax Rule

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

General Rules

1. The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode.

2. The successful execution of the OPEN statement makes the associated record area available to the program.

3. Prior to the successful execution of an OPEN statement for a given file, no statement can be executed that references that file, either explicitly or implicitly.

4. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. In Table 2, Permissible Statements, 'X' at an intersection indicates that the specified statement, used in the access mode given for that row, may be used with the indexed file organization and the open mode given at the top of the column.

Table 7-1. Permissible Combinations of Statements and Open Modes for Indexed I/O.

File Access Mode	Statement	Open Mode		
		Input	Output	Input/Output
Sequential	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE		X	X
	START	X		X
	DELETE			X

5. A file may be opened with the INPUT, OUTPUT, and I-O phrases in the same program. Following the initial execution of an OPEN statement for a file, each subsequent execution for that same file must be preceded by the execution of a CLOSE statement for that file.

6. Execution of the OPEN statement does not obtain or release the first data record.

7. The assigned name in the select statement for a file is processed as follows:

   a. When the INPUT phrase is specified, the execution of the OPEN statement causes the assigned name to be checked in accordance with the operating system conventions for opening files for input.

   b. When the OUTPUT phrase is specified, the execution of the OPEN statement causes the assigned name to be written in accordance with the operating system conventions for opening files for output.

8. The file description entry for file-name-1, file-name-2, file-name-5 or file-name-6 must be equivalent to that used when this file was created.

9. For files being opened with the INPUT or I-O phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. For indexed files, the prime record key is established as the key of reference and is used to determine the first record to be accessed. If no records exist in the file, the current record pointer is set such that the next executed Format 1 READ statement for the file will result in an AT END condition. If the file does not exist, OPEN INPUT will cause an error status.

10. The I-O phrase permits the opening of a file for both input and output operations. If the file does not exist, it will be created.

11. Upon successful execution of an OPEN statement with the output phrase specified, a file is created. At that time the associated file contains no data records. If a file of the same number exists it will be deleted. If write protected, an error status occurs.

THE READ STATEMENT

Function

For sequential access, the READ statement makes available the next logical record from a file. For random access, the READ statement makes available a specified record from a mass storage file.

General Format

Format 1

```
READ file-name  [NEXT]  RECORD  [INTO identifier]

        [; AT END imperative-statement]
```

Format 2

```
READ file-name RECORD [INTO identifier]

        [;KEY IS data-name]

        [;INVALID KEY imperative-statement]
```

Syntax Rules

1. The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions. The storage area associated with identifier and the storage area which is the record area associated with file-name must not be the same storage area.

2. Data-name must be the name of a data item specified as a record key associated with file-name.

3. Data-name may be qualified.

4. Format 1 must be used for all files in sequential access mode.

5. The NEXT phrase must be specified for files in dynamic access mode, when records are to be retrieved sequentially.

6. Format 2 is used for files in random access mode or for files in dynamic access mode when records are to be retrieved randomly.

7. The INVALID KEY phrase or the AT END phrase must be specified if no applicable USE procedure is specified for file-name.

General Rules

1. The associated files must be open in the INPUT or I-O mode at the time this statement is executed. (See The Open Statement in this Section).

2. The record to be made available by a Format 1 READ statement is determined as follows:

   a. The record, pointed to by the current record pointer, is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer; if the record is no longer accessible, which may have been caused by the deletion of the record or a change in an ALTERNATE RECORD key. The current record pointer is updated to point to the next existing record within the established key of reference and that record is then made available.

   b. If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file within the established key of reference and then that record is made available.

3. The execution of the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. (See I-O STATUS in this Section.)

4. Regardless of the method used to overlap access time with processing time, the concept of the READ statement is unchanged in that a record is available to the object program prior to the execution of any statement following the READ statement.

5. When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the READ statement.

6. If the INTO phrase is specified, the record being read is moved from the record area to the area specified by identifier according to the rules specified for the MOVE statement. The implied MOVE does not occur if the execution of the READ statement was unsuccessful. Any subscripting or indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

7. When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with identifier.

8. If, at the time of execution of a Format 1 READ statement, the position of current record pointer for that file is undefined, the execution of that READ statement is unsuccessful.

9. If, at the time of execution of a Format 1 READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful. (See I-O STATUS in this Section.)

10. When the AT END condition is recognized the following actions are taken in the specified order:

    a. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an AT END condition. (See I-O STATUS in this Section.)

    b. If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative statement. Any USE procedure specified for this file is not executed.

    c. If the AT END phrase is not specified, then a USE procedure must be specified, either explicitly or implicitly, for this file, and that procedure is executed.

    When the AT END condition occurs, execution of the input-output statement which caused the condition is unsuccessful.

11. Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined. For indexed files the key of reference is also undefined.

12. When the AT END condition has been recognized, a Format 1 READ statement for that file must not be executed without first executing one of the following:

    a. A successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

    b. A successful START statement for that file.

    c. A successful Format 2 READ statement for that file.

13. For a file for which dynamic access mode is specified, a Format 1 READ statement with the NEXT phrase specified causes the next logical record to be retrieved from that file as described in General Rule 2 above.

14. For an indexed file being sequentially accessed, records having the same duplicate value in an alternate record key which is the key of reference are made available in the same order in which they are released by execution of WRITE statements, or by execution of REWRITE statements which create such duplicate values.

15. If the KEY phrase is not specified in a Format 2 READ statement, the prime record key is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of Format 1 READ statement for the file.

16. For an indexed file if the KEY phrase is specified in a Format 2 READ statement, data-name is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of Format 1 READ statements for the file until a different key of reference is established for the file.

17. Execution of a Format 2 READ statement causes the value of the key of reference to be compared with the value contained in the corresponding data item of the stored records in the file, until the first record having an equal value is found. The current record pointer is positioned to this record which is then made available. If no record can be so identified, the INVALID KEY condition exists and execution of the READ statement is unsuccessful. (See The INVALID KEY CONDITION in this Section.)

THE REWRITE STATEMENT

Function

The REWRITE statement logically replaces a record existing in a mass storage file.

General Format

<u>REWRITE</u> record-name [<u>FROM</u> identifier] [; <u>INVALID</u> KEY imperative-statement]

Syntax Rules

1. Record-name and identifier must not refer to the same storage area.

2. Record-name is the name of a logical record in the File Section of the Data Division.

3. The INVALID KEY phrase must be specified in the REWRITE statement for files for which an appropriate USE procedure is not specified.

General Rules

1. The file associated with record-name must be open in the I-O mode at the time of execution of this statement. (See The OPEN STATEMENT in this Section).

2. For files in the sequential access mode, the last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a successfully executed READ statement. The operating system logically replaces the record that was accessed by the READ statement.

3. The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

4. The logical record released by a successful execution of the REWRITE statement is no longer available in the record area unless the associated file is named in a SAME RECORD AREA clause, in which case the logical record is available to the program as a record of other files appearing in the same SAME RECORD AREA clause as the associated I-O file, as well as to the file associated with record-name.

5. The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

   MOVE identifier TO record-name

   followed by the execution of the same REWRITE statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of the REWRITE statement.

6. The current record pointer is not affected by the execution of a REWRITE statement.

7. The execution of the REWRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. (See I-O STATUS).

8. For a file accessed in the sequential access mode, the record to be replaced is specified by the value contained in the prime record key. When the REWRITE statement is executed, the value contained in the prime record key data item of the record to be replaced must be equal to the value of the prime record key of the last record read from this file.

9. For a file in the random or dynamic access mode, the record to be replaced is specified by the prime record key data item.

10. The contents of alternative record key data items of the record being rewritten may differ from those in the record being replaced. The operating system utilizes the content of the record key data items during the execution of the REWRITE statement in such a way that subsequent access of the record may be made based upon any of those specified record keys.

11. The INVALID KEY condition exists when:

    a. The access mode is sequential and the value contained in the prime record key data item of the record to be replaced is not equal to the value of the prime record key of the last record read from this file or,

    b. The value contained in the prime record key data item does not equal that of any record stored in the file, or

    c. The value contained in an alternate record key data item for which a DUPLICATES clause has not been specified is equal to that of a record already stored in the file.

    The updating operation does not take place and the data in the record area is unaffected. (See The INVALID KEY CONDITION in this Section).

THE START STATEMENT

Function

The START statement provides a basis for logical positioning within a indexed file, for subsequent sequential retrieval of records.

General Format

```
                    ┌       ┌ IS  EQUAL TO          ┐                    ┐
                    │       │ IS =                  │                    │
                    │       │ IS  GREATER THAN      │                    │
START file-name     │ KEY   │ IS  >                 │     data-name      │
                    │       │ IS  NOT LESS THAN     │                    │
                    └       └ IS  NOT <             ┘                    ┘
         [ ;INVALID KEY imperative-statement ]
```

NOTE: The required relational characters '>', and '<' and '=' are not underlined to avoid confusion with other symbols such as '≥' (greater than or equal to).

Syntax Rules

1. File-name must be the name of an indexed file.

2. File-name must be the name of a file with sequential or dynamic access.

3. Data-name may be qualified.

4. The INVALID KEY phrase must be specified if no applicable USE procedure is specified for file-name.

5. If file-name is the name of an indexed file, and if the KEY phrase is specified, data-name may reference a data item specified as a record key associated with file-name, or it may reference any data item of category alphanumeric subordinate to the data-name of a data item specified as a record key associated with file-name whose leftmost character position corresponds to the leftmost character position of that record key data item.

General Rules

1. File-name must be open in the INPUT or I-O mode at the time that the START statement is executed. (See The OPEN STATEMENT in this Section).

2. If the KEY phrase is not specified the relational operator 'IS EQUAL TO' is implied.

3. The type of comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referenced by file-name and a data item as specified in General Rule 5. If file-name references an indexed file and the operands are of unequal size, comparison proceeds as though the longer one were truncated on the right such that its length is equal to that of the shorter. All other nonnumeric comparison rules apply except that the presence of the PROGRAM COLLATING SEQUENCE clause will have no effect on the comparison. (See Comparison of Nonnumeric Operands.)

   a. The current record pointer is positioned to the first logical record currently existing in the file whose key satisfies the comparison.

   b. If the comparison is not satisfied by any record in the file, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined. (See The INVALID KEY CONDITION in this Section.)

4. The execution of the START statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. (See I-O STATUS).

5. If the KEY phrase is specified, the comparison described in General Rule 3 uses the data item reference by data-name.

6. If the KEY phrase is not specified, the comparison described in General Rule 3 uses the data item referenced in the RECORD KEY clause associated with file-name.

7. Upon completion of the successful execution of the START statement, a key of reference is established and used in subsequent Format 1 READ statements as follows: (See The READ STATEMENT in this Section).

   a. If the KEY phrase is not specified, the prime record key specified for file-name becomes the key of reference.

   b. If the KEY phrase is specified, and data-name is specified as a record key for file-name, that record key becomes the key of reference.

   c. If the KEY phrase is specified, and data-name is not specified as a record key for file-name, the record key whose leftmost character position corresponds to the leftmost character position of the data item specified by data-name becomes the key of reference.

8. If the execution of the START statement is not successful, the key of reference is undefined.

THE USE STATEMENT

Function

The USE statement specifies procedures for input-output error handling that are in addition to the standard procedures provided by the input-output control system.

General Format

$$\underline{\text{USE}}\ \underline{\text{AFTER}}\ \text{STANDARD}\left\{\begin{matrix}\underline{\text{EXCEPTION}}\\ \underline{\text{ERROR}}\end{matrix}\right\}\ \underline{\text{PROCEDURE}}\ \text{ON}\left\{\begin{matrix}\text{file-name-1 [,file-name-2]}\\ \underline{\text{INPUT}}\\ \underline{\text{OUTPUT}}\\ \underline{\text{I-O}}\end{matrix}\right\}\ldots$$

Syntax Rules

1. A USE statement, when present, must immediately follow a section header in the declaratives section and must be followed by a period followed by a space. The remainder of the section must consist of zero, one or more procedural paragraphs that define the procedures to be used.

2. The USE statement itself is never executed; it merely defines the conditions calling for the execution of the USE procedures.

3. The same file-name can appear in a different specific arrangement of the format. Appearance of a file-name in a USE statement must not cause the simultaneous request for execution of more than one USE procedure.

4. The words ERROR and EXCEPTION are synonymous and may be used interchangeably.

5. The files implicitly referenced in a USE statement need not all have the same organization or access.

General Rules

1. If the INVALID KEY or the AT END phrases have not been specified in the input-output statements, the designated procedures are executed by the input-output system after completing the standard input-output routine upon recognition of the INVALID KEY or AT END condition.

2. After execution of a USE procedure, control is returned to the invoking routine.

3. Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declarative portion, except that PERFORM statements may refer to a USE statement or to the procedures associated with such a USE statement.

4. Within a USE procedure, there must not be the execution of any statement that would cause the execution of a USE procedure that had previously been invoked and had not yet returned control to the invoking routine.

THE WRITE STATEMENT

Function

The WRITE statement releases a logical record for an output or input-output file.

General Format

WRITE record-name [FROM identifier] [;INVALID KEY imperative-statement]

1. Record-name and identifier must not reference the same storage area.

2. The record-name is the name of a logical record in the File Section of the Data Division.

3. The INVALID KEY phrase must be specified if an applicable USE procedure is not specified for the associated file.

General Rules

1. The associated file must be open in the OUTPUT or I-O mode at the time of the execution of this statement. (See The OPEN STATEMENT in this Section).

2. The logical record released by the execution of the WRITE statement is no longer available in the record area unless the associated file is named in a SAME RECORD AREA clause or the execution of the WRITE statement is unsuccessful due to an INVALID KEY condition. The logical record is available to the program from the file associated with record-name and from other files referenced in the same SAME RECORD AREA clause as the associated output file.

3. The results of the execution of the WRITE statement with the FROM phrase is equivalent to the execution of:

   a. The statement:

      MOVE identifier TO record-name

      according to the rules specified for the MOVE statement, followed by:

   b. The same WRITE statement without the FROM phrase.

      The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement.

After execution of the WRITE statement is complete, the information in the area referenced by identifier is available, even though the information in the area referenced by record-name may not be. (See General Rule 2 above).

4. The current record pointer is unaffected by the execution of a WRITE statement.

5. The execution of the WRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. (See I-O STATUS in this Section).

6. The maximum record size for a file is established at the time the file is created and must not subsequently be changed.

7. The number of character positions on a mass storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

8. The execution of the WRITE statement releases a logical record to the operating system.

9. Execution of the WRITE statement causes the contents of the record area to be released. The operating system utilizes the content of the record keys in such a way that subsequent access of the record may be made based upon any of those specified record keys.

10. The value of the prime record key must be unique within the records in the file.

11. The data item specified as the prime record key must be set by the program to the desired value prior to the execution of the WRITE statement.

12. If sequential access mode is specified for the file, records must be released to the operating system is ascending order of prime record key values.

13. If random or dynamic access mode is specified, records may be released to the operating system in any program-specified order.

14. When the ALTERNATE RECORD KEY clause is specified in the file control entry for an indexed file, the value of the alternative record key may be non-unique only if the DUPLICATES phrase is specified for that data item. In this case the operating system provides storage of records such that when records are accessed sequentially, the order of retrieval of those records is the order in which they are released to the operating system.

15. The INVALID KEY condition exists under the following circumstances:

    a. When sequential access mode is specified for a file opened in the output mode, and the value of the prime record key is not greater than the value of the prime record key of the previous record, or

    b. When the file is opened in the output or I-O mode, and the value of the prime record key is equal to the value of a prime record key of a record already existing in the file, or

    c. When the file is opened in the output or I-O mode, and the value of an alternate record key for which duplicates are not allowed equals the corresponding data item of a record already existing in the file, or

    d. When an attempt is made to write beyond the externally defined boundaries of the file.

16. When the INVALID KEY condition is recognized the execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected and the FILE STATUS data item, if any, associated with file-name of the associated file is set of a value indicating the cause of the condition. Execution of the program proceeds according to the rules stated under The INVALID KEY CONDITION (See also I-O STATUS in this Section).

SECTION 8

SORT-MERGE

INTRODUCTION TO THE SORT-MERGE MODULE

The Sort-Merge module provides the capability to order one or more files of records, or to combine two or more identically ordered files of records, according to a set of user-specified keys contained within each record. Optionally, a user may apply some special processing to each of the individual records by input or output procedures. This special processing may be applied before and/or after the records are ordered by the SORT or after the records have been combined by the MERGE.

RELATIONSHIP WITH SEQUENTIAL I-O MODULE

The files specified in the USING and GIVING phrases of the SORT and MERGE statements must be described implicitly or explicitly in the FILE-CONTROL paragraph as having sequential organization. No input-output statement may be executed for the file names in the sort-merge file description.

ENVIRONMENT DIVISION IN THE SORT-MERGE MODULE

INPUT-OUTPUT SECTION

The FILE-CONTROL Paragraph

Function

The FILE-CONTROL paragraph names each file and allows specification of other file-related information.

General Format

FILE-CONTROL. [file-control-entry] ...

The File-Control Entry

Function

The file-control entry names a sort or merge file and specifies the association of the file to a storage medium.

General Format

```
SELECT file-name

    ASSIGN TO implementor-name-1     [, implementor-name-2] ... .
```

Syntax Rules

1. Each sort or merge file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph. Each sort or merge file specified in the file control entry must have a sort-merge file description entry in the Data Division.

2. Since file-name represents a sort or merge file, only the ASSIGN clause is permitted to follow file-name in the FILE-CONTROL paragraph.

General Rule

The ASSIGN clause specifies the association of the sort or merge file referenced by file-name to a storage medium.

The I-O-CONTROL Paragraph

Function

The I-O-CONTROL paragraph specifies the memory area which is to be shared by different files.

General Format

```
I-O-CONTROL

[ ; SAME  [ RECORD      ]   AREA FOR file-name-1
          [ SORT        ]
          [ SORT-MERGE  ]
                 { , file-name-2 } ... ]    ...
```

Syntax Rules

1. The I-O-CONTROL paragraph is optional.

2. In the SAME AREA clause, SORT and SORT-MERGE are equivalent.

3. If the SAME SORT AREA or SAME SORT-MERGE AREA clause is used, at least one of the file-names must represent a sort or merge file. Files that do not represent sort or merge files may also be named in the clause.

4. The three formats of the SAME clause (SAME RECORD AREA, SAME SORT AREA, SAME SORT-MERGE AREA) are considered separately in the following:

More than one SAME clause may be included in a program, however:

a. A file-name must not appear in more than one SAME RECORD AREA clause.

b. A file-name that represents a sort or merge file must not appear in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.

c. If a file-name that does not represent a sort or merge file appears in a SAME AREA clause and one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, all of the files named in that SAME AREA clause must be named in that SAME SORT AREA or SAME SORT-MERGE AREA clause(s).

5. The files referenced in the SAME SORT AREA, SAME SORT-MERGE AREA, or SAME RECORD AREA clause need not all have the same organization or access.

General Rules

1. The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All of the files may be open at the same time. A logical record in the SAME RECORD AREA is considered as a logical record of each opened output file whose file-name appears in this SAME RECORD AREA clause and of the most recently read input file whose file-name appears in this SAME RECORD AREA clause. This is equivalent to implicit redefinition of the area, i.e., records are aligned on the leftmost character position.

2. If the SAME SORT AREA or SAME SORT-MERGE AREA clause is used, at least one of the file-names must represent a sort or merge file. Files that do not represent sort or merge files may also be named in the clause. This clause specifies that storage is shared as follows:

   a. The SAME SORT AREA or SAME SORT-MERGE AREA clause specifies a memory area which will be made available for use in sorting or merging each sort or merge file named. Thus any memory area allocated for the sorting or merging of a sort or merge file is available for reuse in sorting or merging any of the other sort or merge files.

   b. In addition, storage areas assigned to files that do not represent sort or merge files may be allocated as needed for sorting or merging the sort or merge files names in the SAME SORT AREA or SAME SORT-MERGE AREA clause. The extent of such allocation will be specified by the implementor.

   c. Files other than sort or merge files do not share the same storage area with each other. If the user wishes these files to share the same storage area with each other, he must also include in the program SAME AREA or SAME RECORD AREA clause naming these files.

   d. During the execution of a SORT or MERGE statement that refers to a sort or merge file named in this clause, any non sort-merge files named in this clause must not be open.

DATA DIVISION IN THE SORT-MERGE MODULE

FILE SECTION

An SD file description gives information about the size and the names of the data records associated with the file to be sorted or merged. There are no label procedures which the user can control, and the rules for blocking and internal storage are peculiar to the SORT and MERGE statements.

THE SORT-MERGE FILE DESCRIPTION - COMPLETE ENTRY SKELETON

Function

The sort-merge file description furnishes information concerning the physical structure, identification and record names of the file to be sorted or merged.

General Format

```
SD  file-name

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

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

Syntax Rules

1. The level indicator SD identifies the beginning of the sort-merge file description and must precede the file-name.

2. The clauses which follow the name of the file are optional and their order of appearance is immaterial.

3. One or more record description entries must follow the sort-merge file description entry, however, no input-output statements may be executed for this file.

THE DATA RECORDS CLAUSE

Function

The DATA RECORDS clauses serves only as documentation for the names of data records with their associated file.

General Format

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

Syntax Rule

Data-name-1 and data-name-2 are the names of data records and must have 01 level-number record descriptions, with the same names, associated with them.

General Rules

1. 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.

2. Conceptually, all data records within a file share the same area. This is in no way altered by the presence of more than one type of data record within the file.

THE RECORD CONTAINS CLAUSE

Function

The RECORD CONTAINS clause specifies the size of data records.

General Format

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

General Rule

The size of each data record is completely defined within the record description entry, therefore, this clause is never required. When present, however, the following notes apply:

a. Integer-2 may not be used by itelf unless all the data records in the file have the same size. In this case, integer-2 represents the exact number of characters in the data record. If integer-1 and integer-2 are both shown, they refer to the minimum number of characters in the smallest size data records and the maximum number of characters in the largest size data records, respectively.

b. The size is specified in terms of the number of character positions required to store the logical record, regardless of the types of characters used to represent the items within the logical record. The size of a record is determined by the sum of the number of characters in all fixed length elementary items plus the sum of the maximum number of characters in any variable length item subordinate to the record. This sum may be different from the actual size of the record; see SELECTION OF CHARACTER REPRESENTATION AND RADIX in Section 2; and The SYNCHRONIZED Clause and The USAGE Clause in Section 3.

THE MERGE STATEMENT

Function

The MERGE statement combines two or more identically sequenced files on a set of specified keys, and during the process makes records available, in merge order, to an output procedure or to an output file.

General Format

```
                              { ASCENDING  }
MERGE file-name-1 ON          { DESCENDING }  KEY    data-name-1 [, data-name-2] ...

  [ ON { ASCENDING  }  KEY data-name-3 [, data-name-4] ... ]  ...
  [    { DESCENDING }                                      ]
    [COLLATING SEQUENCE IS alphabet-name]

    USING file-name-2, file-name-3 [, file-name-4] ...
  { OUTPUT PROCEDURE IS section-name-1 [ { THROUGH }   section-name-2 ] }
  {                                    [ { THRU    }                  ] }
  {                                                                     }
  { GIVING file-name-5                                                  }
```

Syntax Rules

1. File-name-1 must be described in a sort-merge file description entry in the Data Division.

2. Section-name-1 represents the name of an output procedure.

3. File-name-2, file-name-3, file-name-4, and file-name-5 must be described in a file description entry, not in a sort-merge file description entry, in the Data Division. The actual size of the logical record(s) described for file-name-2, file-name-3, file-name-4, and file-name-5 must be equal to the actual size of the logical record(s) described for file-name-1. If the data descriptions of the elementary items that make up these records are not identical, it is the programmer's responsibility to describe the corresponding records in such a manner so as to cause an equal number of character positions to be allocated for the corresponding records.

4. The words THRU and THROUGH are equivalent.

5. Data-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names and are subject to the following rules:

   a. The data items identified by KEY data-names must be described in records associated with file-name-1.

   b. KEY data-names may be qualified.

   c. The data items identified by KEY data-names must not be variable length items.

   d. If file-name-1 has more than one record description, then the data items identified by KEY data-names need be described in only one of the record descriptions.

   e. None of the data items identified by KEY data-names can be described by an entry which either contains an OCCURS clause or is subordinate to an entry which contains an OCCURS clause.

6. No more than one file-name from a multiple file reel can appear in the MERGE statement.

7. File-names must not be repeated within the MERGE statement.

8. MERGE statements may appear anywhere except in the declaratives portion of the Procedure Division or in an input or output procedure associated with a SORT or MERGE statement.

General Rules

1. The MERGE statement will merge all records contained on file-name-2, file-name-3, and file-name-4. The files referenced in the MERGE statement must not be open at the time the MERGE statement is executed. These files are automatically opened and closed by the merge operation with all implicit functions performed, such as the execution of any associated USE procedures. The terminating function for all files is performed as if a CLOSE statement, without optional phrases, had been executed for each file.

2. The data-names following the word KEY are listed from left to right in the MERGE statement in order of decreasing significance without regard to how they are divided into KEY phrases. In the format, data-name-1 is the major key, data-name-2 is the next most significant key, etc.

   a. When the ASCENDING phrase is specified, the merged sequence will be from the lowest value of the contents of the data items identified by the KEY data-names to the highest value, according to the rules for comparison of operands in a relation condition.

   b. When the DESCENDING phrase is specified, the merged sequence will be from the highest value of the contents of the data items identified by the KEY data-names to the lowest value, according to the rule for comparison of operands in a relation condition.

3. The collating sequence that applies to the comparison of the nonnumeric key data items specified is determined in the following order of precedence:

   a. First, the collating sequence established by the COLLATING SEQUENCE phrase, if specified, in that MERGE statement.

   b. Second, the collating sequence established as the program collating sequence.

4. The output procedure must consist of one or more sections that appear contiguously in a source program and do not form part of any other procedure. In order to make merged records available for processing, the output procedure must include the execution of at least one RETURN statement. Control must not be passed to the output procedure except when a related SORT or MERGE statement is being executed. The output procedure may consist of any procedures needed to select, modify, or copy the records that are being returned one at a time in merge order, from file-name-1. The restrictions on the procedural statements within the output procedure are as follows:

   a. The output procedure must not contain any transfers of control to points outside the output procedure; ALTER, GO TO and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure. COBOL statements are allowed that will cause an implied transfer of control to declaratives.

   b. The output procedures must not contain any SORT or MERGE statements.

   c. The remainder of the Procedure Division must not contain any transfers of control to points inside the output procedures; ALTER, GO TO, and PERFORM statements in the remainder of the Procedure Division are not permitted to refer to procedure-names within the output procedures.

5. If an output procedure is specified, control passes to it during execution of the MERGE statement. The compiler inserts a return mechanism at the end of the last section in the output procedure. When control passes to the last statement in the output procedure, the return mechanism provides for termination of the merge, and then passes control to the next executable procedure. The merge procedure reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.

6. Segmentation, as defined in Section 9, can be applied to programs containing the MERGE statement. However, the following restrictions apply:

a. If the MERGE statement appears in a section that is not in an independent segment, then any output procedure referenced by that MERGE statement must appear:

   * Totally within non-independent segments, or

   * Wholly contained in a single independent segment

b. If a MERGE statement appears in an independent segment, then any output procedure referenced by that MERGE statement must be contained:

   * Totally within non-independent segments, or

   * Wholly within the same independent segment as that MERGE statement

7. If the GIVING phrase is specified, all the merged records in file-name-1 are automatically written on file-name-5 as the implied output procedure for this MERGE statement.

8. In the case of equal compare, according to the rules for comparison of operands in a relation condition, on the contents of the data items identified by all the KEY data-names between records from two or more input files (file-name-2, file-name-3, file-name-4, ...), the records are written on file-name-5 or returned to the output procedure, depending on the phrase specified, in the order that the associated input files are specified in the MERGE statement.

9. The results of the merge operation are predictable only when the records in the files referenced by file-name-2, file-name-3, ..., are ordered as described in the ASCENDING or DESCENDING KEY clause associated with the MERGE statement.

THE RELEASE STATEMENT

Function

The RELEASE statement transfers records to the initial phase of a SORT operation.

General Format

<u>RELEASE</u> record-name [<u>FROM</u> identifier]

Syntax Rules

1. A RELEASE statement may only be used within the range of an input procedure associated with a SORT statement for a file whose sort-merge file description entry contains record-name. (See The SORT Statement.)

2. Record-name must be the name of a logical record in the associated sort-merge file description entry and may be qualified.

3. Record-name and identifier must not refer to the same storage area.

General Rules

1. The execution of a RELEASE statement causes the record named by record-name to be released to the initial phase of a sort operation.

2. If the FROM phrase is used, the contents of the identifier data area are moved to record-name, then the contents of record-name are released to the sort file. Moving files takes place according to the rules specified for the MOVE statement without the CORRESPONDING phrase. The information in the record area is no longer available, but the information in the data area associated with identifier is available.

3. After the execution of the RELEASE statement, the logical record is no longer available in the record area unless the associated sort-merge file is named in a SAME RECORD AREA clause. The logical record is also available to the program as a record of other files referenced in the same SAME RECORD AREA clause as the associated sort-merge file, as well as to the file associated with record-name. When control passes from the input procedure, the file consists of all those records which were placed in it by the execution of RELEASE statements.

THE RETURN STATEMENT

Function

The RETURN statement obtains either sorted records from the final phase of a SORT operation or merged records during a MERGE operation.

General Format

```
RETURN file-name RECORD [INTO identifier]
                  ; AT END imperative-statement
```

Syntax Rules

1. File-name must be described by a sort-merge file description entry in the Data Division.

2. A RETURN statement may only be used within the range of an ouput procedure associated with a SORT or MERGE statement for file-name.

3. The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions. The storage area associated with identifier and the record area associated with file-name must not be the same storage area.

General Rules

1. When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the RETURN statement.

2. The execution of the RETURN statement causes the next record, in the order specified by the keys listed in the SORT or MERGE statement, to be made available for processing in the record areas associated with the sort or merge file.

3. If the INTO phrase 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 phrase. The implied MOVE does not occur if there is an AT END condition. Any subscripting or indexing associated with identifier is evaluated after the record has been returned and immediately before it is moved to the data item.

4. When the INTO phrase is used, the data is available in both the input record area and the data area associated with identifier.

5. If no next logical record exists for the file at the time of the execution of a RETURN statement, the AT END condition occurs. The contents of the record areas associated with the file when the AT END condition occurs are undefined. After the execution of the imperative-statement in the AT END phrase, no RETURN statement may be executed as part of the current output procedure.

THE SORT STATEMENT

Function

The SORT statement creates a sort file by executing input procedures or by transferring records from another file, sorts the records in the sort file on a set of specified keys, and in the final phase of the sort operation, makes available each record from the sort file, in sorted order to some output procedures or to an output file.

General Format

```
                          { ASCENDING  }
SORT file-name-1  ON      { DESCENDING }  KEY data-name-1 [, data-name-2] ...

