

Burroughs 

B 1800/B 1700

Systems

COBOL

REFERENCE MANUAL

PRICED ITEM

Burroughs 

B 1800/B 1700

Systems

COBOL

REFERENCE MANUAL

Copyright © 1978 Burroughs Corporation, Detroit, Michigan 48232

PRICED ITEM

Burroughs believes that the software described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special, or consequential damages. There are no warranties