                     [    { ASCENDING  }                                    ]
                     [ ON { DESCENDING }  KEY data-name-3 [, data-name-4] ... ] ...

[COLLATING SEQUENCE IS alphabet-name]

{ INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
{                                   [ { THRU    }                ] }
{                                                                  }
{ USING file-name-2 [, file-name-3] ...                            }

{ OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
{                                    [ { THRU    }                ] }
{                                                                   }
{ GIVING file-name-4                                                }
```

Syntax Rules

1. File-name-1 must be described in a sort-merge file description entry in the Data Division.

2. Section-name-1 represents the name of an input procedure. Section-name-3 represents the name of an output procedure.

3. File-name-2, file-name-3 and file-name-4 must be described in a file description entry, not in a sort-merge file description entry, in the Data Division. The actual size of the logical record(s) described for file-name-2, file-name-3 and file-name-4 must be equal to the actual size of the logical record(s) described for file-name-1. If the data size of the elementary items that make up these records are not identical, it is the programmer's responsibility to describe the corresponding records in such a manner so as to cause equal amounts of character positions to be allocated for the corresponding records.

4. Data-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names and are subject to the following rules:

   a. The data items identified by KEY data-names must be described in records associated with file-name-1.

   b. KEY data-names may be qualified.

   c. The data items identified by KEY data-names must not be variable length items.

   d. If file-name-1 has more than one record description, then the data items identified by KEY data-names need be described in only one of the record descriptions.

   e. None of the data items identified by KEY data-names can be described by an entry which either contains an OCCURS clause or its subordinate to an entry which contains an OCCURS clause.

5. The words THRU and THROUGH are equivalent.

6. SORT statements may appear anywhere except in the declaratives portion of the Procedure Division or in an input or output procedure associated with a SORT or MERGE statement.

7. No more than one file-name from a multiple file reel can appear in the SORT statement.

General Rules

1. The Procedure Division may contain more than one SORT statement appearing anywhere except:

   a. in the declaratives portion, or

   b. in the input and output procedures associated with a SORT or MERGE statement.

2. The data-names following the word KEY are listed from left to right in the SORT statement in order of decreasing significance without regard to how they are divided into KEY phrases. In the format, data-name-1 is the major key, data-name-2 is the next most significant key, etc.

   a. When the ASCENDING phrase is specified, the sorted sequence will be from the lowest value of the contents of the data items identified by the KEY data-names to the highest value, according to the rules for comparison of operands in a relation condition.

   b. When the DESCENDING phrase is specified, the sorted sequence will be from the highest value of the contents of the data items identified by the KEY data-names to the lowest value, according to the rules for comparison of operands in a relation condition.

3. The collating sequence that applies to the comparison of the nonnumeric key data items specified is determined in the following order of precedence:

   a. First, the collating sequence established by the COLLATING SEQUENCE phrase, if specified, in the SORT statement.

   b. Second, the collating sequence established as the program collating sequence.

4. The input procedure must consist of one or more sections that appear contiguously in a source program and do not form a part of any output procedure. In order to transfer records to the file referenced by file-name-1, the input procedure must include the execution of at least one RELEASE statement. Control must not be passed to the input procedure when a related SORT statement is being executed. The input procedure can include any procedures needed to select, create, or modify records. The restrictions on the procedural statements within the input procedure are as follows:

   a. The input procedure must not contain any SORT or MERGE statements.

   b. The input procedure must not contain any explicit transfers of control to points outside the input procedure; ALTER, GO TO, and PERFORM statements in the input procedure are not permitted to refer to procedure-names outide the input procedure. COBOL statements are allowed that will cause an implied transfer of control to declaratives.

   c. The remainder of the Procedure Division must not contain any transfers of control to points inside the input procedure; ALTER, GO TO and PERFORM statements in the remainder of the Procedure Division must not refer to procedure-names within the input procedure.

5. If an input procedure is specified, control is passed to the input procedure before file-name-1 is sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last section in the input procedure and when control passes the last statement in the input procedure, the records that have been released to file-name-1 are sorted.

6. The output procedure must consist of one or more sections that appear contiguously in a source program and do not form part of any input procedure. In order to make sorted records available for processing, the output procedure must include the execution of at least one RETURN statement. Control must not be passed to the output procedure except whan a related SORT statement is being executed. The output procedure may consist of any procedures needed to select, modify or copy the records that are being returned, one at a time in sorted order, from the sort file. The restrictions on the procedural statements within the output procedure are as follows:

   a. The output procedure must not contain any SORT or MERGE statements.

   b. The output procedure must not contain any explicit transfers of control to points outside the output procedure; ALTER, GO TO, and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure. COBOL statements are allowed that will cause an implied transfer of control to declaratives.

   c. The remainder of the Procedure Division must not contain any transfers of control to points inside the output procedure; ALTER, GO TO and PERFORM statements in the remainder of the Procedure Division are not permitted to refer to procedure-names within the output procedure.

7. If an output procedure is specified, control passes to it after file-name-1 has been sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last section in the output procedure and when control passes to the last statement in the output procedure, the return mechanism provides for termination of the sort and then passes control to the next executable statement after the SORT statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record in sorted order when requested. The RETURN statements in the output procedure are the requests for the next record.

8. Segmentation as defined in Section 9 can be applied to programs containing the SORT statement. However, the following restrictions apply:

   a. If a SORT statement appears in a section that is not in an independent segment, then any input procedures or output procedures referenced by that SORT statement must appear:

      * Totally within non-independent segments, or

      * Wholly contained in a single independent segment

   b. If a SORT statement appears in an independent segment, then any input procedures or output procedures referenced by that SORT statement must be contained:

      * Totally within non-independent segments, or

      * Wholly within the same independent segment as that SORT statement

9. If the USING phrase is specified, all the records in file-name-2 and file-name-3 are transferred automatically to file-name-1. At the time of execution of the SORT statement, file-name-2 and file-name-3 must not be open. The SORT statement automatically initiates the processing of, makes available the logical records for, and terminates the processing of file-name-2 and file-name-3. These implicit functions are performed such that any associated USE procedures are executed. The terminating function for all files is performed as if a CLOSE statement, without optional phrases, had been executed for each file. The SORT statement also automatically performs the implicit functions of moving the records from the file area of file-name 2 and file-name-3 to the file area for file-name-1 and the release of records to the initial phase of the sort operation.

10. If the GIVING phrase is specified, all the sorted records in file-name-1 are automatically written on file-name-4 as the implied output procedure for this SORT statement. At the time of execution of the SORT statement file-name-4 must not be open. The SORT statement automatically initiates the processing of, releases the logical records to, and terminates the processing of file-name-4. These implicit functions are performed such that any associated USE procedures are executed. The terminating function is performed as if a CLOSE statement, without optional phrases, had been executed for the file. The SORT statement also automatically performs the implicit functions of the return of the sorted records from the final phase of the sort operation and the moving of the records from the file area for file-name-1 to the file area for file-name-4.

SECTION 9

SEGMENTATION

INTRODUCTION TO THE SEGMENTATION MODULE

The Segmentation module provides a capability to specify object program overlay requirements.

Segmentation provides a facility for specifying permanent and independent segments. All sections with the same segment-number nust be contiguous in the source program. All segments specified as permanent segments must be contiguous in the source program.

GENERAL DESCRIPTION OF SEGMENTATION

COBOL segmentation is a facility that provides a means by which the user may communicate with the compiler to specify object program overlay requirements.

COBOL segmentation deals only with segmentation of procedures. As such, only the Procedure Division is considered in determining segmentation requirements for an object program.

ORGANIZATION

Program Segments

Although it is not mandatory, the Procedure Division for a source program is usually written as a consecutive group of sections, each of which is composed of a series of closely related operations that are designed to collectively perform a particular function. [However, when segmentation is used, the entire Procedure Division must be in sections.] In addition, each section must be classified as belonging either to the fixed portion or to one of the independent segments of the object program.

Fixed Portion

The fixed portion is defined as that part of the object program which is logically treated as if it were always in memory. This portion of the program is composed of fixed permanent segments.

A fixed permanent segment is a segment in the fixed portion which cannot be overlaid by any other part of the program.

Independent Segments

An independent segment is defined as part of the object program which can overlay, and can be overlaid by another independent segment. An independent segment is in its initial state whenever control is transferred (either implicitly or explicitly) to that segment for the first time during the execution of a program. On subsequent transfers of control to the segment, an independent segment is also in its initial state when:

1. Control is transferred to that segment as a result of the implicit transfer of control between consecutive statements from a segment with a different segment-number.

2. Control is transferred explicitly to that segment from a segment with a different segment-number (with the exception noted in paragraph 2 below.)

On subsequent transfer of control to the segment, an independent segment is in its last-used state when:

1. Control is tranferred implicitly to that segment from a segment with a different segment-number (except as noted in paragraph 1 above).

2. Control is transferred explicitly to that segment as the result of the execution of an EXIT PROGRAM statement.

SEGMENTATION CLASSIFICATION

Sections which are to be segmented are classified, using a system of segment-numbers and the following criteria:

1. Logic Requirements - Sections which must be available for reference at all times, or which are referred to very frequently, are normally classified as belonging to one of the permanent segments; sections which are used less frequently are normally classified as belonging to one of the independent segments, depending on logic requirements.

2. Frequency of Use - Generally, the more frequently a section is referred to, the lower its segment-number, the less frequently it is referred to, the higher its segment-number.

3. Relationship to Other Sections - Sections which frequently communicate with one another should be given the same segment-numbers.

SEGMENTATION CONTROL

The logical sequence of the program is the same as the physical sequence except for specific transfers of control. Control may be transferred within a source program to any paragraph in a section; that is, it is not mandatory to transfer control to the beginning of a section.

STRUCTURE OF PROGRAM SEGMENTS

SEGMENT-NUMBERS

Section classification is accomplished by means of a system of segment-numbers. The segment-number is included in the section header.

General Format

section-name SECTION [segment-number]

Syntax Rules

1. The segment-number must be an integer ranging in value from 0 through 99.
2. If the segment-number is omitted from the section header, the segment-number is assumed to be 0.
3. Sections in the declaratives must contain segment-numbers less than 50.

General Rules

1. All sections which have the same segment-number constitute a program segment. All sections which have the same segment-number must be together in the source program.
2. Segments with segment-number 0 through 49 belong to the fixed portion of the object program.
3. Segments with segment-number 50 through 99 are independent segments.

RESTRICTIONS ON PROGRAM FLOW

When segmentation is used, the following restrictions are placed on the ALTER and PERFORM statement.

THE ALTER STATEMENT

A GO TO statement in a section whose segment-number is greater than or equal to 50 must not be referred to by an ALTER statement in a section with a different segment-number.

THE PERFORM STATEMENT

A PERFORM statement that appears in a section that is not in an independent segment can have within its range, in addition, to any declarative sections whose execution is caused within that range, only one of the following:

* Sections and/or paragrahs wholly contained in one or more non-independent segments.

* Sections and/or paragraph wholly contained in a single independent segment.

A PERFORM statement that appears in an independent segment can have within its range, in addition to any declarative sections whose execution is caused within that range, only one of the following:

a. Sections and/or paragraphs wholly contained in one or more non-independent segments.

b. Sections and/or paragraphs wholly contained in the same independent segment as that PERFORM statement.

EXTRA INTERMEDIATE CODE FILES

When segmentation is used, extra intermediate code files are generated by the compiler as follows:

filename.Inn -	Intermediate code files one for each independent segment
filename.ISR -	Inter-Segment Reference table one per segmented program
filename.Dnn -	Dictionary files one for each independent segment except the last

where:

filename is the name without the extension of the principal intermediate code file

nn is a segment number that identifies the particular segment

NOTE

The filename.Dnn files are written and used solely by the compiler, and need not be retained after compilation. The filename.Inn files and the filename.ISR file must be retained as part of the object program and must also be copied when the program is copied.

SECTION 10

LIBRARY

INTRODUCTION TO THE LIBRARY MODULE

The Library module provides a capability for specifiying text that is to be copied from a source user-library file. This is usually created using any suitable source text editor.

The B 20 COBOL libraries consist of disk files that contain source to be made available to the compiler. The effect of the interpretation of the COPY statement is to insert text into the source program where it will be treated by the compiler as part of the source program.

THE COPY STATEMENT

FUNCTION

The COPY statement incorporates text into B 20 COBOL source program.

GENERAL FORMAT

COPY "text-name".

SYNTAX RULES

1. Text-name must be a unique standard operating system file name.
2. The COPY statement must be preceded by a space and terminated by the separator period.
3. A COPY statement may occur in the source program anywhere a character-string or a separator may occur except that a COPY statement must not occur within a COPY statement.

GENERAL RULES

1. The compilation of a source program containing COPY statement is logically equivalent to processing all COPY statements prior to the processing of the resulting source program.
2. The effect of processing a COPY statement is that the library text associated with text-name is copied into the source program, logically replacing the entire COPY statement, beginning with the reserved word COPY and ending with the punctuation character period, inclusive.
3. The library text is copied unchanged.
4. If the unit identifier is not explicitly specified, default is to the drive from which the compiler is loaded.

SECTION 11

DEBUG AND INTERACTIVE DEBUGGING

INTRODUCTION

Standard ANSI COBOL debugging provides a means by which the user can describe the conditions under which procedures are to be monitored during the execution of the object program.

The B 20 COBOL Run-Time Debug Package is an extension to ANSI COBOL that provides break-point facilities in the user's program. Programs may be run from the start until a specified break-point is reached, when control is passed back to the user. At this point, data areas may be inspected or changed.

B 20 COBOL RUN-TIME DEBUG EXTENSION

The Run-Time Debug Package is entered as an option by the user and the user program is then tested line by line, paragraph by paragraph, and so on, as required. The commands to the package can reference procedure statements and data areas by means of a 4-digit hexadecimal code output by the compiler against each line of the compilation listing. Powerful macros of commands can be used to give very sophisticated debugging facilities. The precise details for using the package are contained in Appendix J.

STANDARD ANSI COBOL DEBUG

The decisions of what to monitor and what information to display are explicitly in the domain of the user. The COBOL Debug facility simply provides a convenient access to pertinent information.

The features of the language that support the COBOL Debug module are:

* A compile time switch -- WITH DEBUGGING MODE

* An object time switch

* A USE FOR DEBUGGING statement

* A special register -- DEBUG-ITEM

* Debugging lines

The reserved word DEBUG-ITEM is the name for a special register generated automatically by the compiler that supports the debugging facility. Only one DEBUG-ITEM is allocated per program. The names of the subordinate data items in DEBUG-ITEM are also reserved words.

COMPILE-TIME SWITCH

The DEBUGGING MODE clause is written as part of the SOURCE-COMPUTER paragraph in the Environment Division. It serves as a compile-time switch over debugging statements written in the program.

When DEBUGGING MODE is not specified in a program, all the debugging lines are compiled as if they were comment lines and their syntax is not checked.

COBOL DEBUG OBJECT TIME SWITCH

An object time switch dynamically activates the debugging code inserted by the compiler. This switch cannot be addressed in the program; it is controlled outside the COBOL environment. If the switch is 'on', the effects of any USE FOR DEBUGGING statements written in the source program are permitted. If the switch is 'off', all the effects described in the USE FOR DEBUGGING Statement are inhibited. Recompilation of the source program is not required to provide or take away this facility.

The object time switch has no effect on the execution of the object program if the WITH DEBUGGING MODE clause was not specified in the source program at compile time.

The switch is described in Appendix J.

ENVIRONMENT DIVISION IN COBOL DEBUG

The WITH DEBUGGING MODE Clause

Function

The WITH DEBUGGING MODE clause indicates that all debugging sections and all debugging lines are to be compiled. If this clause is not specified, all debugging lines and sections are compiled as if they were comment lines.

General Format

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

General Rules

1. If the WITH DEBUGGING MODE clause is specified in the SOURCE-COMPUTER paragraph of the Configuration Section of a program, all USE FOR DEBUGGING statements and all debugging lines are compiled.

2. If the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph of the Configuration Section of a program, any USE FOR DEBUGGING statements and all associated debugging sections, and any debugging lines are compiled as if they were comment lines.

The USE FOR DEBUGGING Statement

Function

The USE FOR DEBUGGING statement identifies the user items that are to be monitored by the associated debugging section.

General Format

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

     USE FOR DEBUGGING ON { procedure-name-1 }
                          { ALL PROCEDURES   }

   [ procedure-name-2 ]
   [ ALL PROCEDURES   ]
```

Syntax Rules

1. Debugging sections, if specified, must appear together immediately after the DECLARATIVES header.
2. Except in the USE FOR DEBUGGING statement itself, there must be no reference to any non-declarative procedure within the debugging section.
3. Statemets appearing outside of the set of debugging sections must not reference procedure-names defined within the set of debugging sections.
4. Except for the USE FOR DEBUGGING statement itself, statements appearing within a given debugging section may reference procedure-names defined within a different USE procedure only with a PERFORM statement.
5. Procedure-names defined within debugging sections must not appear within USE FOR DEBUGGING statements.
6. Any given procedure-name may appear in only one USE FOR DEBUGGING statement and may appear only once in that statement.
7. The ALL PROCEDURES phrase can appear only once in a program.
8. When the ALL PROCEDURES phrase is specified, procedure-name-1, procedure-name-2, . . . must not be specified in any USE FOR DEBUGGING statement.
9. References to the special register DEBUG-ITEM are restricted to references from within a debugging section.

General Rules

1. In the following general rules all references to procedure-name-1, apply equally to procedure-name-2.

2. Automatic execution of a dubugging section is not caused by a statement appearing in a debugging section.

3. When procedure-name-1 is specified in a USE FOR DEBUGGING statement that debugging section is executed:

   a. Immediately before each execution of the named procedure;
   b. Immediately after the execution of an ALTER statement which references procedure-name-1.

4. The ALL PROCEDURES phrase causes the effects described in General Rule 3 to occur for every procedure-name in the program, except those appearing within a debugging section.

5. The associated debugging section is not executed for a specific operand more than once as a result of the execution of a single statement, regardless of the number of times that operand is explicitly specified. In the case of a PERFORM statement which caused iterative execution of a referenced procedure, the associated debugging section is executed once for each iteration.

   Within an imperative statement, each individual occurrence of an imperative verb identifies a separate statement for the purpose of debugging.

6. A reference to procedure-name-1 as a qualifier does not consititute reference to that item for the debugging described in the general rules above.

7. Associated with each execution of a debugging section is the special register DEBUG-ITEM, which provides information about the conditions that caused the execution of a debugging section. DEBUG-ITEM has the following implicit description:

```
01  DEBUG-ITEM.
    02  DEBUG-LINE       PICTURE IS X(6).
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-NAME       PICTURE IS X(30).
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-SUB-1      PICTURE IS S9999 SIGN IS LEADING SEPARATE
                         CHARACTER.
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-SUB-2      PICTURE IS S9999 SIGN IS LEADING SEPARATE
                         CHARACTER.
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-SUB-3      PICTURE IS S9999 SIGN IS LEADING SEPARATE
                         CHARACTER.
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-CONTENTS   PICTURE IS X(n).
```

8. Prior to execution of a debugging section, the contents of the data item referenced by DEBUG-ITEM are space-filled. The contents of data items subordinate to DEBUG-ITEM are then updated, according to the following general rules, immediately before control is passed to that debugging section. The contents of any data item not specified in the following general rules remain spaces.

   Updating is accomplished in accordance wit the rules for the MOVE statement, the sole exception being the move to DEBUG-CONTENTS when the move is treated exactly as if it was an alphanumeric to alphanumeric elementary move with no conversion of data from one form of internal representation to another.

9. The contents of DEBUG-LINE is the relevant COBOL source line number. This provides the means of identifying a particular source statement.

10. DEBUG-NAME contains the first 30 characters of the name that caused the debugging section to be executed.

    Subscripts/indices, if any, are not entered into DEBUG-NAME.

11. DEBUG-CONTENTS is a data item that is large enough to contain the data required by the following general rules.

12. If the first execution of the first nondeclarative procedure in the program causes the debugging section to be executed, the following conditions exist:

    a. DEBUG-LINE Identifies the first statement of that procedure.
    b. DEBUG-NAME contains the name of that procedure.
    c. DEBUG-CONTENTS contains 'START PROGRAM.'

13. If a reference to procedure-name-1 in an ALTER statement causes the debugging section to be executed, the following conditions exist:

    a. DEBUG-LINE identifies the ALTER statement that references procedure-name-1.
    b. DEBUG-NAME contains procedure-name-1.
    c. DEBUG-CONTENTS contains the applicable procedure-name associated with the TO Phrase of the ALTER statement.

14. If the transfer of control associated with the execution of a GO TO statement causes the debugging section to be executed, the following conditions exist:

    a. DEBUG-LINE identifies the GO TO statement whose execution transfers control to procedure-name-1.
    b. DEBUG-NAME contains procedure-name-1.

15. If the transfer of control from the control mechanism associated with a PERFORM statement caused the debugging section associated with procedure-name-1 to be executed, the following conditions exist:

    a. DEBUG-LINE Identifies the PERFORM statement that references procedure-name-1.
    b. DEBUG-NAME contains procedure-name-1.
    c. DEBUG-CONTENTS contains 'PERFORM LOOP'.

16. If procedure-name-1 is a USE procedure that is to be executed, the following conditions exist:

    a. DEBUG-LINE identifies the statement that causes execution of the USE procedure.
    b. DEBUG-NAME contains procedure-name-1.
    c. DEBUG-CONTENTS contains 'USE PROCEDURE'.

17. If an implicit transfer of control from the previous sequential paragraph to procedure-name-1 causes the debugging section to be executed, the following conditions exist:

    a. DEBUG-LINE identifies the previous statement.
    b. DEBUG-NAME contains procedure-name-1.
    c. DEBUG-CONTENTS contains 'FALL THROUGH'.

DEBUGGING LINES

A debugging line is any line with a 'D' in the indicator area of the line. Any debugging line that consists solely spaces from margin A to margin R is considered the same as a blank line.

The contents of a debugging line must be such that a syntactically correct program is formed with or without the debugging lines being considered as comment lines.

A debugging line will be considered to have all the characteristics of a comment line, if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph.

Successive debugging lines are allowed. Continuation of debugging lines is permitted, except that each continuation line must contain a 'D' in the indicator area, and character-strings may not be broken across two lines.

A debugging line is only permitted in the program after the OBJECT-COMPUTER paragraph.

SECTION 12

INTERPROGRAM COMMUNICATION

INTRODUCTION TO THE INTER-PROGRAM COMMUNICATION MODULE

The Inter-Program Communication module provides a facility by which a program can communicate with one or more programs. This provides a programmer with a modular programming capability. Each module when CALLed is loaded dynamically by the Run Time System. Communication is provided by:

* The ability to transfer control from one program to another within a run unit
* The ability for both programs to have access to the same data items.

DATA DIVISION IN THE INTER-PROGRAM COMMUNICATION MODULE

LINKAGE SECTION

The Linkage Section in a program is meaningful if and only if the object program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

The Linkage Section is used for describing data that is available through the calling program but is to be referred to in both the calling and the called program. No space is allocated in the program for data items referenced by data-names in the linkage Section of that program. Procedure Division references to these data items are resolved at object time by equating the reference in the called program to the location used in the calling program. In the case of index-names, no such correspondence is established. Index-names in the called and calling program always refer to separate indices.

Data items defined in the Linkage Section of the called program may be referenced within the Procedure Division of the called program only if they are specified as operands of the USING phrase of the Procedure Division header or are subordinate to such operands, and the object program is under the control of a CALL statement that specifies a USING phrase.

The structure of the Linkage Section is the same as that previously described for the Working-Storage Section, beginning with a section header, followed by data description entries for noncontiguous data items and/or record description entries.

Each Linkage Section record-name and noncontiguous item name must be unique within the called program since it cannot be qualified. Data items defined in the Linkage Section of the called program must not be associated with data items defined in the Report Section of the calling program.

Of those items defined in the Linkage Section only data-name-1, data-name-2, . . . in the USING phrase of the Procedure Division header, data items subordinate to these data-names, and condition-names and/or index-names associated with such data-names and/or subordinate data items, may be referenced in the Procedure Division.

Noncontiguous Linkage Storage

Items in the Linkage Section that bear no hierarchic relationship to one another need not be grouped into records and are classified and defined as noncontiguous elementary items. Each of these data items is defined in a separate data description entry which begins with the special level-number 77.

The following data clauses are required in each data description entry:

* Level-number 77
* Data-name
* The PICTURE clause or the USAGE IS INDEX clause

Other data description clauses are optional and can be used to complete the description of the item, if necessary.

THE PROCEDURE DIVISION HEADER

The Procedure Division is identified by and must begin with the following header:

<u>PROCEDURE</u> <u>DIVISION</u> [<u>USING</u> data-name-1 [, data-name-2] ...]

The USING phrase is present if and only if the object program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

Each of the operands in the USING phrase of the Procedure Division header must be defined as a data item in the Linkage Section of the program in which this header occurs, and it must have a 01 or 77 level-number.

Within a called program, Linkage Section data items are processed according to their data descriptions given in the called program.

When the USING Phrase is present, the object program operates as if data-name-1 of the Procedure Division header in the called program and data-name-1 in the USING phrase of the CALL statement in the calling program refer to a single set of data that is equally available to both the called and calling programs. Their descriptions must define an equal number of character positions; however they need not be the same name. In like manner, there is an equivalent relationship between data-name-2, . . . , in the USING phrase of the called program and data-name-2, . . . , in the USING phrase of the CALL statement in the calling program. A data-name must not appear more than once in the USING phrase in the Procedure Division header of the called program; however, a given data-name may appear more than once in the same USING phrase of a CALL statement.

THE CALL STATEMENT

Function

The CALL statement causes control to be transferred from one object program to another within the run unit.

General Format

Format 1

```
CALL   { identifier-1 }   [USING data-name-1       [, data-name-2] ...]
       { literal-1    }
       [ON OVERFLOW imperative-statement]
```

Format 2

```
CALL   { literal-2    }   [USING data-name-3       [, data-name-4] ...]
       { identifier-2 }
```

Syntax Rules

1. Literal-1 must be a nonnumeric literal.
2. Identifier-1 must be defined as a category alphanumeric, usage is display data item.
3. The USING phrase is included in the CALL statement only if there is a USING phrase in the Procedure Division header of the called program and the number of operands in each USING phrase must be identical.
4. Each of the operands in the USING phrase must have been defined as a data item in the File Section, Working-Storage Section, or Linkage Section, and must have a level-number of 01 or 77.
5. Literal-2 must be defined as a nonnumeric literal.
6. Identifier-2 must be defined as an alphanumeric data item. (See Appendix J for details.)

General Rules

1. The program whose name is specified by the value of literal-1 or identifer-1 is a called intermediate code module, literal-2 is a called run time subroutine; the program in which the CALL statement appears is the calling program.
2. The execution of a CALL statement causes control to pass to the called program.

3. In format 1, a called intermediate code module is loaded from disk the first time it is called within a run-unit and the first time it is called after a CANCEL to the called program.

   On all other entries into the called program, the state of the program remains unchanged from its state when last executed. This includes all data fields, the status and positioning of all files, and all alterable switch settings.

4. In format 2, a called run time subroutine is always in the state in which it last existed.

5. If during the execution of a CALL statement, it is determined that the available portion of run-time memory is incapable of accommodating the program specified in the CALL statement, the next sequential instruction is executed. If ON OVERFLOW has been specified, the associated imperative statement is executed before the next instruction is executed.

6. Called programs may contain CALL statements. However, a called program must not contain a call statement that directly or indirectly calls the calling program.

7. The data-names, specified by the USING phrase of the CALL statement, indicate those data items available to a calling program that may be referred to in the called program. The order of appearance of the data-names in the USING phrase of the CALL statement and the USING phrase in the Procedure Division header is critical. Corresponding data-names refer to a single set of data which is available to the called and calling program. The correspondence is positional, not by name. In the case of index-names, no such correspondence is established. Index-names in the called and calling program always refer to separate indices.

8. The CALL statement may appear anywhere within a segmented program. Therefore, when a CALL statement appears in a section with a segment-number greater than or equal to 50, that segment is in its last used state when the EXIT PROGRAM statement returns to the calling program.

THE CANCEL STATEMENT

Function

The CANCEL statement releases the memory areas occupied by the referred to program.

General Format

$$\underline{\text{CANCEL}} \quad \left\{ \begin{matrix} \text{identifier-1} \\ \text{literal-1} \end{matrix} \right\} \quad \left[\left\{ \begin{matrix} \text{, identifier-2} \\ \text{literal-2} \end{matrix} \right\} \right]$$

Syntax Rules

1. Literal-1, literal-2, . . . , must each be a nonnumeric literal.
2. Identifier-1, identifier-2, . . . , must each be defined as an alphanumeric data item such that its value can be a program name.

General Rules

1. Subsequent to the execution of a CANCEL statement, the program referred to therein ceases to have any logical relationship to the run unit in which the CANCEL statement appears. A subsequently executed CALL statement naming the same program will result in that program being initiated in its initial state. The memory areas associated with the named programs are released so as to be made available for disposition by the operating system.
2. A program named in the CANCEL statement must not refer to any program that has been called and has not yet executed an EXIT PROGRAM statement.
3. A logical relationship to a cancelled subprogram is established only by execution of a subsequent call statement.
4. A called program is cancelled either by being referred to as the operand of a CANCEL statement or by the termination of the run unit of which the program is a member.
5. No action is taken when a CANCEL statement is executed naming a program that has not been called in this run unit or has been called and is at present cancelled. Control passes to the next statement.

THE EXIT PROGRAM STATEMENT

Function

The EXIT PROGRAM statement marks the logical end of a called program.

General Format

EXIT PROGRAM.

Syntax Rules

1. The EXIT PROGRAM statement must appear in a sentence by itself.

2. The EXIT PROGRAM sentence must be the only sentence in the paragraph.

General Rule

An execution of an EXIT PROGRAM statement in a called program causes control to be passed to the calling program. Execution of an EXIT PROGRAM statement in a program which is not called behaves as if the statement were an EXIT statement. (See The EXIT STATEMENT in Section 3.)

SECTION 13

PROGRAMMING TECHNIQUES, USEFUL HINTS AND PROGRAM SIZING

PROGRAMMING TECHNIQUES

Although COBOL is written in an essentially free form, the user will nevertheless reap many advantages from a few self-imposed disciplines. It is suggested that these should include the following:

1. Use of the first 256 bytes of working-storage for variables which are frequently referenced will produce more compact and efficient code.
2. Use subscripts as sparingly as possible because each subscript has a storage requirement approximately equal to the size of a normal instruction.
3. For ACCEPT and DISPLAY, the compiler generates one instruction per elementary item of the data-name being displayed/accepted. Therefore, redefine a group of fields as a single field for DISPLAY whenever possible and avoid unnecessary numbers of small fields in ACCEPT.
4. Use FILLER instead of a data-name for any elementary field not referenced explicitly because the word FILLER is compacted to one character in the Data Dictionary.
5. Keep the number of digits in numeric fields as small as possible.
6. Whenever possible move a group instead of several elementary moves.

USEFUL HINTS

When writing interactive programs the following facilities of L/II COBOL should be remembered:

1. By use of the CURSOR IS facility and the ACCEPT statement it is easy to program continually depending on the cursor position after a menu type of prompt. The operator need then only move the cursor to the option required to reply to the prompt, or just press RETURN in the default case.
2. Remember always to end your B 20 COBOL program with a full stop (period). Invalid intermediate code can result if this final full stop is missing.
3. If the STOP "literal" statement is used in a program, execution of the program halts at this statement with the literal displayed on the CRT screen. Execution continues on pressing the Carriage Return (CR) key. It should be noted that if any string of characters terminated by a CR character has been keyed and is waiting to be read, the program will appear not to halt. Examples of this are if the CR key was inadvertently pressed twice at the last command entered or if parameters to the CRUN command have not yet been read.

SIZING

GENERAL DESCRIPTION

There are three aspects to sizing a program; the source code, the Data Dictionary and the compiled code.

The maximum number of source statements per program is limited, firstly by the space available to load the generated program.

The Data Dictionary contains an entry for every user-defined name in the program. Detailed information is containeq in the next section.

A guide for calculating the size of the generated program is as follows:

The sum of the Record size for each file in bytes
- + the Record size for each Working-Storage record in bytes
- + the number of characters in all Procedure Division literals
- + 60 bytes per File
- + 300 bytes control area
- + 6 bytes per COBOL instruction with the following qualifiers

for an ACCEPT/DISPLAY statement, add 3 bytes per elementary item within the Accepted/Displayed data-name.

for every subscript used in a statement, add 7 bytes

for a comparison, add 6 bytes

for an implicitly generated comparison, e.g., PERFORM UNTIL, READ AT END - add 6 bytes

DATA DICTIONARY

The Data Dictionary is constructed as the program is compiled. Each user-defined name will have an entry in this dictionary. The number of bytes required for each entry is given in Table 14-1.

Table 14-1. Data Dictionary Entry Sizing.

User-defined name	Number of Bytes [1]	
File-name	18 + n	
Record-name	8 + n	
Key-name	8 + n	
Status-name	8 + n	
Paragraph-name	6 + n	2
Data-name Grop	8 + n	2
Alphanumeric $<$ 32 characters	7 + n	2
Alphanumeric $>$ 32 characters	8 + n	2
Numeric integer	7 + n	2
Numeric non integer	8 + n	
Numeric edited	7 + n + x	

1 - n = number of characters in user-defined name.

For a FILLER, n = 1.

x = number of characters in PICture, after coalescing repetitions.

e.g.	9999.9	= 3 bytes
	9(4).9	= 3 bytes
	Z(2)9(4).9(3)	= 4 bytes

2 - Subtract 1 byte if item is in the first 256 bytes of Working-Storage.

Add 4 bytes if item has an OCCURS clause associated with it.

Add 2 bytes if item is subordinate to an item described with OCCURS.

APPENDIX A

RESERVED WORD LIST

RESERVED WORD LIST

This appendix contains a full list of COBOL and L/II COBOL reserved words. A shaded reserved word is a L/II COBOL extension to ANSI COBOL.

This / symbol denotes that the text up to that point is a reserved word, as is the whole word.

e.g., In INDEX/ED, INDEX and INDEXED are reserved words. In SPACE/S, SPACE and SPACES are reserved words.

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
ARE
AREA/S
ASCENDING
ASSIGN
AT
AUTHOR

BEFORE
BLANK
BLOCK
BY

CALL
CANCEL
CD
CHARACTER/S
CLOCK-UNITS
CLOSE
COBOL
CODE/-SET
COLLATING
COMMA
COMMUNICATION
COMP-M
COMP-N
COMP-3
COMP/UTATIONAL/-3
COMPUTE
CONFIGURATION
CONSOLE
CONTAINS
COPY
CORR/ESPONDING
COUNT
CRT
CRT-UNDER
CURRENCY
CURSOR

DATA
DATE-WRITTEN
DATE/-COMPILED
DAY
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

ELSE
ENABLE
END
ENTER
ENVIRONMENT
EQUAL
ERROR
EVERY
EXCEPTION
EXIT
EXTEND

FD
FILE
FILE-CONTROL
FILLER
FIRST
FOR
FROM

GIVING
GO
GREATER

HIGH-VALUE/S

I-O/-CONTROL
IDENTIFICATION
IF
IN
INDEX/ED
INITIAL
INPUT/-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST/IFIED

KEY

LABEL
LEADING
LEFT
LESS
LIMIT/S
LINAGE/-COUNTER
LINE/S
LINKAGE
LOCK
LOW-VALUE/S

MEMORY
MERGE
MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEGATIVE
NEXT
NOT
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL

OR
ORGANIZATION
OUTPUT
OVERFLOW

PAGE
PERFORM
PIC/TURE
POINTER
POSITIVE
PROCEED
PROCEDURE/S
PROGRAM/-ID

QUEUE
QUOTE/S

RANDOM
RD
READ
RECEIVE
RECORD/S
REDEFINES
REEL
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
RERUN
REWRITE
RIGHT
ROUNDED
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT/-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT

SORT-MERGE
SOURCE/-COMPUTER
SPACES
SPECIAL-NAMES
STANDARD/-1
START
STATUS
STOP
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SWITCH
SYMBOLIC
SYNC/HRONIZED

TAB
TABLE
TALLYING
TAPE
TERMINAL
THAN
THEN
THROUGH
THRU
TIME/S
TO
TOP
TRAILING
TYPE

UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING

VALUE/S
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO/ES or S

. (period)
(
-
*
**
)
;
+
/
,

=

APPENDIX B

CHARACTER SETS AND COLLATING SEQUENCE

ASCII	HEX	COBOL	ASCII	HEX	COBOL	ASCII	HEX	COBOL
NUL	00	X	/	2F			5E	X
SOH	01	X	0	30			5F	X
STX	02	X	1	31			60	X
ETX	03	X	2	32		a	61	
EOT	04	X	3	33		b	62	
ENG	05	X	4	34		c	63	
ACK	06	X	5	35		d	64	
BEL	07	X	6	36		e	65	
BS	08	X	7	37		f	66	
HT	09	X	8	38		g	67	
LF	0A	X	9	39		h	68	
VT	0B	X	:	3A	X	i	69	
FF	0C	X	;	3B		j	6A	
CR	0D	X	<	3C		k	6B	
SO	0E	X	=	3D		l	6C	
SI	0F	X	>	3E		m	6D	
DLE	10	X	?	3F	X	n	6E	
DCI	11	X	@	40	X	o	6F	
DC2	12	X	A	41		p	70	
DC3	13	X	B	42		q	71	
DC4	14	X	C	43		r	72	
NAK	15	X	D	44		s	73	
SYN	16	X	E	45		t	74	
ETB	17	X	F	46		u	75	
CAN	18	X	G	47		v	76	
EM	19	X	H	48		w	77	
SUB	1A	X	I	49		x	78	
ESC	1B	X	J	4A		y	79	
FS	1C	X	K	4B		z	7A	
GS	1D	X	L	4C			7B	X
RS	1E	X	M	4D			7C	X
US	1F	X	N	4E			7D	X
space	20		O	4F			7E	X
!	21	X	P	50		DEL	7F	X
"	22		Q	51				
#	23	X	R	52				
$	24		S	53				
%	25	X	T	54				
&	26	X	U	55				
'	27	X	V	56				
(	28		W	57				
)	29		X	58				
*	2A		Y	59				
+	2B		Z	5A				
,	2C			5B	X			
-	2D			5C	X			
.	2E			5D	X			

APPENDIX C

GLOSSARY

INTRODUCTION

The terms in this Section are defined in accordance with their meaning as used in this document describing L/II COBOL and may not have the same meaning for other languages.

These definitions are also intended to be either reference material or introductory material to be reviewed prior to reading the detailed language specifications that are contained in this manual. For this reason, these definitions are, in most instances, brief and do not include detailed syntactical rules.

DEFINITIONS

Abbreviated Combined Relation Condition.

The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

Access Mode.

The manner in which records are to be operated upon within a file.

Actual Decimal Point.

The physical representation, using either of the decimal point characters . (period) or , (comma) of the decimal point position in a data item.

Alphabet-Name.

A user-defined word in the SPECIAL-NAMES paragraph of the Environment Division that assigns a name to a specific character set and/or collating sequence.

Alphabetic Character.

A character that belongs to the following set of letters: A,B,C,D,E,F,G,H,I,J,K, L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z and the space. Also a,b,c,d,e,f,g,h,i,j,k, l,m,n,o,p,q,r,s,t,u,v,w,x,y and z which are converted to their upper case equivalents.

Alphanumeric Character.

Any character in the computer's character set.

Arithmetic Expression.

An arithmetic expression can be an identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

Arithmetic Operator.

A single character, or a fixed two-character combination, that belongs to the following set:

Character	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Ascending Key.

A key upon the values of which data is ordered starting with the lowest value of key up to the highest value of key in accordance with the rules for comparison of the data items.

Assumed Decimal Point.

A decimal point position which does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

At End Condition.

A condition caused in one of two circumstances:

1. During the execution of a READ statement for a sequentially accessed file.

2. During the execution of a RETURN statement when no next logical record exists for the associated sort or merge file.

Block.

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either continued within the block or that overlap the block. The term is synonymous with physical record.

Cd-Name.

A user-defined word that names an MCS interface area described in a communication description entry within the Communication Section of the Data Division.

Called Program.

A program which is the object of a CALL statement combined at run time with the calling program to produce a run unit.

Calling Program.

A program which executes a CALL to another program.

Character.

The basic indivisible unit of the language.

Character Set (L/II COBOL).

The complete L/II COBOL character set consists of all characters listed below:

Character	Meaning
0, 1, ..., 9	Numeric digit
A, B, ..., Z	Uppercase alphabetic
a, b, ..., z	Lowercase alphabetic
	Space (Blank)
+	Plus Sign
-	Minus Sign
*	Asterisk
/	Stroke (Virgule or Slash)
=	Equal Sign
$	Currency Sign
,	Comma
;	Semicolon
.	Period (Decimal Point, Fullstop)
'	Quotation Mark
(	Left Parenthesis
)	Right Parenthesis
>	Greater Than Symbol
<	Less Than Symbol

Character Position.

A character position is the amount of physical storage required to store a single standard data format character described as usage in DISPLAY. Further characteristics of the physical storage are defined by the implementor.

Character-String.

A sequence of contiguous characters which form a L/II COBOL word, a literal, a PICTURE character-string or a comment-entry.

Class Condition.

The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric.

Clause.

A clause is an ordered set of consecutive L/II COBOL character-strings whose purpose is to specify an attribute of an entry.

COBOL Word.

(See Word)

Collating Sequence.

The sequence in which the characters that are acceptable in a computer are ordered for purposes of sorting, merging and or comparing.

Column.

A character position within a print line. The columns are numbered from one, by one, starting at the left-most character position of the print line and extending to the right-most character position of the print line.

Combined Condition.

A condition that is the result of connecting two or more conditions with the 'AND' or the 'OR' logical operator.

Comment Entry.

An entry in the Identification Division that may be any combination of characters from the computer character set.

Comment Line.

A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a stroke (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection before printing the comment.

Communication Description Entry.

An entry in the Communication Section of the Data Division that is composed of the level indicator CD, followed by a cd-name, and then followed by a set of clauses as required. It describes the interface between the Message Control System (MCS) and the COBOL program.

Communication Device.

A mechanism (hard or hardware/software) capable of sending data to a queue and/or receiving data from a queue. This mechanism may be a computer or a peripheral device. One or more programs containing communication description entries and residing within the same computer define one or more of these mechanisms.

Communication Section.

The section of the Data Division that describes the interface areas between the MCS and the program, composed of one or more CD description entries.

Compile Time.

The time at which an L/II COBOL source program is translated by the compiler to an L/II COBOL Intermediate code program.

Compiler-Directing Statement.

A statement, beginning with a compiler-directing verb, that causes the compiler to take a specific action during compilation.

Complex Condition.

A condition in which one or more logical operators act upon one or more conditions. (See Negated Simple Condition, Combined Condition, Negated Combined Condition).

Computer-Name.

A system-name that identifies the computer upon which the program is to be compiled or run.

Condition.

A status of a program at execution time for which a truth value can be determined. Where the term "condition" (condition-1, condition-2, ...) appears in these language specifications in or in reference to "condition" (condition-1, condition-2, ...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a negated simple condition.

Condition Name.

The user-defined word assigned to a status of an implementor-defined switch or device.

Condition-Name Condition.

The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

Conditional Expression.

A simple condition specified in an IF, or PERFORM. (See Simple Condition and Complex Condition.)

Conditional Statement.

A conditional statement specifies that the truth value of a condition is to be determined, and that the subsequent action of the run-time program is dependent on this truth value.

Conditional Variable.

A data item one or more values of which has a condition-name assigned to it.

Configuration Section.

A section of the Environment Division that describes overall specifications of source and run computers.

Connective.

A reserved word that is used to:

1. Associate a data-name, paragraph-name, condition-name, or text-name with its qualifier.
2. Link two or more operands written in a series.
3. Form conditions (logical connectives). (See Logical Operator.)

Contiguous Items.

Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to one another.

Counter.

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

Currency Sign.

The character "$" (dollar sign) in the L/II COBOL character set.

Currency Symbol.

The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in an L/II COBOL source program, the currency symbol is identical to the currency sign.

Current Record.

The record which is available in the record area associated with the file.

Current Record Pointer.

A conceptual entity that is used in the selection of the next record.

Cursor.

The indicator on a CRT screen that marks the line and character position which the input/output control is currently referencing.

Data Clause.

A clause that appears in a data description entry in the Data Division and provides information describing a particular attribute of a data item.

Data Description Entry.

An entry in the Data Division that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses as required.

Data Dictionary.

A dictionary file of user defined names constructed by the Compiler containing the number of bytes of each entry.

Data Item.

A character or set of contiguous characters (excluding, in either case, literals) defined as a unit of data by the L/II COBOL program.

Data-name.

A user-defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, "data-name" represents a word which can neither be subscripted, nor indexed unless specifically permitted by the rules for that format.

Debugging Line.

A debugging line is any line with "D" in the indicator area of the line.

Debugging Section.

A debugging section is a section that contains a USE FOR DEBUGGING statement.

Declaratives.

A set of one or more special purpose sections written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sequence, followed by a set of associated paragraphs (0 or more).

Declarative-Sentence.

A compiler-directing sentence consisting of a single USE statement terminated by the separator period (.).

Default Disk.

The disk from which the compiler or run-time system is loaded.

Delimiter.

A character (or sequence of contiguous characters) that identifies the end of a string of characters, and separates that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending Key.

A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

Destination.

The symbolic identification of the receiver of a transmission from a queue.

Digit Position.

A digit position is the amount of physical storage required to store a single digit. This amount varies depending on the usage of the data item describing the digit position. Further characteristics of the physical storage are defined by the implementor.

Division.

A set of sections or paragraphs (0 or more) that are formed and combined in accordance with a specific set of rules are called a division body. There are 4 divisions in an L/II COBOL program: Identification, Environment, Data and Procedure.

Division Header.

A combination of words followed by a period and a space that indicate the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION    USING data-name-1   data-name-2 ... .
```

Dynamic Access.

An access mode in which specific logical records can be obtained from or placed into a disk file in a non-sequential manner (see Random Access) and obtained from a file in a sequential manner (see Sequential Access) during the scope of the same OPEN statement.

Editing Character.

A single character or a fixed two character combination belonging to the same set.

Character	Meaning
B	Space
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
Z	Zero Suppress
*	Check Protect
$	Currency Sign
,	Comma
.	Period (Decimal Point)
/	Stroke (Virgule, Slash)

Elementary Item.

A data item that is described as not being further logically subdivided.

End of Procedure Division.

The physical position in a L/II COBOL source program after which no further procedures appear.

Entry.

Any descriptive set of consecutive clauses terminated by a period (.) and written in the Identification Division, Environment Division or Data Division of an L/II COBOL source program.

Environment Clause.

A clause that appears as part of an Environment Division entry.

Extend Mode.

With the EXTEND phrase specified, the state of a file after execution of an OPEN statement, and before the execution of a CLOSE statement for the file.

Figurative Constant.

A compiler-generated value referenced through the use of certain reserved words.

File.

A collection of records.

File Clause.

A clause that appears as part of any of the following Data Division entries:

File Description (FD)
Sort-Merge File Description (SD)
Communication Description (CD)

FILE-CONTROL.

The name of an Environment Division paragraph in which the data files for a given source program are declared.

File Description Entry.

An entry in the File Section of the Data Division that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

File-Name.

A user-defined word that names a file described in a file description entry or a sort-merge file description entry within the File Section of the Data Division.

File Organization.

The permanent logical file structure established at the time that a file is created.

File Section.

The section of the Data Division that contains file description entries together with their associated record descriptions.

Format.

A specific arrangement of a set of data.

Group Item.

A named contiguous set of elementary or group items.

High Order End.

The leftmost character of a string of characters.

I-O-CONTROL.

The name of an Environment Division paragraph in which object program requirements for specific input/output techniques, rerun points, sharing of same areas by several data files, and multiple file storage on a single input/output device are specified.

I-O Mode.

The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement for that file.

Identifier.

A data-name, followed as required, by the syntactically correct combination of subscripts and indices necessary to make unique reference to a data item.

Imperative Statement.

A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement may consist of a sequence of imperative statements.

Implementor-Name.

A system-name that refers to a particular feature available on the implementors computing system.

Index.

A computer storage position or register, the contents of which represent the identification of a particular element in a table.

Index Data Item.

A data item in which the value associated with an index-name can be stored in a form specified by the implementor.

Index-Name.

A user-defined word that names an index associated with a specific table.

Indexed Data-Name.

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

Indexed File.

A file with indexed organization.

Indexed Organization.

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

Indicator Area.

The leftmost parameter position of a L/II COBOL source record that indicates the use of the record.

Input File.

A file that is opened in the input mode.

Input Mode.

The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement for that file.

Input-Output File.

A file that is opened in the I-O mode.

Input-Output Section.

The section of the Environment Division that names the files and the external media by a program and which provides information required for transmission and handling of data during execution of the run-time program.

Input Procedure.

A set of statements that is executed each time a record is released to the sort file.

Integer.

A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point. Where the 'integer' appears in general formats, integer must not be a numeric data item, and must not be signed, nor zero unless explicitly allowed by the rules of that format.

Intermediate Code.

The code produced by the L/II COBOL compiler from the source code entered, and which the Run Time System 'fast loads' for execution.

Invalid Key Condition.

A condition, at object time, caused when a specified value of the key associated with an indexed or relative file is determined to be invalid.

Key.

A data item which identifies the location of a record, or a set of data items which serve to identify the ordering of data.

Key of Reference.

The key currently being used to access records within an indexed file.

Key Word.

A reserved word whose presence is required when the format in which the word appears is used in a source program.

Language-Name.

A system-name that specifies a particular programming language.

Level Indicator.

Two alphabetic characters that identify a specific type of file or a position in hierarchy.

Level-Number.

A user-defined word which indicates the position of a data item in the hierarchical structure of a logical record or which indicates special properties of a data description entry. A level-number is expressed as a one or two digit number. Level-numbers in the range 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record.

Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-number 77 identifies special properties of a data description entry.

Library-Name.

A user-defined word that names a L/II COBOL library intermediate file that is to be used by the compiler for a given source program compilation.

Library-Text.

A sequence of character-string and/or separators in a COBOL Library.

Line Sequential File Organization.

A sequential file containing variable-length records separated by the new line character.

Linkage Section.

The section in the Data Division of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

Literal.

A character-string whose value is implied by the ordered set of chaacters comprising the string.

Logical Operator.

The reserved word 'NOT'. It can be used for logical negation.

Logical Record.

The most inclusive data item. The level-number for a record is 01.

Low Order End.

The rightmost character of a string of characters.

MCS.

(See Message Control System).

Merge File.

A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

Message.

Data associated with an end of message indicator or an end of group indicator. (See Message Indicators)

Message Control System (MCS).

A communication control system that supports the processing of messages.

Message Count.

The count of the number of complete messages that exist in the designated queue of messages.

Message Indicators.

EGI (end of group indicator), EMI (end of message indicator), and ESI (end of segment indicator) are conceptual indications that serve to notify the MCS that a specific condition exists (end of group, end of message, end of segment).

Within the hierarchy of EGI, EMI, and ESI, an EGI is conceptually equivalent to an ESI, EMI, and EGI. An EMI is conceptually equivalent to an ESI and EMI. Thus, a segment may be terminated by an ESI, EMI, or EGI. A message may be terminated by an EMI or EGI.

Message Segment.

Data that forms a logical subdivision of a message normally associated with an end of segment indicator. (See Message Indicators).

Mnemonic-Name.

A user-defined word that is associated in the Environment Division with a specified implementor-name.

Native Character Set.

The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

Native Collating Squence.

The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

Negated Combined Condition.

The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated Simple Condition.

The 'NOT' logical operator immediately followed by a simple condition.

Next Executable Sentence.

The next sentence to which control will be transferred after execution of the current statement is complete.

Next Executable Statement.

The next statement to which control will be transferred after execution of the correct statement is complete.

Next Record.

The record which logically follows the current record of a file.

Noncontiguous Items.

Elementary data items, in the Working-Storage and Linkage Sections, which bear no hierarchic relationship to other data items.

Nonnumeric Item.

A data item whose description permits its contents to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

Nonnumeric Literal.

A character-string bounded by quotation marks. The string of characters may include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

Numeric Character.

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric Item.

A data item whose description restricts its contents to a value represented by characters chosen from the digits '0' through '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

Numeric Literal.

A literal composed of one or more numeric characters that also may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

OBJECT-COMPUTER.

The name of an Environment Division paragraph in which the computer environment, within which the run-time program is executed, is described.

Open Mode.

The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

Operand.

Whereas the general definition of operand is 'that component which is operated upon', for the purposes of this publication, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

Operational Sign.

An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Optional Word.

A reserved word that is included in a specified format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

Output File.

A file that is opened in either the output mode or extend mode.

Output Mode.

The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified for that file and before the execution of a CLOSE statement for that file.

Output Procedure.

A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function has selected the next record in merged order.

Paragraph.

In the Procedure Division, a paragraph-name followed by a period and a space and optionally by one, or more sentences. In the Identification and Environment Divisions, a paragraph header followed by zero, one, or more entries.

Paragraph Header.

A reserved word, followed by a period and a space that indicates the beginning of a paragraph in the Identification and Environment Divisions. The permissible paragraph headers are:

In the Identification Division:

PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.

In the Environment Division:

SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.

Paragraph-Name.

A user-defined word that identifies and begins a paragraph in the Procedure Division.

Phrase.

A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a L/II COBOL procedural statement or of a COBOL clause.

Physical Record.

(See Block)

Prime Record Key.

A key whose contents uniquely identify a record within an indexed file.

Procedure.

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

Procedure-Name.

A user-defined word which is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name or a section-name.

Program-Name.

A user-defined word that identifies a COBOL source program.

Pseudo-Text.

A sequence of character-strings and/or separators bounded by, but not including, pseudo-text delimiters.

Pseudo-Text Delimiter.

Two contiguous equal sign (=) characters used to delimit pseudo-text.

Punctuation Character.

A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
.	period
"	quotation mark
(	left parenthesis
)	right parenthesis
	space
=	equal sign

Qualified Data-Name.

An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

Qualifier.

1. A data-name which is used in a reference together with another data name at a lower level in the same hierarchy.
2. A section-name which is used in a reference together with a paragraph-name specified in that section.
3. A library-name which is used in a reference together with a text-name associated with that library.

Queue.

A logical collection of messages awaiting transmission or processing.

Queue Name.

A symbolic name that indicates to the MCS the logical path by which a message or a portion of a completed message may be accessible in a queue.

Random Access.

An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from or placed into a relative or indexed file.

Record.

(see Logical Record)

Record Area.

A storage area allocated for the purpose of processing the record described in a record description entry in the File Section.

Record Description.

(See Record Description Entry)

Record Description Entry.

The total set of data description entries associated with a particular record.

Record Key.

A key, either the prime record key or an alternate record key, whose contents identify a record within an indexed file.

Record-Name.

A user-defined word that names a record described in a record description entry in the Data Division.

Reference-Format.

A format that provides a standard method for describing COBOL source programs.

Relation.

(See Relational Operator)

Relation Character.

A character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

Relation Condition.

The proposition, for which a truth value can be determined, that the value of an arithmetic expression or data item has a specified relationship to the value of another arithmetic expression or data item. (See Relational Operator.)

Relational Operator.

A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meaning are:

Relational Operator	Meaning
IS NOT GREATER THAN IS NOT >	Greater than or not greater than
IS NOT LESS THAN IS NOT<	Less than or not less than
IS NOT EQUAL TO IS NOT =	Equal to or not equal to

Relative File.

A file with relative organization.

Relative Key.

A key whose contents identify a logical record in a relative file.

Relative Organization.

The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

Reserved Word.

A COBOL word specified in the list of words which may be used in COBOL source programs, but which must not appear in the programs as user-defined words or system-names.

Routine-Name.

A user-defined word that identifies a procedure written in a language other than COBOL.

Run-Time Debug.

An option available to L/II COBOL programmers entered as a user option enabling break-point facilities in run-time programs.

Run-Time.

The time at which the intermediate code produced by the compiler is interpreted by the Run-Time-System for execution.

Run-Time-System-(RTS).

The software that interprets the intermediate code produced by the L/II COBOL compiler and enables it to be executed by providing interfaces to the operating system and CRT.

Run Unit.

A set of one or more intermediate code programs which function, at run time, as a unit to provide problem solutions.

Section.

A set of none, one, or more paragraphs or entries, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section Header.

A combination of words followed by a period and a space that indicates the beginning of a section in the Environment, Data and Procedure Division.

In the Environment and Data Divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers are:

In the Environment Division:

CONFIGURATION SECTION
INPUT-OUTPUT SECTION

In the Data Division:

FILE SECTION
WORKING-STORAGE SECTION
LINKAGE SECTION

In the Procedure Division, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a segment-number (optional), followed by a period and a space.

<u>Section-Name.</u>

A user-defined word which names a section in the Procedure Division.

<u>Segment-Number.</u>

A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0', '1', ..., '9'. A segment-number may be expressed either as a one or two digit number, and is checked for syntax only.

<u>Sentence.</u>

A sequence of one or more statements, the last of which is terminated by a period followed by a space.

<u>Separator.</u>

A punctuation character used to delimit character-strings.

<u>Sequential Access.</u>

An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

<u>Sequential File.</u>

A file with sequential organization.

<u>Sequential Organization.</u>

The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

<u>Sign Condition.</u>

The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

Simple Condition.

Any single condition chose from the set:

relation condition
class condition
switch-status condition
sign condition
(simple-condition)

Sort File.

A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

Sort-Merge File Description Entry.

An entry in the File Section of the Data Division that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

Source.

The symbolic definition of the originator of a transmission to a queue.

SOURCE-COMPUTER.

The name of an Environment Division paragraph in which the computer environment, within which the source program is compiled, is described.

Source Program.

Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements beginning with an Identification Division and ending with the end of the Procedure Division. In contents where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'source program'.

Special Character.

A character that belongs to the following set:

Character	Meaning
+	plus sign
-	minus sign
*	asterisk
/	stroke (virgule or slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

Special-Character Word.

A reserved word which is an arithmetic operator or a relation character.

SPECIAL-NAMES.

The name of an Environment Division paragraph in which implementor-names are related to user specified mnemonic-names.

Special Registers.

Compiler generated storage area whose primary use is to store information produced in conjunction with the user of specified COBOL features.

Standard Data Format.

The concept used in describing the characteristics of data in a COBOL Data Division under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

Statement.

A syntactically valid combination of words and symbols written in the Procedure Division beginning with a verb.

Sub-Queue.

A logical hierarchical division of a queue.

Subject of Entry.

An operand or reserved word that appears immediately following the level indicator or the level-number in a Data Division entry.

Subprogram.

(See Called Program)

Subscript.

An integer whose value identifies a particular element in a table.

Subscripted Data-Name.

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

Switch-Status Condition.

The proposition, for which a truth value can be determined, that an implementor-defined switch, capable of being set to an 'on' or 'off' status, has been set to a specified status.

Symbol Function.

The use of specified characters in the PICTURE clause to represent data types.

System-Name.

A COBOL word which is used to communicate with the operating environment.

Syntax.

The order in which elements must be put together to form a program.

Table.

A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Table Element.

A data item that belongs to the set of repeated items comprising a table.

Terminal.

The originator of a transmission to a queue, or the receiver of a transmission from a queue.

Text-Name.

A user-defined word which identifies library text.

Text-Word.

Any character-string or separator, except space, in a COBOL library or in pseudo-text.

Truth Value.

The representation of the result of the evaluation of a condition in terms of one of two values:

true
false

Unary Operator.

A plus (+) or a minus (-) sign, which precedes a variable or a left parenthesis in an arithmetic expression and which has the effect of multiplying the expressing of +1 or -1 respectively.

User-Defined Word.

A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

Variable.

A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Verb.

A word that expresses an action to be taken by a COBOL compiler or run-time program.

Word.

A character-string of not more than 30 characters which forms a user-defined word, a system-name, or a reserved word.

Working-Storage Section.

The section of the Data Division that describes working storage data items, composed either of noncontiguous items, or of working storage records, or of both.

77 Level-Description-Entry.

A data description entry that describes a noncontiguous data item with the level-number 77.

APPENDIX D
COMPILE-TIME ERRORS

The error descriptions that correspond to error numbers as printed on listings produced by the L/II COBOL compiler are listed below. In the case of alternative meanings, relevancy is obvious from context.

ERROR	DESCRIPTION
01	Compiler Error
02	Illegal format: data-name
03	Illegal format: literal
04	Illegal format: character
05	Declaration violates uniqueness of qualification
06	Too many data and procedure names have been declared
07	Obligatory reserved word missing
08	Nested COPY or unkown library text
09	'.' missing
10	The statement starts in the wrong area of the source line, i.e., reference format violation
22	'DIVISION' missing
23	'SECTION' missing
24	'IDENTIFICATION missing
25	'PROGRAM-ID' missing
26	'AUTHOR' missing
27	'INSTALLATION' missing
28	'DATE-WRITTEN' missing
29	'SECURITY' missing
30	'ENVIRONMENT' missing
31	'CONFIGURATION' missing
32	'SOURCE-COMPUTER' missing
33	MEMORY SIZE/COLLATING SEQUENCE/SPECIAL-NAMES clause in error
34	'OBJECT-COMPUTER' MISSING
36	'SPECIAL-NAMES' missing
37	SWITCH Clause in error <u>or</u> system-name/ mnemonic-name error
38	DECIMAL-POINT Clause in error
39	CONSOLE Clause in error
40	Illegal currency symbol
42	'DIVISION' missing
43	'SECTION' missing
44	'INPUT-OUTPUT' missing
45	'FILE-CONTROL' missing
46	'ASSIGN' missing
47	'SEQUENTIAL' or 'INDEXED' or 'RELATIVE' missing
48	'ACCESS' missing on indexed/relative file
49	'SEQUENTIAL/DYNAMIC' missing <u>or</u> too many alternate keys (>64)
50	Illegal ORGANIZATION/ACCESS/KEY combination
51	Unrecognized phrase in 'SELECT' clause
52	Syntax error in 'RERUN' clause
53	Syntax error in 'SAME AREA' clause
54	file-name missing or illegal

55	'DATA DIVISION' missing
56	'PROCEDURE DIVISION' missing or unknown statement
57	'EXCLUSIVE', 'AUTOMATIC' or 'MANUAL' missing
58	Non-exclusive lock mode specified for restricted file
62	'DIVISION' missing
63	'SECTION' missing
64	file-name not specified in SELECT statement
65	Record size integer missing
66	Illegal level number (01-49) or 01 level required
67	FD qualification contains syntax error
68	'WORKING-STORAGE' missing
69	'PROCEDURE DIVISION' missing or unknown statement
70	Data Description Qualifier or '.' missing
71	Incompatible PICTURE Clause and qualifiers
72	'BLANK' is illegal with nonnumeric data-item
73	PICTURE clause too long (Numeric 18 Numeric Edited 512 Alphanumeric 8192)
74	VALUE clause on non-elementary data-item, or truncation, or wrong data type
75	'VALUE' in error or illegal for PICTURE type
76	FILLER/SYNCHRONIZED/JUSTIFIED/BLANK non-elementary item
77	Preceding item at this level has more than 8192 bytes or 0 bytes
78	REDEFINES of unequal fields or different levels
79	Data storage exceeds 64K bytes
81	Data Description Qualifier inappropriate or repeated
82	REDEFINES data-name not declared
83	USAGE must be COMP, DISPLAY or INDEX
84	SIGN must be LEADING or TRAILING
85	SYNCHRONIZED must be LEFT or RIGHT
86	JUSTIFIED must be RIGHT
87	BLANK must be ZERO
88	OCCURS must be numeric, non-zero and unsigned
89	VALUE must be a literal, numeric literal or figurative constant
90	PICTURE string has illegal precedence or illegal character
91	INDEXED data-name missing or already declared
92	numeric edited PICTURE string is too large
101	Unrecognized verb
102	'IF' ... 'ELSE' mismatch
103	Wrong data-type or data-name not declared
104	Procedure name declared twice
105	Procedure name same as data-name
106	Name required
107	Wrong combination of data types
108	Conditional statement not allowed in this context; must be an imperative statement
109	Malformed subscript
110	ACCEPT/DISPLAY wrong
111	Illegal I-O Syntax
112	'LOCK' clause specified for file with lock mode 'EXCLUSIVE'
113	'KEPT' specified for uncommittable file

115	'KEPT' omitted for committable file
116	IF statements nested too deep
117	Incorrect structure of Procedure Division, e.g., Sections out of order
118	Reserved Word missing, or incorrectly used
119	Too many subscripts in one statement
120	Too many operands in one statement
141	Inter-segment procedure name duplication
142	'IF' ... "ELSE' mismatch at end of Source Input
143	Wrong data-type or data-name not declared
144	Procedure name undeclared
145	INDEX data-name declared twice
146	Bad cursor control: AT clause incorrectly specified
147	KEY declaration missing
148	STATUS declaration missing
149	Bad STATUS record
150	Undefined inter-segment reference, or error in ALTERed paragraph
151	PROCEDURE DIVISION in error
152	USING parameter not declared in Linkage Section
153	USING parameter is not level 01 or 77
154	USING parameter used twice in parameter list
157	Incorrect structure of Procedure Division: e.g., Sections out of order
160	Too many operands in one statement

In addition to these numbered error messages, the following message can be displayed with subsequest termination of the compilation:

FATAL I-0 ERROR: filename

where filename is the erroneous file.

Any intermediate code file produced is not usable.

The following conditions will cause this error:

Disk overflow
File directory overflow
File full
Impossible I-O device usage

Other operating system dependent conditions can also cause this error.

NOTE

You will notice that the numbers of the numbered error messages listed above are not continuous, i.e., there are gaps in the numbering. The compiler should never have cause to generate an error message with a number not listed above.

APPENDIX E

COBOL RUN-TIME ERRORS

If the COBOL run-time system detects an error condition while executing a COBOL program, program execution is terminated and an error report is displayed.

If the error is due to filling in a command form incorrectly, one of the following messages is displayed.

Improper Yes/No input in command form

> Fill in fields that require Yes/No input with Y or N followed by RETURN. Not responding to a field is the same as entering N.

Improper input in command form: use single parameter

> Fill in fields that require a parameter with only one parameter. Parameter lists are not accepted.

Missing name of source file

> You must enter a filename in the first field of the COBOL command form.

If an error is detected while loading intermediate code, the message,

```
Error on loading file<filename>
```

is displayed.

Otherwise, the run-time system displays,

```
Error detected while executing XXX in segment YY at COBOL program
address ZZZZH
```

where

XXX	is the filename of the currently executing intermediate code.
YY	is the current segment number or RT if the current segment is not independent.
ZZZZ	is a program address that corresponds to the locations address printed along the right side of the COBOL program listing.

The run-time system next displays a message that describes the error that was detected.

If an error is detected during a Sort/Merge or Indexed Sequential file operation, the run-time system also displays

Detailed status: WWWW

where

WWWW	is a status code. Status codes are described in the **B 20 BTOS Manual.**

After displaying the error report, the run-time system displays a prompt message and waits until a key is pressed before returning to the Executive.

After return to the Executive, the Executive may display an additional status message that gives more information about the run-time error.

The error messages displayed by the COBOL run-time system are described below in alphabetic order. The number in parentheses following the message is the run-time system error number.

Attempt to open file failed: file not found (183)

Make sure that the input file exists and is in the correct directory.

File operation failed: check ORGANIZATION and ACCESS (168)

The attempted file access is not allowed according to the file's ORGANIZATION and ACCESS attributes.

File operation failed: file not open for OUTPUT (156)

A file must be opened for OUTPUT or I-0 before it can be written.

Illegal inter-segment reference (176)

An illegal flow of control between segments has been attempted. See the Segmentation chapter of the COBOL Manual for details about restrictions on program flow.

Illegal intermediate code (161)

The file containing the executing intermediate code has been corrupted or there is an internal error in the COBOL run-time system.

Illegal literal operands (163)

An internal error has occurred in the COBOL run-time system.

Illegal variable length count (193)

An internal error has occurred in the COBOL run-time system.

Improper input in command form: check switches (155)

If you are setting more than one switch, enclose the switch parameters in single quotes. For example, use '+1+2' to turn on switches 1 and 2.

Incompatible operation for indexed file lock mode (173)

Access to the indexed file failed because of the lock mode of the file. See Appendix J for details on using locks with indexed files.

Incompatible operation for indexed file open mode (172)

Access to the indexed file failed because of the open mode of the file. A file must be opened in OUTPUT or I-O mode to be written and INPUT or I-O mode to be read.

Incompatible releases of compiler and run-time system (165)

To use the current release of the COBOL run-time system, you must recompile your COBOL program with the current release of the COBOL compiler.

Internal error (199)

An internal error has occurred in the COBOL run-time system.

Invalid DELETE Operation for indexed file (170)

Access to the indexed file failed.

Invalid REWRITE Operation for indexed file (171)

Access to the indexed file failed.

Intermediate code file too large (157)

The intermediate code file cannot be loaded. The COBOL Program that generated the large file must be split into two or more modules.

Malformed intermediate code file (181)

The intermediate code file has been corrupted. The program that generated this file must be recompiled.

Module is already active (166)

An attempt to recursively CALL a COBOL module has failed. Recursive calls are not allowed.

Non-COBOL procedure not found (190)

Check the spelling of the non-COBOL procedure is the CALL statement. Check that the version of Cobol.run on the sys directory has been linked with the target non-COBOL procedure. See Appendix J for details on configuring COBOL to call non-COBOL procedures.

Not enough memory to continue (167)

Make more memory available or reduce the size of your COBOL program.

Random read attempted on a sequential file (151)

Only RELATIVE and INDEXED files can be accessed randomly.

REWRITE attempted on a file not open I-O (152)

A file must be opened in mode I-O to be rewritten.

REWRITE attempted on a line sequential file (158)

Line sequential files cannot be rewritten. Use a sequential file.

Return to Executive for a status message (182, 189)

The status message displayed after COBOL returns to the Executive identifies the problem.

Subscript out of range (153)

A table index is beyond the range of the table.

Too few parameters to non-COBOL procedure (192)

Check the interface of the non-COBOL procedure. Make sure you are passing the required number of arguments. If the procedure returns a value, pass an extra argument at the beginning of the parameter list to receive the returned value.

Too many parameters to non-COBOL procedure (191)

Check the interface of the non-COBOL procedure. Make sure you are passing the required number of arguments.

Unable to load COBOL intermediate code file (164)

The intermediate code file does not exist or it is determined not to contain intermediate code.

The error messages are repeated below in numeric order.

151	Random read attempted on a sequential file
152	REWRITE attempted on a file not open I-O
153	Subscript out of range
155	Improper input in command form: check switches
156	File operation failed: file not open for OUTPUT

157	Intermediate code file too large
158	REWRITE attempted on a line sequential file
161	Illegal intermediate code
163	Illegal literal operands
164	Unable to load COBOL intermediate code file
165	Incompatible releases of compiler and run-time system
166	Module is already active
167	Not enough memory to continue
168	File operation failed: check ORGANIZATION and ACCESS
170	Invalid DELETE operation for indexed file
171	Invalid REWRITE operation for indexed file
172	Incompatible operation for indexed file open mode
173	Incompatible operation for indexed file lock mode
176	Illegal inter-segment reference
181	Malformed intermediate code file
182	Return to Executive for a status message
183	Attempt to open file failed: file not found
189	Return to Executive for a status message
190	Non-COBOL procedure not found
191	Too many parameters to non-COBOL procedure
192	Too few parameters to non-COBOL procedure
193	Illegal variable length count
199	Internal error

APPENDIX F

SYNTAX SUMMARY

All the syntax for L/II COBOL is summarized below. E denotes that the feature is a L/II COBOL extension to ANSI COBOL. D denotes that the feature is documentary only in L/II COBOL.

GENERAL FORMAT FOR IDENTIFICATION DIVISION

```
[IDENTIFICATION DIVISION.]

[PROGRAM-ID.  program name]

[AUTHOR.           [comment entry] ...]

[INSTALLATION.     [comment entry] ...]

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

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

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

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
[ENVIRONMENT DIVISION.]

[CONFIGURATION SECTION.]

[SOURCE-COMPUTER. source-computer-entry      [WITH DEBUGGING MODE].]

[OBJECT-COMPUTER. object-computer-entry

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

     [,PROGRAM COLLATING SEQUENCE IS alphabet-name].

[SPECIAL-NAMES.

     [,{SYSIN   IS mnemonic-name-1 ]
     [ {SYSOUT                     ]

     [, TAB     IS mnemonic-name-2 ]

     [            { 0 }
     [            { . }
     [ SWITCH     { . }   [IS mnemonic-name]   ON STATUS IS condition-name-1
     [            {   }
     [            { . }
     [            { 7 }

                  [OFF STATUS IS condition-name-2] ]

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

     [,CURRENCY SIGN IS literal-9]
     [,DECIMAL-POINT IS COMMA]
     [,CURSOR IS data-name-1]            E
     [,CONSOLE IS CRT]                   E
[INPUT-OUTPUT SECTION.
 FILE-CONTROL.
    { file-control-entry } ...  . ]
[I-O-CONTROL.
       [ ; RERUN  [ON { file-name-1       } ]
                       { implementor-name  }

                          ( { END OF { REEL }  }        OF file-name-2 )     ]   D
                  EVERY (  {        { UNIT }  }                        )     ]
                          ( { integer-1 RECORDS. }                       )     ]
                          ( integer-2 CLOCK-UNITS                        )
                          ( condition-name                               )

     [ ;SAME    [ RECORD      ]      AREA FOR file-name-3  ,file-name-4 ... ] ...
                [ SORT        ]
                [ SORT-MERGE  ]
     [ ; MULTIPLE FILE TAPE CONTAINS file-name-5   [POSITION integer-3]
              [, file-name-6  [POSITION integer-4]  ]    ... ] ... ] .
```

GENERAL FORMAT FOR FILE-CONTROL ENTRY

Sequential SELECT:

```
    SELECT file-name       [ OPTIONAL ] file-name

    ASSIGN TO {external-file-name-literal}        [, {external-file-name-literal}]
              {file-identifier           }        [  {file-identifier           }]

   [; RESERVE integer-1      [{AREA }]]
   [                         [{AREAS}]]                   D

    ;ORGANIZATION IS[{SEQUENTIAL     }]
                    [{LINE SEQUENTIAL}]                   E

    [;ACCESS MODE IS SEQUENTIAL]

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

Relative Select:

```
    SELECT file-name

    ASSIGN TO {external-file-name-literal}        [, {external-file-name-literal}]
              {file-identifier           }        [  {file-identifier           }]

   [; RESERVE integer-1      [{AREA }]]
   [                         [{AREAS}]]                   D

    ORGANIZATION IS    RELATIVE

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

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

Indexed Select:

```
    SELECT file-name

    ASSIGN TO {external-file-name-literal}        [, {external-file-name-literal}]
              {file-identifier           }        [  {file-identifier           }]

   [; RESERVE integer-1         [{AREA }]]
   [                            [{AREAS}]]                 D
```

;$\underline{\text{ORGANIZATION}}$ IS $\underline{\text{INDEXED}}$

$$\left[\text{[;}\underline{\text{ACCESS}}\text{ MODE IS} \quad \left\{ \begin{array}{l} \underline{\text{SEQUENTIAL}} \\ \underline{\text{RANDOM}} \\ \underline{\text{DYNAMIC}} \end{array} \right\} \right]$$

;$\underline{\text{RECORD}}$ KEY IS data-name-1

[; $\underline{\text{ALTERNATE}}$ $\underline{\text{RECORD}}$ KEY IS data-name-2 [WITH $\underline{\text{DUPLICATES}}$]] ...

[;FILE $\underline{\text{STATUS}}$ IS data-name-3] .

```
[DATA DIVISION.]

[FILE SECTION.]

[FD file-name

   ┌                                                  ┌             ┐┐
   │;BLOCK CONTAINS integer-1  [TO]  integer-2        │RECORDS      ││
   └                                                  └CHARACTERS   ┘┘

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

    ;LABEL  ┌RECORD IS  ┐    ┌STANDARD┐
            └RECORDS ARE┘    └OMITTED ┘

   ┌;VALUE OF  implementor-name-1    IS  ┌data-name-1┐
   │                                     └literal-1  ┘
   └
            ┌,implementor-name-2    IS   ┌data-name-2┐...┐
            └                            └literal-2  ┘   ┘

   ┌;DATA    ┌RECORD IS  ┐   data-name-3    [,data-name-4]...┐
   └         └RECORDS ARE┘                                   ┘

   ┌;LINAGE IS  ┌data-name-5┐    LINES    ┌,WITH FOOTING AT      ┌data-name-6┐┐
   └            └integer-4  ┘             └                      └integer-6  ┘┘
           ┌,LINES AT TOP  ┌data-name-7┐┐   ┌,LINES AT BOTTOM   ┌data-name-8┐┐┐
           └               └integer-7  ┘┘   └                   └integer-8  ┘┘┘

    [;CODE-SET  IS  alphabet-name]

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

┌ WORKING-STORAGE SECTION        ┐
│┌77-level-description-entry┐    │
│└record-description-entry  ┘ ...│
└                                ┘
┌ LINKAGE SECTION                ┐
│┌77-level-description-entry┐    │
│└record-description-entry  ┘ ...│
└                                ┘
┌COMMUNICATION SECTION               ┐
│┌communication-description-entry    │
│└record-description-entry ]...] ... │
└                                    ┘
```

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

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

      [;REDEFINES data-name]

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

   [               (COMPUTATIONAL    )]
   [               (COMP             )]
   [               (COMPUTATIONAL -3 )]
   [;USAGE IS      (COMP-3           )]
   [               (DISPLAY          )]

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

      [; 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]  ...]]

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

      [;  {JUSTIFIED}        RIGHT]
      [   {JUST     }             ]
      [;BLANK WHEN ZERO]
      [;VALUE IS literal] .

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

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

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

GENERAL FORMAT FOR COMMUNICATION DESCRIPTION ENTRY

FORMAT 1:

```
CD cd-name;
                          ┌ [[; SYMBOLIC QUEUE IS data-name-11]                  ┐
                          │                                                      │
                          │      [; SYMBOLIC SUB-QUEUE-1 IS data-name-2]         │
                          │                                                      │
                          │      [; SYMBOLIC SUB-QUEUE-2 IS data-name-3]         │
                          │                                                      │
                          │      [; SYMBOLIC SUB-QUEUE-3 IS data-name-4]         │
                          │                                                      │
                          │      [; MESSAGE DATE IS data-name-5]                 │
   FOR [INTIAL ] INPUT    │                                                      │
                          │      [; MESSAGE TIME IS data-name-6]                 │
                          │                                                      │
                          │      [; SYMBOLIC SOURCE IS data-name-7]              │
                          │                                                      │
                          │      [; TEXT LENGTH IS data-name-8]                  │
                          │                                                      │
                          │      [; END KEY IS data-name-9]                      │
                          │                                                      │
                          │      [; STATUS KEY IS data-name-10]                  │
                          │                                                      │
                          │      [; MESSAGE COUNT IS data-name-11]               │
                          └ [data-name-1, data-name-2, ..., data-name-11]]       ┘
```

FORMAT 2:

```
   CD cd-name; FOR OUTPUT

        [; DESTINATION COUNT IS data-name-1]

        [; TEXT LENGTH IS data-name-2]

        [; STATUS KEY IS data-name-3]

        [[; DESTINATION TABLE OCCURS integer-2 TIMES

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

        [; ERROR KEY IS data-name-4]

        [; SYMBOLIC DESTINATION IS data-name-4]
```

GENERAL FORMAT FOR PROCEDURE DIVISION

Declarative format:

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

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

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

Non-declarative format:

```
PROCEDURE DIVISION [USING data-name-1  [data-name-2]  ...].
     {paragraph-name  [sentence] ... } ....
```

GENERAL FORMAT FOR VERBS

```
ACCEPT  data-name-1 [AT    { data-name-2 } ]     FROM CRT
                     [      { literal-1   } ]

ACCEPT identifier        [FROM CONSOLE]

                          { DATE }
ACCEPT identifier FROM    { DAY  }
                          { TIME }

ACCEPT cd-name MESSAGE COUNT

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

                 [; ON SIZE ERROR imperative-statement]

ADD { identifier-1 }      { identifier-2 } ;    { identifier-3 }   ...
    { literal-1    }      { literal-2    }      { literal 3    }

                 GIVING identifier    [ROUNDED]

                 [; ON SIZE ERROR    imperative-statement]

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

ALTER {procedure-name-1    [TO  PROCEED  TO] procedure-name-2 }       ...

CALL     { identifier-1 }     USING  data-name-1        [, data-name-2] ...
         { literal-1    }

CANCEL   { identifier-1 }   [ ,identifier-2 ]      ...
         { literal-1    }   [ ,literal-2    ]

                          D                D                         D

CLOSE file-name { REEL }         [WITH LOCK] [ , file-name    [WITH LOCK] ]
                { UNIT }

                   [ { REEL }       [ WITH NO REWIND ] ]
                   [ { UNIT }       [ FOR REMOVAL    ] ]
CLOSE file-name-1  [                                   ]
                   [   WITH         { NO REWIND }      ]
                   [                { LOCK      }      ]

[                  [ { REEL }       [ WITH NO REWIND ] ] ]
[                  [ { UNIT }       [ FOR REMOVAL    ] ] ]
[ , file-name-2    [                                   ] ]
[                  [   WITH         { NO REWIND }      ] ]
[                  [                { LOCK      }      ] ]
```

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

COMPUTE   identifier-1  [ ROUNDED ]   [, identifier-2   [ROUNDED] ]...

   = arithmetic-expression [; ON SIZE ERROR imperative-statement]

DELETE  file-name   RECORD   [; INVALID KEY   imperative-statement]    D

         {INPUT }  [TERMINAL]                     {identifier-1}      D
DISABLE  {OUTPUT}              cd-name WITH KEY  {literal-1   }

DISPLAY   {identifier-1}  ,  {identifier-2}   ...     [UPON CONSOLE]
          {literal-1   }     {literal-2   }

DISPLAY   {data-name-1}  [AT {data-name-2}]  UPON  {CRT      }         E
          {literal-3  }  [   {literal-4  }]        {CRT-UNDER}

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

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

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

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

                                                                        D

ENABLE     {INPUT     [TERMINAL]}  cd-name WITH KEY   {identifier-1}
           {OUTPUT              }                     {literal-1   }

ENTER      language-name      [routine-name].                D

EXIT       [PROGRAM].

GO TO procedure-name.

GO TO procedure-name-1 {, procedure-name-2}. . .

      DEPENDING ON identifier

IF condition;   {statement-1  }          [; ELSE statement-2      ]
                {NEXT SENTENCE}          [; ELSE NEXT SENTENCE    ]
```

```
INSPECT  identifier-1     TALLYING
                            { ALL      }          { identifier-3 }
    identifier-2 FOR        { LEADING  }          { literal-2    }
                              CHARACTERS

          [ { BEFORE }      INITIAL             { identifier-4 } ]
          [ { AFTER  }                          { literal-3    } ]

INSPECT  identifier-1     REPLACING

                          identifier-6
    CHARACTERS BY         literal-4

   [,] { ALL     }    ,    { identifier-5 }     BY  { identifier-6 }
       { LEADING }         { literal-4    }         { literal-4    }
       { FIRST   }

          { BEFORE }        INITIAL            { identifier-7 }
          { AFTER  }                           { literal-5    }

INSPECT identifier TALLYING tally-clause REPLACING replacing-clause

                         { ASCENDING  }
MERGE file-name-1  ON    { DESCENDING } KEY data-name-1   [, data-name-2] ...

                         { ASCENDING  }
              [    ON    { DESCENDING } KEY data-name-3   [, data-name-4] ... ] ...

        [COLLATING SEQUENCE IS alphabet-name]

        USING file-name-2, file-name-3 [,file-name-4] ...

      { OUTPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
      {                                    [ { THRU    }                ] }
      {                                                                   }
      { GIVING file-name-5                                                }

MOVE     { identifier-1 }         TO identifier-2         [,identifier-3]
         { literal-1    }

MOVE     { CORRESPONDING } identifier-1 TO identifier-2
         { CORR          }
```

```
MULTIPLY  {identifier-1}        BY identifier-2       [ROUNDED]
          {literal-1   }

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

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

   [, identifier-4 [ROUNDED]  . . .

   [, ON SIZE ERROR imperative-statement]

      ( INPUT file-name-1 [REVERSED        ]    , file-name-2 [REVERSED        ] D
      (                   [WITH NO REWIND  ]                  [WITH NO REWIND  ]
OPEN  ( OUTPUT file-name-3  [WITH NO REWIND]  ,file-name-4  [WITH NO REWIND]  ]... ...
      ( I-O file-name-5          [, file-name-6] ...
      ( EXTEND file-name-7          [, file-name-8] ...

PERFORM procedure-name-1 {THROUGH} procedure-name-2
                         {THRU   }


                                                          identifier-3
PERFORM perform-limits  VARYING {identifier-2  } FROM     index-name-2
                                {index-name-1  }          literal-1

                                {identifier-4}
                        BY      {literal-2   }   UNTIL    condition-1

                                                          identifier-6
                     [  AFTER   {identifier-5  } FROM     index-name-4
                     [          {index-name-4  }          literal-3

                                {identifier-7}
                        BY      {literal-4   }   UNTIL    condition-2

                                                          identifier-9
                     [  AFTER   {identifier-8  } FROM     index-name-6
                     [          {index-name-5  }          literal-5

                                {identifier }
                        BY      {literal-6  }    UNTIL    condition-3 ]]

READ file-name    [NEXT]   RECORD     [INTO identifier]

          [;AT END imperative-statement]

READ file-name    RECORD    [INTO identifier]  [;KEY IS data-name]

          [;INVALID KEY imperative-statement]
```

```
RECEIVE cd-name  {MESSAGE} INTO identifier-1   [;NO DATA imperative-statement]
                 {SEGMENT}

RELEASE record-name   [FROM identifier]

RETURN file-name RECORD   [INTO identifier]    ; AT END imperative-statement

REWRITE  record-name   [FROM identifier]

         [;INVALID KEY imperative-statement]

SEARCH   identifier-1          [ VARYING       {identifier-2  } ]
                                               {index-name-1  }

      [; AT END imperative-statement-1]
     [ ; WHEN condition-1          {imperative-statement-2}
                                   {NEXT SENTENCE         }

                                   {imperative-statement-3} ]
       ; WHEN condition-2          {NEXT SENTENCE         } ]

SEARCH ALL identifier -1         [;AT END imperative-statement-1]

                                                           {identifier-3           }}
                                       IS EQUAL TO         {literal-1              }}
       ; WHEN         { data-name-1    IS =                {arithmetic-expression-1}}
                      {
                      { condition-name-1

     [                                                     {identifier-4           }]
     [                                 IS EQUAL TO         {literal-2              }]
     [ AND            { data-name-2    IS =                {arithmetic-expression-2}]
     [                {                                                             ]
     [                { condition-name-2                                            ]
      imperative-statement-2
      NEXT SENTENCE

SEND cd-name [FROM identifier-1]

                                     { WITH identifier-2 }
                                     { WITH ESI          }
SEND cd-name   [FROM identifier-1]   { WITH EMI          }
                                     { WITH EGI          }

[                      {{identifier-3}         [LINE ]}]
[                      {{integer     }         [LINES]}]
[ BEFORE  ADVANCING    {                              }]
[ AFTER                {{mnemonic-name}               }]
[                      {{PAGE         }               }]
```

```
                                                          (TO        ) (identifier-3  )
SET    {identifier-2 }  {[identifier-2]  }    ...         {UP BY     } {index-name-3  }
       {index-name-1 }  {[index-name-2]  }                (DOWN BY   ) (integer-1     )

                         {ASCENDING }
SORT   file-name-1    ON {DESCENDING}  KEY data-name-1   [, data-name-2] ...

                   [     {ASCENDING }                                   ]
                   [  ON {DESCENDING}  KEY data-name-3   [, data-name-4]] ... ...

       [COLLATING SEQUENCE IS alphabet-name]

                                                      [ {THROUGH}                 ] }
      { INPUT PROCEDURE IS section name-1             [ {THRU   }  section-name-2 ] }
      {
      { USING file-name-2   ,[file-name-3] ...

                                                      [ {THROUGH}                 ] }
      { OUTPUT PROCEDURE IS section-name-3            [ {THRU   }  section-name-4 ] }
      {
      { GIVING file-name-4

                         [      IS EQUAL =          )                               ]
                         [     (IS =                )                               ]
START file-name          [ KEY (IS GREATER than      (            data-name         ]
                         [     (IS >                )                               ]
                         [     (IS NOT LESS THAN    )                               ]
                         [      IS NOT <            )                               ]

       [;INVALID KEY imperative-statement]

      {RUN    }
STOP  {literal}

                                                                  (identifier-3 )
        {identifier-1}[  {identifier-1}] ...DELIMITED BY          {literal-3    }
STRING  {literal-1   }[, {literal-2   }]                          (SIZE         )

                                                                  (identifier-6 )  ]
    [  {identifier-4}[  {identifier-5}] ...DELIMITED BY           {literal-6    }  ]
    [  {literal-4   }[, {literal-5   }]                           (SIZE         )  ]

     INTO identifier-7  [WITH POINTER  identifier-8]

     [, ON OVERFLOW imperative-statement

            {identifier-1} [, {identifier-2} ]
SUBTRACT    {literal-1   } [  {literal-2   } ] ...  FROM identifier-3     [ROUNDED]

             [,identifier-n [ROUNDED]

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

```
         {identifier-1}  [ , identifier-2 ]
SUBTRACT {literal-1   }  [ , literal-2    ]  ...FROM identifier-m

         [ROUNDED]
    ┌──────────────────────────────────────────────────────────────────────┐
    │ [, identifier-n [ROUNDED]] [;ON SIZE ERROR imperative statement]     │
    └──────────────────────────────────────────────────────────────────────┘
UNSTRING identifier-1

    DELIMITED BY [ALL] {identifier-2} [ , OR [ALL] {identifier-3} ]...
                       {literal-1   } [            {literal-2   } ]

    INTO identifier-4    [, DELIMITER IN identifier-5] [, COUNT IN identifier-6]

[, identifier-7  [, DELIMITER IN identifier-8] [, COUNT IN identifier-9]] ...

    [WITH POINTER identifier-10] [TALLYING IN Identifier-11]  ...

    [; ON OVERFLOW imperative-statement]


                                                             {file-name}
                                                             {INPUT    }
                      {EXCEPTION}  PROCEDURE ON              {OUTPUT   }
USE AFTER STANDARD    {ERROR    }                            {I-O      }
                                                             {EXTEND   }

                       {cd-name-1                           }
                       {[ALL REFERENCES OF] identifier-1    }
USE FOR DEBUGGING ON   {file-name-1                         }
                       {procedure-name-1                    }
                       {ALL PROCEDURES                      }
  [                                              ]
  [         cd-name-2                            ]
  [         [ALL REFERENCES OF] identifier-2     ]
  [  ,      file-name-2                          ]
  [         procedure-name-2                     ]
  [         ALL PROCEDURES                       ]
WRITE record-name [ FROM identifier-1 ]
  [                         {{identifier-2} [LINE ]} ]
  [ {BEFORE}                {{integer     } [LINES]} ]
  [ {AFTER }    ADVANCING   {                      } ]
  [                         {{mnemonic-name}       } ]
  [                         {{PAGE         }       } ]
  [ ; AT    {END-OF-PAGE}        imperative-statement ]
  [         {EOP        }                             ]

WRITE record-name [ FROM identifier ] [;INVALID KEY imperative-statement]
```

GENERAL FORM FOR COPY STATEMENT

```
COPY     "text-name".
```

GENERAL FORMAT FOR CONDITIONS

Relation condition:

```
                             IS [NOT] GREATER THAN
{ identifier-1            }  IS [NOT] LESS THAN      { identifier-2            }
{ literal-1               }  IS [NOT] EQUAL to       { literal-2               }
{ arithmetic-expression-1 }  IS [NOT]                { arithmetic-expression-2 }
{ index-name-1            }  IS [NOT]                { index-name-2            }
                             IS [NOT] =
```

Class Condition:

```
identifier IS [NOT]     { NUMBERIC   }
                        { ALPHABETIC }
```

Sign Condition:

```
arithmetic-expression IS [NOT]     { POSITIVE }
                                   { NEGATIVE }
                                   { ZERO     }
```

Condition-name Condition:

condition-name

Switch-status Condition:

condition-name

Negated Simple Condition:

NOT simple-condition

Combined Condition:

```
condition  { {AND}   condition  }  ...
           { {OR }              }
```

Abbreviated Combined Relation Condition:

```
relation-condition  { {AND} NOT  [relational-operator]     object }...
                    { {OR }                                       }
```

MISCELLANEOUS FORMATS

QUALIFICATION:

```
{ data-name-1    }  [ { OF }            data-name-2 ] ...
{ condition-name }  [ { IN }                        ]

  paragraph-name    [ { OF }       section-name  ]
                    [ { IN }                     ]

  text-name  [ { OF }  library-name  ]
             [ { IN }                ]
```

SUBSCRIPTING:

```
{ data-name      }      subscript-1      [ , subscript-2  ,  subscript-3    ) ]
{ condition-name }
```

INDEXING:

```
{ data-name      }  (  { index-name-1 [ { PLUS  } literal-2 ] }
{ condition-name }     { literal-1    [ { MINUS }           ] }

  [ , { index-name-2 [ { PLUS  } literal-4 ] } [ , { index-name-3 [ { PLUS  } literal-6 ] } ] ]
  [   { literal-3    [ { MINUS }           ] } [   { literal-5    [ { MINUS }           ] } ] ]
```

IDENTIFIER: FORMAT 2

```
data-name-1  { OF }   data-name-2    [ (  { index-name-1 } [ { PLUS  } literal-2 ]
             { IN }                  [    { literal-1    } [ { MINUS }           ]

    [ , { index-name-2 } [ literal -4 ]     [ , { index-name-3 } [ literal-6 ] ] ) ]
    [   { literal-3    } [            ]     [   { literal-5    } [           ] ]   ]
```

APPENDIX G

SUMMARY OF EXTENSIONS TO ANSI COBOL

L/II COBOL provides extensions for interactive working, program control of files, text file handling, and rapid development and testing. These facilities are summarized below.

SCREEN FORMATTING AND DATA ENTRY

THE ACCEPT STATEMENT

An additional format for the ACCEPT statement is provided as follows:

Format

$$\underline{\text{ACCEPT}}\ \ \text{dataname-1}\ \left[\underline{\text{AT}}\ \left\{\begin{array}{l}\text{dataname-2}\\\text{literal-1}\end{array}\right\}\right]\ \ \underline{\text{FROM}}\ \underline{\text{CRT}}$$

data-name-2

allows the start of screen to be changed dynamically. It refers to a PIC 9999 field where the most significant 99 is a line count 1-34 and the least significant 99 is a character position 1-80.

data-name-1

refers to a record, group or elementary item but may not be subscripted.

literal-1

is an alphanumeric literal

NOTE

See Section 3 for description. See also Appendix H for Environment Division changes.

THE DISPLAY STATEMENT

An additional format for the DISPLAY statement is provided as follows:

Format

```
DISPLAY  { data-name-1 }  [ AT  { dataname-2 } ]  UPON   { CRT       }
         { literal-3   }  [     { literal-1  } ]         { CRT-UNDER }
```

literal-3

is an alphanumeric literal

dataname-1

refers to a record, group or elementary item but may not be subscripted

dataname-2

defines the left-most position on the screen. It refers to a PIC 9999 field where the most significant 99 is a line count 1-25 and the least significant 99 is a character position 1-80.

> NOTE
>
> See Section 3 for description.

DISK FILES

Two extensions are offered by L/II COBOL file processing. These are as follows:

1. Line sequential files
2. Run time input of filenames

LINE SEQUENTIAL FILES

When LINE SEQUENTIAL ORGANIZATION is specified in the FILE CONTROL paragraph ORGANIZATION IS entry, the file is treated as consisting of variable length records separated by the line delimiter characters. Trailing spaces in output records are replaced by a new line character.

RUN-TIME INPUT OF FILENAMES

The ASSIGNed name in the SELECT statement for a file is processed on OPENing as follows:

When the INPUT or OUTPUT phrase is specified, execution of OPEN causes checking of the files names in accordance with the Operating System connections for opening on input or output file. The full Operating System features for file reallocation and device control are therefore available to the B 20 COBOL program.

LOWERCASE CHARACTERS

The full alphabetic lowercase a to z is available in B 20 COBOL. Reserved and user word characters are read as their uppercase equivalents (A to Z).

HEXADECIMAL VALUES

Hexadecimal binary values can be attributed to non-numeric literals in B 20 COBOL by expressing them as X "xx", where x is a hexadecimal character in the set 0-9, A-F; xx can be repeated up to 128 times, but the number of hexadecimal digits must be even.

INTERACTIVE DEBUGGING

There is a Run-Time Debug Package to provide break-point facilities in the user's program. Programs may be run from the start until a specified break-point is reached when control is passed back to the user. At this point, data areas may be inspected or changed.

The debug package is entered as an option by the user and the user program is then tested line by line, paragraph by paragraph, and so on as required. The commands to the package can reference procedure statements and data areas by means of a 4 digit hexadecimal code output by the compiler against each line of the compilation listing. Powerful macros of commands can be used to give very sophisticated debugging facilities. The precise details for using the package vary according to the host Operating System and are described in Appendix J.

APPENDIX H

SYSTEM DEPENDENT LANGUAGE FEATURES

This Appendix summarizes those parts of a COBOL program that need to be changed to run them as L/II COBOL programs and those parts that do not need changing specifically but are ignored by the L/II COBOL compiler when generating the object program.

MANDATORY CHANGES

ENVIRONMENT DIVISION

The only statements in the Environment Division that must be specialized for L/II COBOL are shown below:

Configuration Section

SPECIAL-NAMES. special names entry

special names entry must include the following:

CURSOR IS data-name-1

The CURSOR IS data-name-1 clause specifies the data-name which will contain the CRT cursor address as used by ACCEPT statements. Data-name-1 must be declared in the Working-Storage section as a 4 character item. The interpretation of the 4 characters is given in the ACCEPT statement description.

Input-Output Section

File-names must be as described in the "File Management" section of the BTOS Operating System Manual.

STATEMENTS COMPILED AS DOCUMENTATION ONLY

COBOL programs not specifically written for compilation as L/II COBOL on microcomputers can still be compiled. Statements using features that are not available are treated as documentary only, and are not compiled. A summary of these features follows:

ENVIRONMENT DIVISION

I-O-Control Paragraph

The clauses that refer to a real time clock and magnetic tape in this paragraph are ignored by the compiler during compilation but do not cause compile time errors. These clauses are as follows:

$\underline{\text{END}}$ OF $\left\{\begin{array}{l}\underline{\text{REEL}}\\ \underline{\text{UNIT}}\end{array}\right\}$ OF file-name-2 (no magnetic tape)

integer-2 $\underline{\text{CLOCK}}$ $\underline{\text{UNITS}}$ (no clock)

DATA DIVISION

File Description Paragraph

The following complete statements in the file description are ignored by the compiler during compilation but do not cause compile time errors:

$\underline{\text{BLOCK}}$ CONTAINS integer-1 $\underline{\text{TO}}$ integer-2 $\left\{\begin{array}{l}\underline{\text{RECORDS}}\\ \underline{\text{CHARACTERS}}\end{array}\right\}$

$\underline{\text{CODE-SET}}$ IS alphabet-name

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

$\underline{\text{VALUE}}$ $\underline{\text{OF}}$ implementor-name-1 IS literal-1
[,implementor-name-2 IS literal-2] ...

PROCEDURE DIVISION

CLOSE Statement

The following phrases in the CLOSE statement are ignored by the compiler during compilation but do not cause compiler-time errors+

$\left\{\begin{array}{l}\underline{\text{REEL}}\\ \underline{\text{UNIT}}\end{array}\right\}$ (No magnetic tape)

APPENDIX I

LANGUAGE SPECIFICATIONS

B 20 COBOL is ANSI COBOL as specified in "American National Standard Programming Cobol" (ANSI X3.23 1974), with extensions and restrictions. The B 20 COBOL Implementation has been selected from both levels of ANSI COBOL. The following modules are fully implemented at Level 1:

- o Segmentation
- o Library
- o Debug

In addition, the following modules are fully implemented at Level 2:

- o Nucleus
- o Table Handling
- o Sequential Input and Output
- o Relative Input and Output
- o Indexed Input and Output
- o Inter-Program Communication
- o Sort/Merge

The full Level 2 syntax of the Communications module is accepted at this release but the Run-time System does not yet include the capability to execute it.

The appendix specifies the implementation of B 20 COBOL. The implementation of each of the ten (10) standard COBOL modules listed above is given under the following headings as applicable:

<u>Level 1 Implementation</u>
<u>Level 2 Implementation</u>
<u>B 20 COBOL Extensions</u>

Appendix F in this manual is a B 20 COBOL syntax summary.

NUCLEUS

Level One Implementation

Fully implemented to Level ONE.

Level Two Implementation

Fully implemented to Level Two.

L/II COBOL Extensions

1. Lower case letters a to z are read as upper case letters A to Z.

2. Hexadecimal binary values can be attributed to nonnumeric values by expressing literals as X"nn".

3. Reserved word SPACE can be used to clear the whole CRT screen.

4. COMPUTATIONAL-3 or COMP-3 can be specified in the USAGE clause to specify packed internal decimal storage, (BCD).

5. $\text{ACCEPT data-name-1} \left[\text{AT} \left\{\begin{array}{l}\text{data-name-2}\\ \text{literal-1}\end{array}\right\}\right] \text{ FROM CRT}$

   gives enhanced CRT input features.

6. $\text{DISPLAY} \left\{\begin{array}{l}\text{data-name-1}\\ \text{literal-1}\end{array}\right\} \left[\text{AT} \left\{\begin{array}{l}\text{data-name-2}\\ \text{literal-2}\end{array}\right\}\right] \text{UPON} \left\{\begin{array}{l}\text{CRT}\\ \text{CRT-UNDER}\end{array}\right\}$

   gives enhanced CRT ouput facilities.

7. 'CURSOR IS data-name' can be specified in SPECIAL-NAMES and 'data-name' in WORKING-STORAGE section to specify CRT cursor address for ACCEPT statements.

8. The function names SYSIN, SYSOUT and TAB can be assigned to user specified mnemonic-names in the SPECIAL NAMES paragraph. SYSIN and SYSOUT are equivalent to ACCEPT and DISPLAY from and to CONSOLE, respectively. TAB is used with the WRITE statement to cause the printer to throw a page. A directive is available in the compiler command line to alter these function names if they are already used in your COBOL program for other purposes.

In addition, the following extensions are incorporated.

1. Redefinition of data names need not be the same length - the compiler reserves the largest area.

2. Level numbers need not be specified in sequence. Thus:
   01 - - - -
   03 - - - -
   02 - - - -
   will be valid - with 03 being treated as if it were 02.

3. Introduction of FILLER group items.

SEQUENTIAL, RELATIVE AND INDEXED I-O

Level One Implementation

Fully implemented to Level ONE.

Level Two Implementation

Fully implemented to Level Two.

L/II COBOL Extensions

1. Run Time allocation of file-names.

2. LINE SEQUENTIAL is an additional file type.

3. All File Description (FD) clauses are optional.

4. Tabbing is available, specified by TAB in the WRITE statement. (See item 8 under NUCLEUS L/II COBOL Extensions above.)

TABLE HANDLING

Level One Implementation

Fully implemented to Level One.

Level Two Implementation

Fully implemented to Level Two.

SEGMENTATION

Level One Implementation

Fully implemented to Level One.

LIBRARY

Level One Implementation

Fully implemented to Level One.

DEBUG

Level One Implementation

Fully implemented to Level 1 plus an additional Interactive Run-Time Debug package.

L/II COBOL Extensions

A powerful Run-Time Debug package is available. (See Appendix J.)

INTER-PROGRAM COMMUNICATION

Level One Implementation

Fully implemented to Level One.

Level Two Implementation

Fully implemented to Level Two.

SORT-MERGE

Level One Implementation

Fully implemented to Level One.

Level Two Implementation

Fully implemented to Level Two.

APPENDIX J

USING COBOL

INSTALLING COBOL

Cobol level 4.0 (style ID B20CB4) is packaged so that it can be installed onto a hard disk. It also contains the necessary files so that it can be used on a Dual Floppy Standalone system.

The package consists of two disks. Disk One enables you to run Cobol on a 4.0 Operating system (hard disk or Dual Floppy Standalone). Disk Two is for customizing capabilities on a 4.0 or later Operating system (Hard disk).

CONTENTS OF THE B 20 COBOL DISKETTES

The following files are present on all 5-1/4" and 8" diskettes:

```
<Sys>CrashDump.sys
<Sys>FileHeaders.sys
<Sys>Mfd.sys
<Sys>sysImage.sys
<Sys>DiagTest.sys      (only on 8" diskettes)
<Sys>Log.sys
<Sys>BootExt.sys       (only on 8" diskettes)
<Sys>BadBlk.sys
```

The distribution diskettes contain the following files:

Language Disk 1, B 20 Cobol

```
<Sys>Cobol.erR
<Sys>CobolForms.edf
<Sys>Cobol.I02
<Sys>fdsys.version
<Sys>Sys.cmds
<Sys>Cobol.I03
<Sys>Cobol.aDS
<Sys>Cobol.isR
<Sys>Cobol.I09
<Sys>Install.sub
<Sys>Cobol
<Sys>Cobol.dyn
<Sys>Cobol.dbg
<Sys>Cobol.I01
<sys>Cobol.run
<sys>XEInstall.sub
```

Language Disk 2, B 20 Cobol

```
<Burroughs>Cobol.Res.run
<Burroughs>CobolLib.mod
<Burroughs>CobolGen.asm
<Burroughs>LinkCobol.sub
<Burroughs>Cobol.fls
<Burroughs>CobolRes.fls
<Burroughs>Cobol.lib
<Burroughs>LinkCobolRes.sub
```

COBOL MEMORY REQUIREMENTS

The B 20 Cobol Compiler distribution diskettes contain the swapping version of the runtime. This version uses the Burroughs Virtual Code Management facility. Selected runtime procedures remain on the disk as overlays until they are required.

The swapping version reduces the memory requirements of the runtime with the cost of a small performance degradation.

The amount of memory required to run a Cobol program can be computed by adding the bytes required as specified below:

Swapping runtime	63,624 Bytes	
Cobol WORKING-STORAGE	47,000 Bytes	*
LINE SEQUENTIAL file	1,154 Bytes	
SEQUENTIAL or RELATIVE file	1,088 Bytes	
Each Cobol source statement	10 Bytes	

*The 47,000 bytes for working storage will vary depending on the configured Cobol run-time. To calculate after linking, type Cobol.Map. Subtract the first address for Data (located under Class heading) from the first address of Memory (located under Class heading). Add 1,000 bytes of overhead to the computation. Subtract from the 64K of data available. The result will be the available bytes for working storage. The packaged version of Cobol will contain approximately 30,000 bytes.

Note that if indexed files are used, additional memory is required to install Burroughs Multiuser ISAM. On a cluster system, this additional memory is required only at the master workstation.

Note that the combined data space of active Cobol modules is limited to 60K bytes.

See the B 20 Systems ISAM Reference Manual, form 1148723, for details on the memory requirements of Multiuser ISAM.

COBOL IMPLEMENTATION SPECIFICATIONS

The Indexed Input and Output module is implemented using B 20 Multiuser ISAM.

The Sort-Merge module is implemented using B 20 Sort/Merge.

COBOL LINE SEQUENTIAL files are implemented using the BTOS Sequential Access Method (SAM). The Organization Line Sequential clause is required to Write to files assigned to device "[Splb]".

COBOL SEQUENTIAL and RELATIVE files are implemented using the BTOS Direct Access Method (DAM).

HARD DISK INSTALLATION INSTRUCTIONS

Boot the system from the master or cluster where you want the installation of COBOL Compiler. The software is installed in the system files of the system directory.

- o Insert the COBOL Compiler diskette in floppy drive [f0].
- o Do not press the **RESET** button.
- o Enter the **SOFTWARE INSTALLATION** command on the command line and press **GO.**
- o Follow the instructions displayed on the screen.
- o When installation is complete, remove the distribution diskette and store it in a safe place.

XE520 INSTALLATION

Boot the cluster workstation you want to use for software installation from the XE520.

- o Power off all other cluster workstations.
- o Log onto user ADMIN.
- o Insert the COBOL Compiler diskette in floppy drive [f0].

  Do not press the **RESET** button.
- o Enter the **SUBMIT** Command on the command line and press **RETURN.** The following parameter appears on the screen:

```
SUBMIT
   File List      [f0]<sys>XEInstall.sub
```

  Press GO to invoke the **SUBMIT** command.
- o Follow the instructions displayed on the screen.
- o When installation is complete, remove the distribution diskette and store it in a safe place.

DUAL FLOPPY STANDALONE INSTALLATION INSTRUCTIONS

COBOL 5.0 can be used on B 26 Dual Floppy Standalone systems. Before creating, compiling, and running COBOL programs on the B 26 Dual Floppy, duplicate the COBOL disk by using the following procedures.

1. Place the system disk of the 4.0 or later B 26 Dual Floppy into disk drive [f0].

2. Execute the **FLOPPY COPY** command as follows:

FLOPPY COPY	
[Number of copies]	
[Overwrite OK?]	
[Dual floppy?]	Yes
[Suppress verify?]	
[Device names(s)]	
[Device password(s)]	

3. Remove the system disk.

4. Place the COBOL compiler disk in [f0] and the COBOL source file in [f1].

5. Press the **GO** key.

6. Execute the **COBOL** command as follows:

COBOL	
Source file	[f1]<YourDir>YourFile.Bas
[Intermediate file]	[f1]<YourDir>YourFile.Int
[listfile]	[f1]<yourDir>Yourfile.Lst
•	
•	
•	
[Animate?]	

This compilation creates a list file called YourFile.Lst and an intermediate file called YourFile.int on the source disk. To change the destination or name of these files, explicitly state the path and their name. When the compilation is complete, the message

Please mount a system volume in [sys] and press GO to continue

displays. Put the system disk of the 4.0 or later B 26 Dual Floppy disks into drive [f0] and and press **GO.**

RUNNING A PROGRAM

Use the following procedure to run a program:

1. Place the COBOL compiler disk in [f0].

2. Place the source disk with YourFile.Int in disk drive [f1].

3. Execute the **CRUN** command as follows:

CRUN FILE	
Intermediate file	[f1]<YourDir>Your File.Run
[Parameters]	
[Switches]	
[Enable COBOL debugger?]	
[Enable ANSI debug switch?]	
[Prompt on return?]	

When execution is complete, the message

Please mount a system volume in [sys] and press GO to continue

displays. Put the system disk of the 4.0 B 26 Dual Floppy disks into drive [f0] and press **GO.**

<u>Helpful Hints For Dual Floppy Standalone Systems</u>

You should have a second disk containing programs to compile and run. Enter COBOL on the command line, then press the RETURN key. Fill in the name of the source program on the first line, referencing [F1] and the directory containing the program. On the second line the output file is listed. It always ends with ".int" and the prefix should be [f1]<directory>programname. If no name is given the prefix will default to [F1]<directory>first word of the program name. Press the GO key to compile the program. At the end of the compile the following message will be displayed: "Please mount a system volume in [Sys] and press GO to continue". Remove Disk 1 from [F0], insert the 4.0 system disk; then press GO. When the command line is displayed, remove the disk from [F0] and insert Disk 1.

the program. On the second line the output file is listed. It always ends with ".int" and the prefix should be [F1]<directory>programname. If no name is given the prefix will default to [F1]<directory>first word of the program name. Press the GO key to compile the program. At the end of the compile the following message will be displayed: "Please mount a system volume in [Sys] and press GO to continue". Remove Disk 1 from [F0], insert the 4.0/IE system disk; then press GO. When the command line is displayed, remove the disk from [F0] and insert Disk 1.

To work around this situation, copy the Exec.run file onto Disk 1 so that you can return to the system after the compile is finished and proceed normally.

When the program is compiled and ready for testing, enter CRUN on the command line; then press the RETURN key. On the first line fill in the *.int file generated from the compile with the prefix [F1]<directory>, then press the GO key. Unless Exec.run was copied onto Disk 1, at the end of the program the "Mount a system volume" message will be displayed, proceed as explained above.

To use a parallel printer the file Lptconfig.sys must be loaded on Disk 1. To use a serial printer the file Ptrbconfig.sys must be loaded on Disk 1. Both files can be copied from B26SF4 4.0 System disk.

For ISAM programs an additional step is needed. Insert Disk 1 of B26IF4 ISAM 4.0, enter ISAM INSTALL, then press the GO key to install ISAM in the system. Then proceed as stated above. For ISAM programs in which data sets are created, they would normally default to [F0]. You should specify in the programs that the data sets be written to [F1]. Note that the ISAM parameters used depend on the memory available and the program requirements.

COBOL CONFIGURATION

Cobol (Cobol.run) has been configured with the following BTOS interfaces. If you want to configure Cobol with different or additional BTOS interfaces, refer to CUSTOMIZING COBOL in this document.

Forms
Video Access Method (VAM)
Video Display Management (VDM)
Memory Management
Task Management
Openfile/Closefile
File Management
Keyboard Management
Timer Management
ISAM

USING A B9251 PARALLEL OR AP1300 SERIES SERIAL PRINTER WITH COBOL

The following procedure insures that a carriage return/form feed is inserted whenever a form feed is encountered in the text to be printed. This will correct previous problems when using these printers from Cobol.

1. Create an EDITOR file containing the following information:
   0C = 0D,0C

2. Invoke the command: MAKE TRANSLATION FILE.

```
Make Translation File
    Source file name          "file-name from step 1"
    Translation file name     "any-name.txl"          (Press GO)
```

3. Invoke the command CREATE CONFIGURATION FILE.

```
Create Configuration File
    Configuration file name
    Device type (comm. parallel, lpt, or serial ptr)
```

   Type in the information appropriate for the printer(s) that will be used with Cobol, such as "Splconfig.sys" for spooled parallel printing.

   The last parameter ([Translation file (default=none)]) of the subform asks for the translation file name. Enter the name of the translation file you created with the MAKE TRANSLATION FILE command.

4. This procedure must be followed for every configuration file associated with the AP1300 series and B9251 printers.

5. Reboot the system.

6. First Install Queue Manager and Install Spooler if [SPL] or [SPLB] is being used.

CUSTOMIZING COBOL ON HARD DISK SYSTEMS

The files located on the Cobol diskettes allow you to configure the Cobol runtime. This is accomplished when the Cobol.run file is reconfigured with non-Cobol procedures such as Forms, ISAM, Memory Management, etc. To create a new Cobol.run file, perform the following steps:

To build a new Cobol.run to run on a 4.0 Operating system:

IMPORTANT - Customizing of the Dual Floppy Standalone system Cobol run file is not supported.

Copy the files on Disk 2 in directory [F0]<Burroughs> to the desired directory (for this example, Cobol is the directory).

```
Copy
  File from          [F0]<Burroughs>*
  File to            [Sys]<Cobol>*
  [Overwrite OK?]    y
  [Confirm each?]
```

Press GO

The directory Cobol now contains the files:

```
LinkCobolRes.sub
LinkCobol.sub
Cobol.fls
CobolRes.fls
CobolLib.mod
CobolGen.asm
Cobol.Lib
CobolRes.Run
```

Be sure that 4.0 Assembler.run, 4.0 Linker.run, 4.0 Ctos.lib, 4.0 Forms.lib, 4.0 Isam.lib, and 4.0 SortMerge.lib have been copied to the SYS directory. If this has not been done, copy Assembler.run, Linker.run, Forms.lib, Isam.lib, and SortMerge.lib from the language development disks and Ctos.lib from the editor disk (4.0 Operating system package) to the SYS directory.

Assemble Cobolgen.asm: Burroughs creates Cobol.run with certain capabilities. You can create Cobol.run to contain more or less capabilities. This example describes the procedures you would take to create the packaged 4.0 Swp Cobol.run.

```
Assemble
  Source file            <COBOL>Cobolgen.asm
```

Press GO

The program will ask you to choose the capabilities which the Cobol.run will contain. To answer Yes, type Y and then press RETURN. To answer No, press RETURN.

To create the packaged Cobol.run, answer Yes to the following options:

Forms	Task Management
Openfile/Closefile	File Management
Video Access Method	Keyboard Management
Video Display Management	Timer Management
Memory Management	ISAM

The file CobolLib.mod has the following files:

```
<sys>Isam.lib
<sys>Forms.lib
<sys>SortMerge.lib
```

If you choose other options when assembling Cobolgen.asm, for example, Graphics.lib, you should edit CobolLib.mod and add a file <sys>Graphics.lib.

Submit the file LinkCobol.sub

```
 Submit
    LinkCobol.sub
```

To create a swapping Cobol.run, press GO. To create a resident Cobol.run, submit the file LinkCobolRes.sub.

In general, the swapping version of the Cobol.run is preferable to the resident version, because it requires less memory, with only a small performance degradation.

After pressing GO, the following Link command will be displayed with the following information filled in. Since the packaged version is a swapping one, the example fields contain information necessary to build a swapping version of Cobol.run. LinkCobolRes.sub creates a CobolRes.Run.

```
Link
  Object Module               Cobol.Lib(12)Cobol.Gen.O...
  Run file                    Cobol.run
  [List file]                 Cobol.map
  [Public?]
  [Line numbers?]
  [Stack Size]
  [Max memory array size]     47000
  [Min memory array size]     6000
  [System build?]
  [Version]                   '4.0 Swp'
  [Libraries]                 [Sys]<Sys>Isam.lib .....
  [Disk allocation?]
  [Symbol file]               Cobol.sym
```

The cursor will be positioned in the [symbol file] field. Remember that these libraries should be taken from the 4.0 Operating System language development disk. Ctos.lib is automatically called. Press GO.

The resident version compiles without any errors. Copy Cobol.run to <Sys>Cobol.run. If you submit LinkCobolRes.sub, copy CobolRes.run to <sys>Cobol.Run.

USING SamGenAll WITH COBOL

The Cobol.run file is configured with the following byte streams:

Spooler	[SPL] and [SPLB]
Video	[VID]
Keyboard	[KBD]
Null	[NULL]
Parallel Printer	[LPT]

In order to use the Communication [COMM] and Serial Line Printer [PTRB] byte streams, Cobol must be configured as follows:

1. Login to [Sys]<Sys> and copy the following files from the language development disk.

   SamGen.asm   SamGenAll.asm   SamGen.mdf

2. Assemble SamGenAll.asm

```
Command ASSEMBLE
Assemble
        Source files                    SamGenAll.asm
        [Errors only?]
        [GenOnly, NoGen, or Gen]
        [Object file]                   SamGen.obj  (Press GO)
        [List file]
        [List on pass 1?]
```

3. Invoke the Librarian to add to object module samGen.obj into CTOS.lib.

```
Command LIBRARIAN
Librarian
        Library file                    BTOS.lib
        [Files to add]                  SamGen.obj  (Press GO)
        [Modules to delete]
        [Modules to extract]
        [Cross-reference file]
        [suppress confirmation?]
```

   the message "SamGen already exists. Replace? (Press GO to confirm, CANCEL to deny)" will be displayed. Press GO. Otherwise, go to step 4.

4. Now proceed with the normal configuration of Cobol for use with non-Cobol procedures, beginning with Assembling CobolGen.asm. Be sure to say yes to the question "Are you calling the Sequential Access Method (y or n)?" if SAM is to be accessed by way of CALLs.

The following program will transmit a COMM byte stream out of the [COMM]B port. Use a cluster communications cable between the [COMM]B ports on the two B 20 systems.

```
Environment Division.
Input-Output Section.
File-Control.
    Select Xmt-file Assign to "[Comm]B"
    Organization Line Sequential.
File Section.
FD  Xmt-file.
01    Tokens              Pic X(14).
Procedure division.
    Open Output xmt-file.
    Move "12345678901234" to Tokens.
    Write Tokens.
    Move "ABCDEFGHIJKLMN" to Tokens.
    Write Tokens.
    Close Xmt-file.
    Stop run.
```

To test the new Cobol.run with the above source:

Create the configuration file "COMMBConfig.sys" using the default parameters on the transmitting and receiving B 20 systems. A cluster communications cable should be used. Perform a COPY from [COMM]B to [VID] on the receiving B 20. The receiving B 20 should display the characters shown in the program.

INVOKING THE COBOL COMPILER

To invoke the COBOL compiler from the Executive, type COBOL in the command field of the command form. The form illustrated below then appears.

```
COBOL
    Source file
    [Intermediate file]
    [List file]
    [List errors only?]
    [List COPY files?]
    [Flagging level (Low, L-I, H-I, High, Ext)?]
    [Suppress flagging?]
    [Suppress intermediate code?]
    [Suppress listing?]
    [Suppress location addresses?]
    [Suppress page headers?]
    [Suppress error echo?]
    [Resequnce lines?]
    [Lines per page?]
    [Animate?]
```

Field Descriptions

Source file

is the name of a COBOL source file to be compiled.

[Intermediate file]

is the file in which the compiler writes intermediate code. The default is the file name constructed by replacing the extension (suffix beginning with".") of the source file name with ".int".

[List file]

is the file in which the compiler writes the listing. The default is the file name that is constructed by replacing the extension (suffix beginning with ".") of the source file name with ".lst."

[List errors only?]

is Yes or No (the default). If Yes, the compiler lists only lines containing errors. If no, the compiler produces a full listing.

[List COPY files?]

is Yes or No (the default). If Yes, the compiler lists files specified by the COBOL COPY verb and includes the names of any COPY files that are open in and page headers. If no, the compiler suppresses the listing of COPY files.

[Flagging level (Low, L-I, H-I, High, Ext)?]

is Low, L-I, H-I, High, Ext. Validation flags are extra lines in the listing that indicate the level of a COBOL source statement.

Low produces validation flags for all features higher than the Low Level of compiler certification of the General Services Administration (GSA).

L-I produces validation flags for all features higher than the Low-Intermediate Level of compiler certification of GSA.

H-I produces validation flags for all features higher than the High-Intermediate Level of compiler certification of GSA.

High produces validation flags for all features higher than the High Level of compiler certification of GSA.

Ext produces validation flags for only the Level II COBOL extensions to standard COBOL as it is specified in the ANSI COBOL Standard X 3.23 1974.

[Suppress flagging?]

is Yes or No (the default). If Yes, the compiler disregards any response that was given in the previous field.

[Suppress intermediate code?]

is Yes or No (the default). If Yes, the compiler does not generate intermediate code. The compiler, in effect, only checks syntax.

[Suppress listing?]

is Yes or No (the default). If Yes, the compiler does not produce a listing. Use this for fast compilation of clean programs.

[Suppress location addresses?]

is Yes or No (the default). If Yes, the compiler does not include in the listing 4-digit location addresses for each source line. These location addresses are needed if you use the Level II COBOL Debugger.

[Suppress page headers?]

is Yes or No (the default). If Yes, the compiler does not put form feeds and page headers in the listing.

[Suppress error echo?]

is Yes or No (the default). If Yes, the compiler suppresses error line echoing on the video display.

[Resequence lines?]

is Yes or No (the default). If Yes, the compiler includes new sequence numbers in the listing, replacing those in the source. The compiler generates sequence numbers in columns 1 - 6 in numerical order from 000010 in increments of 10.

You can resequence a COBOL source by answering Yes to [Reseqeunce lines?], [Suppress location addresses?], and [Suppress page headers?]. You can use the listing produced as COBOL source after you use the Editor to remove the compilation statistics line from the end of the listing.

[Lines per page?]

is the number of lines on a listing page. The minimum is 5. The default is 60.

[Animate?]

is a COBOL-oriented debugger that allows you to debug a program by interacting directly with the COBOL source while the program is executing.

Compiler Error Messages

If an error is encountered during compilation, the compiler prints an error report on the video display and in the listing. The format of this report is:

```
nnnnn<invalid statement>
**mmm*****************
**      *<error message>
```

nnnnn is the sequence number of the erroneous line.

mmm is the error number.

<error message>is text describing the error.

The asterisks following mmm end at the location where the compiler detected the error. Often, this location is one or two words beyond the true location of the error.

In the example below, the asterisks end at B, which is one work beyond the location of the erroneous reserved word TOO.

```
031900      MOVE A TOO B.
**118******************
**      *Reserved word missing or incorrectly used
```

Compilation Statistics

At the end of compilation, the compiler prints compilation statistics on the video display and in the listing file. The format of this line is:

```
Errors=n  Data=n  Code=n  Dict-m:n/p  GSA flags=n
```

- Errors is the number of errors detected.
- Data is the size (in bytes) of the program's data area.
- Code is the size (in bytes) of the program's intermediate code area.
- Dict is m:n/p where:

    m is the number of bytes used in the data dictionary;

    n is the number of bytes remaining in the data dictionary;

    and p is the total number of bytes in the data dictionary.

- GSA flags is the number of validation flags or "off".

Examples

1. This example is a compilation of the COBOL source file Pi.cbl. The listing and intermediate code files are Pi.lst and Pi.int. A full listing is produced.

```
Command COBOL RETURN
COBOL
     Source file                                      Pi.cbl GO
     [Intermediate file]
     [List file]
     [List errors only?]
     [List COPY files?]
     [Flagging level (Low, L-I, H-I, High, Ext)?]
     [Suppress flagging?]
     [Suppress intermediate code?]
     [Suppress listing?]
     [Suppress location addresses?]
     [Suppress page headers?]
     [Suppress error echo?]
     [Resequence lines?]
     [Lines per page?]
     [Animate?]
```

2. This example demonstrates how to generate new sequence numbers for Pi.cbl. Intermediate code generation is suppressed. The new source file is named NewPi.cbl.

```
Command COBOL RETURN
COBOL
     Source file                                      Pi.cbl GO
     [Intermediate file]
     [List file]                                      New Pi.cbl
     [List errors only?]
     [List COPY files?]
     [Flagging level (Low, L-I, H-I, High, Ext)?]
     [Suppress flagging?]
     [Suppress intermediate code?]                    Y
     [Suppress listing?]
     [Suppress location addresses?]                   Y
     [Suppress page headers?]                         Y
     [Suppress error echo?]
     [Resequence lines?]                              Y GO
     [Lines per page?]
     [Animate?]
```

RUNNING A COBOL PROGRAM

After a COBOL program is compiled, invoke the COBOL run-time system to execute the intermediate code produced by the compilation.

The run-time system is invoked either by using the CRun command or by using your own custom command, created by the Executive's 'New Command' facility.

The CRun Command

To run your COBOL program, type CRun in the command field of the command form. The form illustrated below then appears.

```
CRun
    Intermediate file
    [Parameters]
    [Switches]
    [Enable COBOL Debugger?]
    [Enable ANSI debug switch?]
    [Prompt on return?]
```

Field Descriptions

Intermediate file

is the name of the file that contains the intermediate code to be executed.

[Parameters]

are the invocation parameters to be passed to the COBOL Program. To read parameters from a COBOL program, open [KBD] as a LINE SEQUENTIAL file and read the first input record. You can also read parameters using the BTOS Parameter Management routines. (See the BTOS Operating System Manual for details.) Not responding causes no parameters to be passed.

[Switches]

are switches (up to 8) to turn on or off. With Level II COBOL, events can be controlled at run-time depending on the setting of programmable switches. See the SPECIAL NAMES paragraph in Section 3.

The first switch is +0, the second is +1, and so on through +7. Preceding the switch with a + (plus) sign turns the switch on; a - (minus) sign turns the switch off. The sign is required. To specify more than one switch, type a single quote character at the beginning and end of the sequence. Not responding causes no switches to be turned on.

[Enable COBOL Debugger?]

is Yes or No (the default). If Yes, the interactive COBOL Debugger is enabled. (See Section following entitled COBOL Debugger.)

[Enable ANSI debug switch?]

is Yes or No (the default). If Yes, the standard ANSI COBOL debug module is invoked. (See Section 11 above.)

[Prompt on return?]

is Yes or No (the default). If Yes, a prompt message is printed on the video display after the COBOL program terminates but before the Executive is entered. You must respond to the prompt to continue. This option permits you to view the video display before returning to the Executive.

Sample Invocations using the CRun Command

1. This example shows how to run Pi.int, the intermediate code generated by the compilation of Pi.cbl.

```
Command CRun RETURN
CRun
     Intermediate file                                    Pi.int GO
     [Parameters]
     [Switches]
     [Enable COBOL Debugger?]
     [Enable ANSI debug switch?]
     [Prompt on return?]
```

2. This example shows how to run TEst.int with switches 1 and 4 turned on. Parameters are passed and the ANSI COBOL debug switch is enabled.

```
Command CRun RETURN
CRun
     Intermediate file                                    Test.int
     [Parameters]                                         test-100
     [Switches]                                           '+1 +4'
     [Enable COBOL Debugger?]
     [Enable ANSI debug switch?]                          Yes GO
     [Prompt on return?]
```

3. This example shows how to run Pi.int using the COBOL Debugger.

```
Command CRun RETURN
CRun
     Intermediate file                                    pi.int
     [Parameters]
     [Switches]
     [Enable COBOL Debugger?]                             y GO
     [Enable ANSI debug switch?]
     [Prompt on return?]
```

Using a Custom Command to Run a COBOL Program

To use a custom command to run a COBOL program, first create the command using the Executive's 'New Command' facility.

Fill inthe 'Command name' field with the name of the intermediate code file, minus the '.int' extension.

Fill in the 'Run file' field with '[sys]<sys>COBOL.run', which is the file containing the COBOL run-time system.

Fill in the 'Field names' field with field names of your choice. Data entered in these fields can be read by a COBOL program using LINE SEQUENTIAL input from [KBD]. This process is described in detail below.

After creating the command, move all intermediate code files that make up the program in the [sys]<sys>directory. If the program uses segmentation, do not forget to move the files containing independent segments and inter-segment reference information. These files have extensions '.Ixx' and '.ISR', where xx represents a segment number. If the program calls other COBOL modules, move these into [sys]<sys>as well.

in the example below, a command named 'Update' is created with two fields, 'Your name' and 'Update file'.

```
Command  New Command  RETURN
New Command
     Command name        'Update'
     Run file            [sys]<sys>COBOL.run
     Field names         'Your name' 'Update file' GO
     Description
     [Overwrite ok?]
```

After compiling Update.cbl and moving Update.int into the [sys]<sys>directory, the program can be invoked as follows.

```
Command  Update  RETURN
Update
     Your name           Donna
     Update file         Current GO
```

Reading the Fields of a Custom Command Form

A COBOL Program reads the data entered in the fields of a custom command form by reading sequential records of a file that is opened using filename [KBD], with LINE SEQUENTIAL organization, in INPUT mode.

The first record returned corresponds to the data entered in the first field. The second record corresponds to the second field, and so on. If the field contains no data, the corresponding record is empty, that is, it contains only spaces.

When all the fields have been read, subsequent read operations will take input from the keyboard.

The following sample program is invoked with the Update form described above. The program first reads the two fields of the form, then writes the data contained in the 'Your name' field into the file named by the 'Update file' field.

```
* Program Update.cbl

select form
     assign "[KBD]"
     organization line sequential.

select update-file
     assign update-name
     organization line sequential.

fd form.
01 form-buffer pic X(80).
fd update-file
01 update-buffer pic X(80).

procedure division.

* Read the two fields of the Update command form

     open input form.
     read form.
     move form-buffer to update-buffer.
     read form.
     move form-buffer to update-name.
     close form.

* Write data to the update-file

     open output update-file.
     write update-buffer.
     close update-file.
     stop run.
```

<u>Advanced Invocation Techniques for Debugging</u>

A COBOL program that is designed to be run from a custom command form can also be run using the CRun form. This is required to run such a program using the COBOL Debugger.

If you are using the COBOL Debugger, follow these steps:

1. Run the program using the CRun form. Leave the [Parameters] field empty. Fill in [Enable COBOL Debugger] with Y.

2. Once in the Debugger, use the G command to execute to a breakpoint in the program beyond the section that reads the fields.

3. Type in the data for each field, terminating each line with a RETURN.

4. When all the fields are entered, the breakpoint will be reached. Continue debugging.

The sample invocation below shows how to run Update.int using the COBOL Debugger. Address 004f corresponds to the statement 'open ouput update-file' in the program shown above.

```
Command CRun RETURN
CRun
     Intermediate file                        [sys]<sys>update.int
     [Parameters]
     [Switches]
     [Enable COBOL Debugger?]                 Y GO
     [Enable ANSI debug switch?]
     [Prompt on return?]

COBOL Debugger 6.2
?G004F
Donna RETURN
current RETURN
?
```

FILES REQUIRED FOR COMPILING AND RUNNING A COBOL PROGRAM

COBOL.run

is the COBOL run-time system. It is needed for compiling and running a COBOL program.

COBOL.ads

is needed for running a COBOL program that uses the extended ACCEPT or DISPLAY verbs.

COBOL.dbg

is needed for running a COBOL program with the COBOL Debugger enabled.

The following files constitute the COBOL compiler. They are needed only for compiling a COBOL program.

```
COBOL       COBOL.isr   COBOL.err
COBOL.i01   COBOL.i02   COBOL.i03   COBOL.i09
```

USING FILE- AND RECORD-LEVEL LOCKING WITH INDEXED FILES

COBOL allows access to Burroughs ISAM's powerful record- and file-level locking capabilities.

Locking provides secure and independently controlled file access for each user in a multi-user configuration. File and record locks permit exclusive access to file or a record within a file by one user.

Semantics of File- and Record-Level Locking

A lock regulates concurrent access to a file or record, thereby maintaining data integrity when more than one user accesses the same file. Locking protects a file or record in use by one user from updating operations of a concurrent user.

A file-level lock restricts access for all the records in a file, while a record-level lock only restricts access to a single record.

When a user holds a record lock for a record, other users cannot access the record. When a user holds a file lock for a file, other users may only access the file for reading if the holder is reading, otherwise concurrent users may not access the file.

The level of locking for each file is determined by a statement in the file control entry as described below.

Locking Modes

Locking modes are specified in the SELECT clause of the FILE-CONTROL paragraph. The three locking modes are:

EXCLUSIVE file-level locking

EXCLUSIVE locking mode prevents concurrent users from updating a locked file. If the user holding this file-level lock has the file open for INPUT, then other users are also allowed to open the file for INPUT. If, however, the user has the file open for OUTPUT or I-0, other users are not allowed to open the file at all.

AUTOMATIC record-level locking

AUTOMATIC locking mode automatically acquires a record-level lock for each record accessed by a user.

MANUAL record-level locking

MANUAL locking mode acquires a record-level lock for a record only if the statement causing the access specifically locks the record.

Specifying the Locking Mode

The default locking mode is AUTOMATIC for files with Indexed organization, whether in I-O, INPUT, or OUTPUT mode.

A record lock is thus acquired by the execution of the READ and START statements referencing the file, which is only released on next access to the file, i.e., at the end of execution of the next I/O statement that references the file. A record lock is also acquired by the execution of the WRITE, REWRITE, and DELETE statements and is released at the end of the current I/O statements and is released at the end of the current I/O statement.

To explicitly specify a locking mode for a file, use the LOCK MODE clause in the FILE-CONTROL paragraph as shown below. Note that extra syntax is not mandatory to invoke locking. If the LOCK MODE IS clause is left out of the FILE-CONTROL paragraph, then the default locking mode is AUTOMATIC.

```
FILE- CONTROL.

SELECT file-name

ASSIGN TO { external-filename-literal }  [ { , external-filename literal } ]
          { file-identifier           }  [ { , file-identifier          } ]

;ORGANIZATION IS  INDEXED

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

;RECORD KEY IS data-name-1

[ ;LOCK MODE IS    { AUTOMATIC  } ]
[                  { EXCLUSIVE  } ]
[                  { MANUAL     } ]

[;ALTERNATIVE RECORD KEY IS data-name-2 [WITH DUPLICATES] . . .

[;FILE STATUS IS data-name-3]
```

The full specification of the File Control entry is contained in Section 7. The only part of the File Control entry that is locking specific is the LOCK MODE clause.

When this clause is omitted, LOCK MODE IS AUTOMATIC is assumed.

When LOCK MODE IS EXCLUSIVE is specified, an exclusive file lock is acquired by the user when the file is opened. While a user holds an exclusive file lock, record-level locking does not occur within that file. If a file is opened in mode INPUT, then concurrent users may also open the file in mode INPUT; otherwise, other users may not open the file.

When LOCK MODE IS AUTOMATIC or LOCK MODE IS MANUAL is specified, record-level locking may occur for the file when it is opened.

If LOCK MODE IS AUTOMATIC is specified, and the file is open for I-O, a record-level lock is acquired by the execution of the READ, WRITE, REWRITE, START, and DELETE statements referencing the file.

If LOCK MODE IS MANUAL is specified, a record-level lock is acquired by the execution of a READ statement referencing the file only if the READ statement includes the WITH LOCK phrase.

Using the READ Statement with MANUAL Locking

When MANUAL locking mode is specified for a file, record-level locks are only acquired when a READ statement with a WITH LOCK phrase is executed.

When a lock is acquired, the record remains locked until the end of execution of the next I/O statement which references the same file. The syntax for the READ statement is shown below.

Format 1:

```
READ file-name [NEXT] RECORD [INTO identifier]

    [;WITH LOCK]

    [;AT END imperative-statement]
```

Format 2:

```
READ file-name RECORD [INTO identifier]

    [;KEY IS data-name]

    [;INVALID KEY imperative-statement]
```

The READ statement is the same as specified in the COBOL Manual for indexed files, except for the WITH LOCK phrase. When the WITH LOCK phrase is included, a record-level lock is acquired for files with MANUAL locking mode. For other details about the semantics of the READ statement, see Section 7.

Acquiring Record-Level Locks

The following summarizes the statements which cause record-level locking to occur:

Statement	Lock Mode	Opened	Lock Acquired
READ WITH LOCK	AUTOMATIC	I-O	YES
	AUTOMATIC	INPUT	NO
	MANUAL	I-O	YES
	MANUAL	INPUT	NO
READ (without WITH LOCK)	AUTOMATIC	I-O	YES
	AUTOMATIC	INPUT	NO
	MANUAL	I-O	NO
	MANUAL	INPUT	NO
START, DELETE, WRITE, REWRITE	AUTOMATIC/ MANUAL	I-O	YES
START	AUTOMATIC/ MANUAL	INPUT	NO
WRITE	AUTOMATIC/ MANUAL	OUTPUT	NO

Error Conditions While Using Locks

Status Key

When the run-time system detects that a lock on a record has already been acquired on behalf of a different user environment, it returns an ANSI error status key 1 value of '9' with 'D' in status key 2.

Current Record Pointer

If at any stage the record pointed to by a currently running program has been deleted by another program, the current program's record pointer will be updated to point to the next record on the file.

If, however, a lock is encountered on attempting to access a record (i.e., 'D' is returned as error status key 2), the current record pointer is unchanged.

Data Record Contents

If a lock is encountered on attempting to READ a record (i.e., 'D' is returned as error status key 2), the record contents will be undefined.

USING CALL IN A COBOL PROGRAM

COBOL provides a powerful CALL facility. Using the CALL verb, you can:

- o CALL another COBOL module using ANSI standard Inter-Program communication.
- o CALL special built-in procedures (such as PEEKB and POKEB) and that are provided by the COBOL run-time system.
- o CALL non-COBOL procedures that have been linked into the COBOL run-time system. You can directly CALL the BTOS Operating System and Burroughs software products, such as Forms.

Using CALL for Inter-Program Communication

You can arrange a COBOL application into a number of separately compiled programs that communicate and invoke each other with the COBOL CALL verb. Using this facility, you can write large and complex COBOL applications whose total size is not constrained by physical memory limitations.

The general format of the CALL verb is given in Section 12.

The COBOL Programs that constitute the application are known as the application suite. All program other than the main one should have a Linkage Section in the Data Division. The Linkage Section permits COBOL programs to communicate, that is, pass parameters.

All programs in the application suite must be compiled prior to executing the application. The COBOL program is run using the filename of the main program.

When the CALL verb is executed, the intermediate code of the called program is loaded into memory, assuming there is sufficient space. The ON OVERFLOW verb detects whether a CALL has failed due to lack of memory space. The CANCEL verb reclaims memory that was allocated to programs which are no longer in use.

Using CALL for Invoking Special Built-In Procedures

You can invoke several useful built-in procedures provided by the COBOL run-time system using the CALL verb.

The general format of the CALL verb is given in Section 12.

When CALLing special built-in procedures, the object of the CALL must be literal or an alphnumberic data item whose value is one of the number below.

GETCH	"259"
PUTCH	"258"
PEEKB	"261"
PEEKW	"276"
POKEB	"262"
POKEW	"277"
GETB	"263"
GETW	"278"
PUTB	"264"
PUTW	"279"
Define Escape	"275"

Each of the special built-in procedures is described below.

GETCH and PUTCH

Syntax:

```
CALL GETCH USING IN-VALB
CALL PUTCH USING OUT-VALB
```

where

GETCH IS PIC X(3) VALUE "259
PUTCH IS PIC X(3) VALUE "258"

OUT-VALB IS PIC X containing a byte to display on the video display.

IN-VALB IS PIC X and gets the byte to be read from the keyboard.

Action:

PUTCH displays the byte value in OUT-VALB on the video display.

GETCH reads a byte from the keyboard into IN-VALB without echoing it on the video display.

NOTE

Use GETCH to read data, such as passwords, that should not be displayed.

PEEKB and PEEKW

Syntax:

```
CALL PEEKB USING SEGMENT, OFFSET, DAT-VALB.
CALL PEEKW USING SEGMENT, OFFSET, DAT-VALW.
```

where

PEEKB	IS PIC X(3)	VALUE "261"
PEEKW	IS PIC X (3)	VALUE "276"
SEGMENT	IS PIC 9(5)	containing the segment number
OFFSET	IS PIC 9(5)	containing the offset in the segment
DAT-VALB	IS PIC X	and gets the data byte
DAT-VALW	IS PIC XX	and gets the data word

Action:

PEEKB and PEEKW return, respectively, the byte or word at the memory location specified by SEGMENT and OFFSET.

POKEB and POKEW

Syntax:

```
CALL POKEB USING SEGMENT, OFFSET, DAT-VALB.
CALL POKEW USING SEGMENT, OFFSET, DAT-VALW.
```

where

POKEB	IS PIC X(3)	VALUE "262"
POKEW	IS PIC X(3)	VALUE "277"
SEGMENT	IS PIC 9(5)	containing the segment number
OFFSET	IS PIC 9(5)	containing the offset in the segment
DAT-VALB	IS PIC X	containing the data byte to be stored
DAT-VALW	IS PIC XX	containing the data word to be stored

Action:

POKEB and POKEW store, respectively, a byte or word into the memory location specified by SEGMENT and OFFSET.

GETB AND GETW

Syntax:

```
CALL GETB USING PORT, VALUEB.
CALL GETW USING PORT, VALUEW.
```

where

GETB	IS PIC X(3)	VALUE "263"
GETW	IS PIC X(3)	VALUE "278"
PORT	IS PIC 9(5)	containing the port address
VALUEB	IS PIC X	and receives the input data byte
VALUEW	IS PIC XX	and receives the input data word

Action:

GETB and GETW read, respectively, the byte or word from the 8086 input port specified by PORT.

PUTB and PUTW

Syntax:

```
CALL PUTB USING PORT, VALUEB.
CALL PUTW USING PORT, VALUEW.
```

where

PUTB	IS PIC X(3)	VALUE "264"
PUTW	IS PIC X(3)	VALUE "279"
PORT	IS PIC 9(5)	containing the port address
VALUEB	IS PIC X	and contains the data byte to output
VALUEW	IS PIC XX	and contains the data word to output

Action:

PUTB and PUTW write, respectively, the byte or word from the 8086 output port specified by PORT.

DefineEscape

Syntax:

```
CALL DefineEscapte USING EscapeTable, EscapeKey
```

where

DefineEscape	IS PIC X(3)	VALUE "275"
EscapeTable	IS PIC X(n)	VALUE<escape keys>
n is in the range 1..256		
Escape-Key	IS PIC X	

Action:

DefineEscape defines a table of keys that serve as escape keys in the extended ACCEPT statement. DefineEscape also defines an elementary data item to receive the escape key that terminates an ACCEPT statement.

Up to 255 escape keys can be defined. The sequence of escape keys is terminated by a space character.

> NOTE
>
> The GO key is always an escape key. The space key cannot be defined as an escape key.
>
> A program can change the escape keys by repeated calls to DefineEscape.

Example:

Define f1 and f2 to be escape keys.

```
01 DefineEscape PIC X(3)VALUE "275".
01 EscapeTable PIC X(3)VALUE X"151620".
01 EscapeKey PIC X.
        •
        •
CALL DefineEscape USING EscapeTable, EscapeKey.
```

Using CALL For Invoking Non-COBOL Procedures

COBOL can directly CALL non-COBOL procedures that are linked into the COBOL run-time system. Using this facility, you can CALL the BTOS Operating System and Burroughs software products, such as Forms.

The CONFIGURING COBOL section below explains how to link non-COBOL procedures into the COBOL run-time system.

The general format of the CALL verb, including the syntax for passing parameters, is given in Section 12.

When using the CALL verb to invoke non-COBOL procedures, the object of the CALL is the nonnumeric literal that is the name of the procedure preceded by the ampersand (&) character.

For example, to CALL the BTOS Exit procedure write:

```
CALL "&Exit".
```

You can write the name of the procedure in either uppercase or lowercase.

If the non-COBOL procedure does not return a value, pass the number of parameters required by the procedure.

For example, the BTOS ErrorExit procedure requires one parameter, a status code, and does not return a value. To call this procedure write

```
CALL "&ErrorExit" USING ercExit.
```

If the procedure returns a value, pass an extra parameter at the beginning of the parameter list to receive the returned data.

For example, the BTOS CloseFile procedure requires one parameter, a file handle, and returns a status code. To call this procedure write

```
CALL "&CloseFile" USING erc, fh.
```

When passing a parameter, COBOL passes either its address or its value, depending upon the interface of the called procedure. The "Parameter Passing and Parameter Data Types" section below explains the types of parameters that COBOL can pass.

The COBOL run-time system provides several checks to detect incorrect procedure calls. These include calling an unknown procedure and calling a procedure with an incorrect number of parameters.

The non-COBOL Procedures "CLOSEALLFILES" and "CLOSEALLFILESLL" only work on a Resident generated COBOL.

Parameter Passing and Parameter Data Types

COBOL passes either parameter addresses or values, depending on the interface of the called procedure.

The run-time system gets information about procedure interfaces from the assembly language module CobolGen.asm. CobolGen.asm is discussed further in CONFIGURING COBOL below.

COBOL can pass bytes, byte strings, words, and double words (quads).

COBOL cannot correctly pass structures containing words and quads unless certain type conversion statements are added to the COBOL program. The "Passing Structures as Parameters" section below explains these conversion statements.

The data types that can be passed between a COBOL program and a non-COBOL Procedure are described below.

- Byte

  A byte is an 8-bit quantity, normally representing a character, an integer, or a boolean value.

  The COBOL PICTURE clauses that define a byte are

  ```
  PICTURE 9(2) USAGE IS COMP
            and
  PICTURE X(1)
  ```

  which defines a character.

  When using bytes as boolean values, 0 means false and 1 means true.

  The COBOL statements below show the definition and use of a byte parameter as a character (b) and as a boolean value (fOn).

  ```
  01 b PICTURE X(1) VALUE "A".
  01 fOn PICTURE 9(2) USAGE IS COMP VALUE 0.

  CALL "&WriteByte" USING erc, bswa, b.
  CALL "&SetKbdUnencodedMode" USING erc, fOn.
  ```

- Byte string

  A byte string is a contiguous sequence of bytes or characters.

  The COBOL PICTURE clause that defines a byte string is

  ```
  PICTURE X(n)
  ```

  where n is the length of the byte string.

  The COBOL statements below show the definition and use of byte string parameters (rgbFilename and rgbPassword).

  ```
  01 rgbFilename   PIC X(8)    VALUE "TestFile".
  01 rgbPassword   PIC X(5)    VALUE "xyzzy".

  CALL   "&OpenFile" USING erc, fh, rgbFilename,
         cbFilename, rgbPassword, cbPassword, mode-IO.
  ```

- Word

  A word is a 16-bit quantity, normally representing an integer.

  The COBOL PICTURE clauses that define a word are

  ```
  PICTURE 9(4) USAGE IS COMP
  ```

  which defines an integer and

  ```
  PICTURE X(2)
  ```

  which defines two contiguous bytes.

The COBOL statements below show the definition and use of word parameters (erc, fh, cbFilename, cbPassword, mode-IO).

```
01 erc           PIC 9(4)    USAGE IS COMP.
01 fh            PIC 9(4)    USAGE IS COMP.
01 rgbFilename   PIC X(8)    VALUE "TestFile".
01 cbFilename    PIC 9(4)    USAGE IS COMP VALUE 8.
01 rgbPassword   PIC X(5)    VALUE "xyzzy".
01 cbPassword    PIC 9(4)    USAGE IS COMP VALUE 5.
01 mode-IO       PIC X(2)    VALUE "mm".

CALL   "&OpenFile" USING erc, fh, rgbFilename,
       cbFilename, rgbPassword, cbPassword, mode-IO.
```

- Quad

  A quad is a 32-bit quantity, normally representing an 8086 address (pointer) or a logical file address (lfa).

  The COBOL PICTURE clause that defines a quad is

  ```
  PICTURE 9(9) USAGE IS COMP.
  ```

The COBOL statements below show the definition and use of a quad parameter (pSegment).

```
01 pSegment          PIC 9(9) USAGE IS COMP.
CALL "&AllocMemorySL" USING erc, cBytes, pSegment.
```

Passing Structures as Parameters

Some procedures require structures as parameters. The address of the structure is actually passed.

A structure is a contiguous group of data items. The individual data items are bytes, byte strings, words, and quads.

For example, the procedure RgParam. described in the Parameter Management section of the BTOS Operating System Manual, takes a structure as a parameter. The interface of RgParam is

```
RgParam (iParam,lcParam,pSdRet) : ErcType.
```

The final parameter, pSdRet, is a structure composed of a quad (pointer) followed by a word.

COBOL cannot correctly pass structures as parameters.

COBOL stores the bytes that make up words and quads in a different order than is expected by non-COBOL procedures. The COBOL run-time automatically reorders bytes for simple word and quad parameters. However, reordering does not occur for structures.

Two built-in non-COBOL procedures, ConvertWord and ConvertQuad, are provided so that you can explicitly reorder the word and quad components of a structure parameter.

The section Built-in non-COBOL Procedures below describes ConvertWord and ConvertQuad in detail.

If the word or quad contained in the structure is read by the non-COBOL procedure, CALL ConvertWord or ConvertQuad BEFORE the CALL to the non-COBOL procedure.

If the word or quad contained in the structure is written by the non-COBOL procedure, CALL ConvertWord or ConverQuad AFTER the CALL to the non-COBOL procedure.

In the case of RgParam, the sd structure is written by the procedures. The following example demonstrates a CALL to RgParam.

```
01 erc                  PIC 9(4)    COMP.
01 iParam               PIC 9(4)    COMP.
01 jParam               PIC 9(4)    COMP.
01 sd.
   03 pb                PIC 9(9)    COMP.
   03 cb                PIC 9(4)    COMP.

CALL "&RgParam" USING erg,iParam,jParam,sd.
CALL "&ConvertQuad" USING pb,pb.
CALL "&ConvertWord" USING cb,cb.
```

Passing Parameters to the Forms Run-time

COBOL correctly passes parameters, including structures, to all procedures in the Forms runtime if the parameter data definitions contained in the library file CobolForms.edf are used.

CobolForms.edf is installed with the standard COBOL software.

Insert the following statement in the WORKING-STORAGE section of the COBOL program that uses Forms.

```
COPY "CobolForms.edf".
```

The COPY statement causes the parameter data definitions in CobolForms.edf to be included in your COBOL program. These definitions are listed below.

```
01  InitState.
  02  init-ich              PIC 9(2)      COMP.
  02  filler                PIC 9(2)      COMP.
  02  filler                PIC X(6).

01  ExitState.
  02  exit-ich              PIC 9(2)      COMP.
  02  filler                PIC 9(2)      COMP.
  02  exit-ch               PIC X(1).
  02  filler                PIC X(1).
  02  fAutoExit             PIC 9(2)      COMP.
  02  filler                PIC 9(2)      COMP.
  02  fModified             PIC 9(2)      COMP.
  02  filler                PIC 9(2)      COMP.
  02  fEmpty                PIC 9(2)      COMP.
  02  filler                PIC 9(2)      COMP.
  02  filler                PIC X(6).
```

```
        01  cbFieldInfo             PIC 9(4)      COMP VALUE 32.

        01  fieldInfo.
            02  info-iCol           PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-iLine          PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-cCol           PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-fShowDefault   PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-fAutoExit      PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-fRepeating     PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-attrSel        PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-attrUnsel      PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-indexFirst     PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  info-indexLast      PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
            02  filler              PIC X(10).
            02  info-cchDefault     PIC 9(2)      COMP.
            02  filler              PIC 9(2)      COMP.
    *       02  info-rgchDefault    PIC X(?).
```

The parameter data definitions listed above are needed in the procedures GetFieldInfo and UserFillField.

Use fieldInfo and cbFieldInfo as the last two parameters to GetFieldInfo. When GetFieldInfo returns, field information is accessed by referencing the elementary data items subordinate to fieldInfo.

Use InitState and ExitState as the last two parameters to UserFillField. Initialize the init-ich field of InitState before the call to UserFillField. When UserFillField returns, field state is accessed by referencing the elementary items subordinate to ExitState.

Built-in non-COBOL procedures

The following non-COBOL procedures are built-in. You can call them without configuring COBOL.

ConvertWord

Syntax:

```
CALL "&ConvertWord" USING WORD-IN, WORD-OUT.
```

where

```
WORD-IN     IS PIC 9(4)     USAGE IS COMP.
WORD-OUT    IS PIC 9(4)     USAGE IS COMP.
```

Action:

Reorders the bytes that comprise WORD-IN and stores the results in WORD-OUT. WORD-IN and WORD-OUT may be the same data item.

ConvertQuad

Syntax:

```
Call "&ConvertQuad" USING QUAD-IN, QUAD-OUT.
```

where

```
QUAD-IN     IS PIC 9(9)     USAGE IS COMP.
QUAD-OUT    IS PIC 9(9)     USAGE IS COMP.
```

Action:

Reorders the bytes that comprise QUAD-IN and stores the results in QUAD-OUT. QUAD-IN and QUAD-OUT may be the same data item.

GetPointer

Syntax:

```
CALL "&GetPointer" USING POINTER, DATA-VAL.
```

where

```
POINTER     IS PIC 9(9)     USAGE IS COMP.
DATA-VAL    IS any picture clause.
```

Action:

The memory address of DATA-VAL is stored in POINTER.

MakePointer

Syntax:

```
CALL "&MakePointer" USING POINTER, SEGMENT-ADDR, RELATIVE-ADDR.
```

where

POINTER	IS PIC 9(9)	USAGE IS COMP.
SEGMENT-ADDR	IS PIC 9(4)	USAGE IS COMP.
RELATIVE-ADDR	IS PIC 9(4)	USAGE IS COMP.

Action:

The pointer whose segment address is SEGMENT-ADDR and relative address is REALTIVE-ADDR is stored into POINTER.

UnMakePointer

Syntax:

```
CALL "&UnMakePointer" USING POINTER, SEGMENT-ADDR, RELATIVE-ADDR.
```

where

POINTER	IS PIC 9(9)	USAGE IS COMP.
SEGMENT-ADDR	IS PIC 9(4)	USAGE IS COMP.
RELATIVE-ADDR	IS PIC 9(4)	USAGE IS COMP.

Action:

The segment address portion of POINTER is stored in SEGMENT-ADDR. The relative address portion is stored in RELATIVE-ADDR.

WordAligned

Syntax:

```
CALL "&WordAligned" USING FLAG, DATA-VAL.
```

where

FLAG	IS PIC 9(2)	USAGE IS COMP.
DATA-VAL	IS any picture clause.	

Action:

If DATA-VAL is word aligned, a nonzero value is stored into FLAG; otherwise, 0 is stored into FLAG-VAL.

CONFIGURING COBOL

To configure a COBOL run-time system in which non-COBOL procedures can be called, create a run file (Cobol.run) that contains the COBOL run-time system, a data structure defining the non-COBOL procedures, and the actual non-COBOL procedures. The process of creating a new Cobol.run is described below.

To create Cobol.run, follow the five steps below.

1. <u>Copying the COBOL Generation Files</u>

   Login to a working directory of your choice. Copy the contents of the CUSTOMIZER Distribution Diskettes into your directory. Several files, including CobolGen.asm, Cobol.lib, StartCobolLink.sub, objCblSwp.fls, objCblRes.fls, and libCbl.fls, should be copied.

2. <u>Editing CobolGen.asm (Optional)</u>

   If you are simply configuring in procedures that are already included in CobolGen.asm, skip this step.

   Invoke the Editor to modify CobolGen.asm, the assembly language module that defines the interface of non-Cobol procedures.

   Add an entry for each new non-COBOL procedure. Comments within CobolGen.asm explain how to add an entry.

3. <u>Assembling CobolGen.asm</u>

   Assemble Cobol Gen.asm to produce CobolGen.obj. (See the B 20 Systems Programmers Guide Part 2.)

```
Command Assemble RETURN
Assemble
    Source Files                          CobolGen.asm GO
    [Errors only?]
    [GenOnly, NoGen, or Gen]
    [Object File]
    [List File]
    [Error File]
    [List on pass 1?]
```

   During assembly, the assembler asks questions of this type:

   Are you calling Forms (y or n)?

   If you answer y (for yes) to a question, the assembler creates an entry for each procedure in the corresponding Burroughs software package. To answer no to a question, type n RETURN or just RETURN.

   The procedures associated with each software package are described in the documentation of that package. The procedure names and interfaces are also part of the file CobolGen.asm. These interfaces are consistent with the current releases of the various software packages. However, subsequent releases of CTOS.lib, Forms.lib, etc., in which interfaces have been added or changed, may required corresponding revision of CobolGen.asm.

4. Linking Cobol.run

Link CobolGen.obj, Cobol.lib (the COBOL run-time system in object module format), and the object modules for all non-COBOL procedures to produce Cobol.run.

Use the submit file StartCobolLink.sub and the Executive's submit command to link Cobol.run.

StartCobolLink.sub allows an optional parameter to be typed into the [Parameters] field of the Submit command form. Allowed parameter values are:

Swp which causes a swapping version of Cobol.run to be created. The [Object files] field of the Link command forms is filled in using the contents of the file Cobol.fls. This is also the default case.

Res which causes a resident version of Cobol.run to be created. The [Object files] field of the Link command form is filled in using the contents of the file CobolRes.fls.

In general, the swapping version of Cobol.run is preferable to the resident version because it requres less memory, with only a small performance degradation.

If you are adding new object modules to Cobol.run, edit either CobolRes.fls or Cobol.fls to make the additions. Insert new object filenames after CobolGen.obj.

If you are adding additional libraries, such as Forms.lib, edit CobolLib.mod and add the new libraries to the end of the list.

NOTE

The Linker automatically searches CTOS.lib, therefore, you do not need to include CTOS.lib in CobolLib.mod.

The example below shows how to link a COBOL run-time system that can CALL the Forms package. It is assumed that when you assembled CobolGen.asm, you answered yes to these questions:

Are you calling Forms (y or n)?
Are you calling CTOS OpenFile or CloseFile (y or n)?

In this example, it is assumed that libCbl.fls has been edited to include Forms.lib at the end.

```
Command Submit RETURN
Submit
     [Parameters]                    StartCobolLink.sub GO
     [Force expansion?]              Swp
     [Show expansion]
```

LinkCobol.sub displays the Linker command form and fills in these fields.

```
Command Link RETURN
Link
     Object modules                  Cobol.lib (1 2) CobolGen.obj ...
     Run file                        Cobol.run
     [List file]
     [Publics?]
     [Line numbers?]
     [Stack size]
     [Max memory array size]         48400
     [Min memory array size]         6000
     [System build?]
     [Version]                       '4.0 Swp'
     [Libraries]                     [sys]<sys>SortMerge.lib ...
     [DS allocation?]
     [Symbol file]
```

When the form is filled in , the cursor is left at the end of the [Version] field. Make any additions to the form that you require and press GO.

5. Updating the [sys]<sys>Directory

Copy Cobol.run to [sys] <sys> Cobol.run. You can now call non-COBOL procedures.

Linking with Nonstandard Segments

The COBOL run-time system depends upon a particular ordering of segments in memory for correct operation. (The Linker/Librarian Reference Manual describes segment ordering.) If you are calling object modules that are supplied by Burroughs or are created with the FORTRAN or PASCAL compilers, the required segment order is guaranteed.

However, if you are calling object modules produced by the Assembler, the required segment ordering is guaranteed only if you restrict segment class names to 'data,' 'stack,' or 'code'.

THE COBOL DEBUGGER

Using the interactive COBOL Debugger, you can control the execution of a COBOL program, display and modify data defined in the Data Division, and create your own command macros.

Enable the COBOL Debugger by filling in [Enable COBOL Debugger?] of the CRun form with Yes. When enabled, the COBOL Debugger announces its presence as follows:

```
COBOL Debugger 6.2        - title
?                         - prompt
```

The ? means that the Debugger is ready to accept Debugger commands, which are described below.

Command Summary

A addr val	modify a byte of data
B addr	execute until data at addr changes
C val	display ASCII character corresponding to val
D addr	display 16 bytes from specified address
D,	display the next 16 bytes
E addr val	execute until data at addr changes to val
G addr	goto (execute until) the specified address
L	output a linefeed on the video display
M name	start a Debugger macro definition with the specified name
N	set relative addressing base to start of user data area (relative mode)
O	set relative addressing base to the current 8086 segment (absolute mode)
P	display the current program counter (p-c)
Q	return to the Executive
S addr	open an address for display or modification
T addr	trace paragraphs up to the specified breakpoint
X	execute one COBOL instruction
$	end a macro definition
/	display byte at the current open address
. val	modify the byte at the current open address and open the next address
,	open the next address
;	start a comment line (all input up to the next RETURN is ignored)

Debugger command arguments are:

addr	represents either an offset or an 8086 segmented address. As an offset, addr is specified by exactly FOUR hexadecimal digits. If the Debugger is in absolute mode, the offset is based from the currently defined 8086 segment. Otherwise, the offset is relative to the beginning of the current user area. Relative offsets correspond to the location addresses found along the right side of a COBOL listing. As an 8086 segmented address, addr is specified by four hexadecimal digits, a colon, and four more hexadecimal digits (e.g., 0123:4567). The first set of digits specifies an 8086 segment; the last set specifies an offset. When a full address is used, the Debugger enters absolute mode and a new 8086 segment is defined.
val	is a two digit hexadecimal number or an ASCII character preceded by a double quote. For example, an uppercase A is specified by either 41 or "A".
name	is a single ASCII character specifying a macro name.

General Information

The Debugger displays a question mark as a prompt.

All numbers in the interactive COBOL Debugger, both on input and output, are hexadecimal.

Input either upper- or lowercase letters. The Debugger is case insensitive.

You can put more than one Debugger command on the same line. Terminate a line of input by pressing RETURN.

The Debugger ignores spaces.

If you press RETURN before giving enough input (for example, only 2 rather than 4 digits of an address), the Debugger waits for the remainder of the input. Type it in and press RETURN again.

The Debugger responds to incorrect commands by displaying "-error."

The following examples reference the listing below.

```
000010  DATA DIVISION.                                          011E
000020  WORKING-STORAGE SECTION.                                020E 00
000030  01 FIELDS.                                              020E 00
000040      02 FIELD-1 PIC XXX VALUE "ABC".                     020E 00
000050      02 FIELD-2 PIC XXX VALUE "XYZ".                     0211 03
000060      02 FIELD-3 PIC X(80) VALUE SPACE.                   0214 06
000070  PROCEDURE DIVISION.                                     0000
000080  PARA-1.                                                 001C
000090      MOVE FIELD-1 TO FIELD-2.                            001D
000100      GO TO PARA-1.                                       0022
```

COBOL Debugger Commands

A Command

Syntax and Action:

A addr val

The A command modifies a byte of data in the Data Division. Specify the new data using either two hexadecimal digits or an ASCII character preceded by a double quote character.

Example:

To replace the first character of FIELD-1 by "G" and to display the modified byte, type:

```
?A 020E 47 RETURN
?D 020E RETURN
47-G 42-B 43-C 41-A 42-B 43-C 20-   20-   .....
?
```

The following syntax also works.

```
?A 020E "G RETURN
?
```

B Command

Syntax and Action:

B addr

The B command executes the COBOL program until the data value at addr changes. When the value changes, the Debugger displays the current program counter and the new data value.

Example:

Execute until the FIELD-2 is assigned data.

```
?B 0211
022 41-A
?
```

D Command

Syntax and Action:

D addr

The D command displays 16 bytes of data beginning at addr. To display data items in WORKING-STORAGE, use relative mode (the default); the program listing gives the offsets of data items along the right side.

Bytes are displayed in hexadecimal and ASCII (if they can be printed).

Example:

To display the contents of FIELD-1 and FIELD-2 before the MOVE statement is executed, type:

```
?D 020E RETURN
41-A 42-B 43-C 58-X 59-Y 5A-Z 20-  20-  .....
```

The first 3 bytes, "ABC", represent FIELD-1; the next 3, "XYZ", represent FIELD-2.

D Command

Syntax and Action:

D

The D command displays the next 16 bytes of data. The current addr is incremented by 16 and bytes are displayed as in the D command.

E Command

Syntax and Action:

E addr val

The E command executes the COBOL program until the data value at addr is equal to val. When the value changes, the Debugger displays the current program counter and the new data value.

Example:

Execute until FIELD-2 has the value "A".

```
?E 0211 "A
022 41-A
?
```

G Command

Syntax and Action:

G addr

The G command executes the COBOL program until addr is reached. If addr is never reached, the program continues and control never returns to the COBOL Debugger.

Use the location addresses to the right of the program listing to determine the address of a COBOL instruction.

Example:

To go to the statement "MOVE FIELD-1 TO FIELD-2", type:

```
?G 001D RETURN
?
```

The second question mark above indicates that the statement has been reached.

To check on the current address at this point, use the P command as follows:

```
?P RETURN
 001D - returns p-c
?
```

N Command

Syntax and Action:

N

Set the addressing mode to relative. In subsequent Debugger commands, when the offset form of addr is used, the offset is relative to the start of the user data area.

In relative mode, offsets correspond to the location addresses on the right side of the program listing.

Relative mode is the default.

O Command

Syntax and Action:

O

Set the addressing mode to absolute. In subsequent Debugger command, when the offset form of addr is used, the offset is based from the current 8086 segment.

The default 8086 segment is 0000. It is changed by using the segment: offset form of addr in a Debugger command.

Absolute mode is entered automatically when the segment: offset form of addr is used.

P Command

Syntax and Action:

P

The P command displays the current program counter (p-c), that is, the location address of the next instruction.

Example:

At the start of a program the p-c is a 0000 as shown below.

```
?P RETURN                    -command
 0000                        -current p-c
?
```

> NOTE
>
> The location address given by the P command is relative to the start of the Procedure Division.

Q Command

Syntax and Action:

Q

The Q command exits the COBOL Debugger and returns to the Executive.

S and Related Commands

Syntax and Action:

```
S addr
/
. val
,
```

To facilitate the display and modification of data, the COBOL Debugger provides commands for opening an address for display or modification, displaying or modifying a byte of data at the open address, and opening the next byte for display or modification.

The S command opens an address for display or modification.

The / command displays the byte at the current open address.

The .val command modifies the byte at the current open address with the data specified and opens the next address.

The , command opens the next address.

Example:

To display the first byte of FIELD-1, type:

```
?S 020E RETURN          -opens address
?/ RETURN               -displays byte at open address
 020E 47-A
?
```

To change FIELD-1 to "DEF" and display the modified bytes, type:

```
?S 020E RETURN          -opens address
?.44.45.46 RETURN       -modifies 3 bytes
?S 020E RETURN          -reopens original address
?/,/,/ RETURN           -display of bytes
 020E 44-D
 020F 45-E
 0210 46-F
?
```

NOTE

In the last example, you must use the , command to open the next address after you display a byte. This is not necessary when you use the . command because the next address is opened automatically.

T Command

Syntax and Action:

T addr

The G command executes the COBOL Program until addr is reached. If addr is never reached, the program continues and control never returns to the COBOL Debugger.

The T command is the same as the G command, except that the T command provides a trace of paragraphs encountered.

X Command

Syntax and Action:

X

The X command executes a single COBOL instruction. After the instruction is executed, the current p-c is displayed. Since a line of COBOL source can be translated into several instructions, X may appear to halt in the middle of a line.

Example:

To execute th next instruction, which is MOVE FIELD-1 TO FIELD-2, and to redisplay the data, type:

```
?X RETURN                   - assuming the current p-c is 001D
 0022
?D 020E RETURN
41-A 42-B 43-C 41-A 42-B 43-C 20-  20-  .....
```

NOTE

FIELD-2 has changed to "ABC", as expected.

Macro Commands

You can define macros consisting of both basic Debugger commands and othe macros. Macros are named by a single ASCII character.

If you make an error while typling a macro definition, end the current definition and begin again.

The Debugger provides a limited amount of space for macro definitions. If space runs out or if the maximum nesting of macros is exceeded, then the Debugger will display the message "stack overflow". After a stack overflow occurs, the Debugger will attempt to recover and return to command level.

C Command

Syntax and Action:

```
C val
```

The C command displays a single ASCII character on the video display.

Example:

To display the character "A" on the video display, type:

```
?C "A
A
?
```

L Command

Syntax and Action:

```
L
```

The L command displays a linefeed on the video display.

M Command

Syntax and Action:

M name

The M command introduces and names a macro. Type the macro name, a single ASCII character, immediately after M.

Example:

To define a macro named "Z" to execute up to 001D, display 16 bytes beginning at 020E, then single step and display again, type:

```
?MZ G 001D D 020E L X D 020E $ RETURN
?
```

The L and $ commands appearing in this macro are described as follows. To invoke this macro, type:

```
?Z RETURN
41-A 42-B 43-C 58-X 59-Y 5A-Z 20-  20-  .....
0022
41-1 42-B 43-C 41-A 42-B 43-C 20-  20-  .....
?
```

$ Command

Syntax and Action:

$

The $ command ends a macro definition.

; Command

Syntax and Action:

; comment RETURN

; begins a comment. All characters typed after ; up to the next RETURN are ignored by the COBOL Debugger.

Example:

The previous macro definition with a comment is:

```
?MZ G 001D D 020E L X D 020E $; this is a comment RETURN
?
```

Saving Debugger Macros

You can save Debugger macros using the Burroughs Editor and the Executive's Submit facility. First, use the Editor to create a submit file that invokes the CRun form and fills in [Enable COBOL Debugger] with Yes and [Intermediate file] with the appropriate file name. Use the remainder of the submit file for macro definitions.

When you run your program using the submit file, the macro definitions are entered automatically.

The following example enables the Debugger and runs Test.int. The macro Z is defined which prints 4 bytes of data beginning at 020E.

```
CRun RETURN
Test.int RETURN
RETURN
RETURN
Yes GO
MZ S 020E /,/,/,/$ ; print 4 bytes RETURN
```

APPENDIX K

PROGRAMMING HINTS

HINT 1: CALLING MEMORY MANAGEMENT AND RSAM FROM COBOL

Certain BTOS system services require the calling program to pass word aligned buffers. RSAM is one service which requires this type of buffer. Since Cobol does not guarantee word alignment of data areas, BTOS memory management is called to allocate word aligned memory segments. This Hint will demonstrate these calls using a Cobol program.

Since Cobol passes parameters by reference and value, some modifications must be made to the lookup table called rgProcedures. This table is used by the run-time system to obtain information about procedure interfaces. This table is stored in the file called CobolGen.asm. The example table entry must be edited to allow the OpenRSFile procedure access to the memory segment allocated by a call to memory management. Example:

```
%TableEntry(w,OPENRSFILE,8,r,r,w,r,w,w,q,w)
```

The parameter in bold (the q) must be changed from an "r" (pass by reference) to a "q" (pass quad by value) as shown. This parameter must be changed since the contents of its corresponding Cobol variable already contains the address of the buffer area to be passed to RSAM. The next step is to assemble CobolGen.asm. Answer "Y" to the following questions:

Are you calling the Record Sequential Access Method (y or n)?
Are you calling BTOS Memory Management (y or n)?

Cobol will now be configured to run the program included in this Hint.

Call	Comment
"&AllocateMemorySL"	Allocate a 1024 byte word aligned buffer area.
"&OpenRSFile"	Create and open an RSAM file in write mode.
"&WriteRSRecord"	Write an 80 byte record to the file.
"&CheckpointRSFile"	Write the partially full buffer before continuing any further.
"&CloseRSFile"	Close the file.

```
Identification Division
Program-Id.  RSAM Test.
Date-Compiled.
Environment Division.
Configuration Section.
Source-Computer.  B 20.
Object-Computer.  B 20.
Input-Output Section.
File-Control.
Data Division.
Working-Storage Section.
01  RSWA                  Pic X(150).
01  RecordArea.
    03   Rec-Num          Pic X(3).
    03   Rec-Text         Pic X(77).
77  Segment-bytes         Pic 9(4) Comp Value 1024.
77  RecordSize            Pic 9(4) Comp Value 80.
77  Bytes-Returned        Pic 9(4) Comp.
77  FileSpec              Pic X(12)  Value "RSFILE.JERRY".
77  FileSpec-bytes        Pic 9(4)  Comp Value 12.
77  PswdSpec              Pic X.
77  PswdSpec-bytes        Pic 9(4)  Comp Value Zero.
77  Erc                   Pic 9(4)  Comp.
77  Open-Mode             Pic X(2)  Value "mw".
77  SLMemoryPtr           Pic 9(9)  Comp.
77  Error-AllocSL         Pic X(8)  Value "ALLOCSL ".
77  Error-CreateRS        Pic X(8)  Value "CREATERS".
77  Error-WriteRS         Pic X(8)  Value "WRITERS ".
77  Error-CkPoint         Pic X(8)  Value "CKPOINT ".
77  Error-Msg             Pic X(8).
Procedure Division.
Proc-Option-Main.
    Perform AllocSL-Routine.
    Perform CreateRS-Routine.
    Perform WriteRS-Routine.
    Perform CkPoint-Routine.
    Call "&CloseRSFile" using Erc,
                              RSWA.
    Stop run.
AllocSL-Routine.
    Call "&AllocMemorySL" using Erc,
                              Segment-bytes,
                              SLMemoryPtr.
```

```
          If Erc not = Zeroes
                 Then
                     Move Error-AllocSL to Error-Msg
                     Perform Error-Routine
                 Else
                     Next Sentence.
      CreateRS-Routine.
          Call "&OpenRSFile" using Erc,
                                    RSWA,
                                    FileSpec, FileSpec-bytes,
                                    PswdSpec, PswdSpec-bytes,
                                    Open-Mode,
                                    SLMemoryPtr, Segment-bytes.
          If Erc not = Zeroes
                 Then
                     Move Error-CreateRS to Error-Msg
                     Perform Error-Routine.
                 Else
                     Next Sentence.
      WriteRS-Routine.
          Move "747" to Rec-Num.
          Move "SCZEPURA" to Rec-Text.
          Call "&WriteRsRecord" using Erc,
                                       RSWA,
                                       RecordArea, RecordSize.
          If Erc not = Zeroes
                 Then
                     Move Error-WriteRS to Error-Msg
                     Perform Error-Routine
                 Else
                     Next Sentence.
      CkPoint-Routine.
          Call "&CheckpointRsFile" using Erc,
                                          RSWA.
          If Erc not = Zeroes
                 Then
                     Move Error-CkPoint to Error-Msg
                     Perform Error-Routine.
                 Else
                     Next Sentence.
      Error-Routine.
          Display Error-Msg Erc upon Console.
          Stop Run.
```

HINT 2: ALTERNATIVES TO THE COBOL DISPLAY STATEMENT

The Cobol extended DISPLAY statement is normally used to select the location of messages displayed on the video. This Hint describes two alternatives to the use of the Cobol DISPLAY statement.

One method uses only Cobol native syntax, using the Sequential Access Method (SAM) by SELECTing a file with the Organization is Line Sequential clause. Then, a cursor positioning escape sequence is included in the first four bytes of the record (01) level identifier in Working Storage. The message is sent to the video using the Cobol WRITE statement.

The second method uses the Video Access Method (VAM), which Calls the PutFrameChars system common procedure. This procedure allows you to specify the horizontal and vertical coordinates within a frame where the text string is to be moved.

The following program demonstrates an alternative to using the extended DISPLAY statement. Only native Cobol syntax is employed. SAM is implied by way of the SELECT statement.

```
Identification Division.
Program-Id. Video-1.
Environment Division.
Configuration Section.
Input-Output Section.
File-Control.
    Select Vid-file
        Assign to "[vid]"
        Organization is Line Sequential.
Data Division.
File Section.
FD Vid-File.
01 Rec-Desc                Pic X(14).
Working-Storage Section.
01 Video-Record.
    03  Filler             Pic X(1) Value X"FF".
    03  Filler             Pic X(1) Value "C".
    03  Col-position       Pic 99 Comp.
    03  Line-position      Pic 99 Comp.
    03  Message-vid        Pic X(10).
Procedure Division.
Proc-Main.
    Open output Vid-file.
    Move "Video Test" to Message-vid.
    Move 25 to Col-position.
    Move 10 to Line-position.
    Write Rec-desc from Video-Record.
    Close Vid-file.
    Stop run.
```

The following program demonstrates another alternative to using the DISPLAY statement. This method uses a Call to the Video Access Method (VAM).

```
Identification Division.
Program-Id. Video-2.
Environment Division.
Configuration Section.
Data Division.
Working-Storage Section.
77 Frame-number     Pic 9(4) Comp Value Zero.
77 Col-position     Pic 9(4) Comp.
77 Line-position    Pic 9(4) Comp.
77 Message-vid      Pic X(10) Value "Video Test".
77 Message-length   Pic 9(4) Comp Value 10.
77 ERC
Procedure Division.
Proc-Main.
    Move 25 to Col-position.
    Move 10 to Line-position.
    Call "&PutFrameChars" using ERC,
                                Frame-number,
                                Col-position,
                                Line-position,
                                Message-vid,
                                Message-length.
    Stop Run.
```

HINT 3: ACCESSING THE SYSTEM DATE AND TIME USING COBOL

Most applications occasionally need to include the current date and/or time in their processing. There are several procedural calls available in BTOS to allow the user to retrieve the date and time field from the system and expand it into a readable day, date, and time.

With date and time manipulation in BTOS, there are basically two structures involved. The date and time is kept internally in system memory as a three-word field containing the count of 50 or 60 Hz clock ticks, the count of 100ms periods elapsed since the last second, the count of seconds since midnight or noon, and the count of 12-hour periods since March 1, 1952. (See the B 20 Operating system (BTOS) reference Manual, the "Timer Management" section.) The last two words are returned to the program when the date/time is requested; the first word can be examined when precise timings are needed. The expanded date and time format is a four-word structure with the year, month, day of month, day of week, hour, minute, and second imbedded in it.

The compact system format can be used to time-stamp records, for example, while only occupying a four-byte field. The format of the compacted date also makes it useful for date calculations. For example, the date of thirty days from now can be obtained by adding 60 (12 hour periods) to the count which specifies days in the system format, then expand it from there. If two dates are subtracted, the result divided by two is the number of days apart the two are. The day-of-week field can also be examined in a program (it is returned initially as a number 0=Sun to 6=Sat) to perhaps look for the next business day after thirty days from now.

The following calls are available in BTOS to access the system date/time structure, and are documented in the BTOS Operating System manual.

CompactDateTime	Converts the expanded date/time format to the system format.
ExpandDateTime	Expands the system format to the expanded date/time format.
GetDateTime	Returns the current date and time in the system format.
SetDateTime	Sets the data and time for the system.

Analyzing the expanded date/time format using these routines can be tricky in high-level languages. The expanded date is returned to the program as a 64-bit data type, for which few of the languages have a built-in structure. However, facilities are available for the information to be extracted.

Example:

In a Cobol program, the date and time can be obtained from the system using the ACCEPT verb. The statement "ACCEPT Date-Field FROM DATE" returns a six-digit value in the form YYMMDD to Date-Field. The statement "ACCEPT Day-Field FROM DAY" returns a five-digit value to Day-Field in the form YYDDD where DDD is the day number of the year. The statement "ACCEPT Time-Field FROM TIME" returns an eight-digit value to Time-Field in the form HHMMSS00. Refer to the discussion of the ACCEPT verb in the B 20 Systems Cobol Reference Manual.

If "GetDateTime" and "ExpandDateTime" are to be used in a Cobol program, a structure can be defined that breaks the expanded date and time down into the individual fields. Then the reordering of bytes must be handled since Cobol expects the bytes to be ordered in a different way than the non-Cobol procedures "GetDateTime" and "ExpandDateTime". The result of calling "GetDateTime" is returned in a 32-bit field, which is equivalent to a quad value in length. The "ConvertQuad" routine should then be called to reorder the bytes of this field since the Cobol run-time passes this parameter by reference. You should be aware that this field does not follow the general rules for parameter conversion as described in appendix J of the B 20 Systems Cobol Reference Manual. That is, making two calls to "ConvertWord" with the fields "Tim" and "Dat" respectively will not produce the correct conversion. The result of calling "ExpandDateTime" is returned to a 64-bit structure consisting of a word field (the year), and six-byte field (the month, day of month, day of week, hour, minute, and second). The "ConvertWord" routine should then be called to reorder the bytes in the year field only. If "CompactDateTime" is to be called, the fields should be converted back before calling it.

The following program uses the ACCEPT verb to get the date/time.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AcceptDate.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. B 20.
OBJECT-COMPUTER. B 20.
SPECIAL-NAMES. CONSOLE IS CRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TheDate.
    03  Year        PIC  X(2).
    03  Month       PIC  X(2).
    03  DayOfMo     PIC  X(2).
01  JulianDate.
    03  Year2       PIC  X(2).
    03  DayOfYr     PIC  X(3).
01  TheTime.
    03  Hour        PIC  X(2).
    03  Minute      PIC  X(2).
    03  Second      PIC  X(2).
    03  Hundrths    PIC  X(2).
PROCEDURE DIVISION.
GET-DATE.
```

```
    ACCEPT TheDate FROM DATE.
    ACCEPT JulianDate FROM DATE.
    ACCEPT TheTime FROM TIME.
DISPLAY-IT.
    DISPLAY TheDate AT 0105.
    DISPLAY JulianDate AT 0205.
    DISPLAY TheTime AT 0305.
PAUSE.
    STOP "&"
    STOP RUN.
```

The following program uses calls to get the date/time.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DateTime.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. B 20.
OBJECT-COMPUTER. B 20.
SPECIAL-NAMES. CONSOLE IS CRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Erc             PIC 9(4) COMP.
01  DatTimRet.
    03  Tim         PIC 9(4) COMP.
    03  Dat         PIC 9(4) COMP.
01  ExpDateTime.
    03  Word1       PIC 9(4) COMP.
    03  Word2       PIC 9(4) COMP.
    03  Word3       PIC 9(4) COMP.
    03  Word4       PIC 9(4) COMP.
01  LongDate REFEFINES ExpDateTime.
    03  The Date.
        05  Year        PIC 9(4) COMP.
        05  Month       PIC 9(2) COMP.
        05  DayOfMo     PIC 9(2) COMP.
        05  DayOfWk     PIC 9(2) COMP.
    03  The Time.
        05  Hour        PIC 9(2) COMP.
        05  Minute      PIC 9(2) COMP.
        05  Second      PIC 9(2) COMP.
01  DayInAscii.
    03  Label1      PIC X(15) VALUE "DAY OF WEEK IS".
    03  WeekDay     PIC X(3).
01  DateInDec.
    03  Label2      PIC X(9) VALUE " DATE IS ".
    03  Mo          PIC XX.
    03  Slash       PIC X VALUE "/".
    03  Dy          PIC XX.
    03  Label3      PIC X(13) VALUE " OF THE YEAR ".
    03  Yr          PIC X(4).
01  TimeInDec.
    03  Label4      PIC X(9) VALUE "TIME IS ".
    03  Hr          PIC XX.
```

```
        03  Colon1        PIC X VALUE ":".
        03  Min           PIC XX.
        03  Colon2        PIC X VALUE ":".
        03  Sec           PIC XX.
    01  DayName           PIC X(3) OCCURS 7 TIMES.
    01  One               PIC 9(4) COMP VALUE 1.
    PROCEDURE DIVISION.
    INIT.
        MOVE "Sun" TO DayName(1).
        MOVE "Mon" TO DayName(2).
        MOVE "Tue" TO DayName(3).
        MOVE "Wed" TO DayName(4).
        MOVE "Thu" TO DayName(5).
        MOVE "Fri" TO DayName(6).
        MOVE "Sat" TO DayName(7).
    GET-TIME.
        CALL "&GetDateTime" USING Erc, DatTimRet.
        If Erc NOT EQUAL ZERO
          THEN PERFORM ERROR-EXIT.
        CALL "&ConvertQuad" USING DatTimRet, DatTimRet.
    EXPAND-TIME.
        CALL "&ExpandDateTime" USING Erc, DatTimRet, ExpDateTime.
        IF Erc NOT EQUAL ZERO.
          THEN PERFORM ERROR-EXIT.
        CALL "&ConvertWord" USING Word1, Word1.
    MOVE-IT.
        ADD One TO DayOfWk.
        MOVE DayName(DayOfWk) TO WeekDay.
        MOVE DayOfMo TO Dy.
        Add One TO Month.
        MOVE Year TO Yr.
        MOVE Hour TO Hr.
        MOVE Minute TO Min.
        MOVE Second TO Sec.
    DISPLAY-IT.
        DISPLAY DayInAscii AT 0101.
        DISPLAY DateInDec AT 0201.
        DISPLAY TimeInDec AT 0301.
    ERROR EXIT.
        DISPLAY Erc.
        STOP "i".
        STOP RUN.
```

HINT 4: LIMITATIONS AND RESTRICTIONS

If START file-name KEY > relative-key-name clause is used on a file that has ORGANIZATION IS RELATIVE, and the value of the relative-key-name is past the end of the file, the system will hang. Use a READ clause instead of START to determine the end of the file.

CLOSE filename WITH LOCK causes error 9D on any program that uses that file.

DISPLAYing a null string, as in the statment DISPLAY "", causes a runtime error.

An OPEN OUTPUT of a filename that already exists will not be deleted by the system.

Runtime Error 215 occurs if an attempt is made to WRITE to device "[Splb]" using the default ORGANIZATION. This spool file must be ORGANIZATION LINE SEQUENTIAL.

A picture clause described as being Alpha will accept Numeric data.

A picture clause described as being -99 (or numeric) will accept Alphabetic data.

The top margin of logical page printing does not work on the first page.

ADD CORRESPONDING with ON SIZE condition always uses the error path (that is, always uses the ON SIZE condition whether it is true or not).

SUBSTRACT CORRESPONDING with ON SIZE condition always uses the error path (that is, always uses the ON SIZE condition whether it is true or not).

In a COMPUTE statement,the expression "data-name + - (data-name)" does not work. Instead, use "data-name = -1 * data-name" or "data-name = 0 - (data-name)".

The COBOL compiler displays a Run Time error when it encounters more than two logical print files. To avoid this, introduce a variable equal to the linage-counter of one of the files. For this file to always equal the value of the linage-counter, you update the variable after each Write and Open operation.

STOP-RUN does not generate a compile time error.

Cobol error messages for an undefined data-name on an ASCENDING KEY clause are not clear.

A MOVE to an array (or subscripted data-name) from an array (or subscripted data-name) gives the wrong values.

Using READ...AT END causes error 101 at the AT END, when using RANDOM or DYNAMIC access.

When in Sequential Access Mode, the OPEN command cannot be used with the I-O option on an Indexed file. To perform this function, either create two statements (OPEN OUTPUT option and OPEN INPUT option) or use Dynamic Access Mode. If you write two OPEN statements, you must create a line sequential file which is used as an input file, and will be opened with the OPEN INPUT option.

When CALL .. ON OVERFLOW encounters an OVERFLOW condition the system will hang.

ADD or SUBTRACT after DISPLAY gives erroneous results during run time.

WRITE works as a REWRITE after a REWRITE in RELATIVE ORGANIZATION.

ADDS to SIGNED LEADING SEPARATE FIELD are erroneous.

DISPLAY AT statement affects allocated short-lived memory.

Compilation statistics are incorrect.

Call to "READFIELD" for a form in user library wipes out memory in working storage.

The Executive does not display message from Cobol run-file ERC 189 .

CLOSE FILE-NAME WITH LOCK in INDEXED MODE gives ERC 5001 .

CALL command return crashes on EXIT statement with ERC 22 .

When a file name is declared in Working-Storage, Cobol expects the file name to terminate with a space.

The Cobol data dictionary is restricted to 60K.

The CANCEL statement does not guarantee that a Cobol module is in its initial state after a subsequent CALL is made to the same module.

SORTing of records with signed COMP or COMP-3 keys is not implemented because Cobol supports only character keys. Use straight numeric bytes instead (for example, PIC 9(2)).

When using the Editor to create a Cobol source file always press RETURN after the last line of source code. Otherwise, the compiler will "lose" this line.

SORTing of numeric keys (DISPLAY) that have separate signs is not supported. Cobol only supports character keys.

The maximum ISAM primary key cannot be greater than 64 bytes.

Not all graphics calls are available due to Cobol memory limitations. Cobol file 'Cobolgen.asm' on disk 2 contains all valid graphics modules for use with Cobol.

INDEX

INDEX (CONT.)

INDEX (CONT.)

Documentation Evaluation Form

Title: B 20 Systems COBOL II Reference Manual (Release Level 4.0)

Form No: 1180122

Date: May, 1985

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Comment/Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error ☐ Other

Comments:

From:

Name

Title

Company

Address

Phone Number Date

Remove form and mail to:

Burroughs Corporation
Corporate Product
Information East
209 W. Lancaster Ave.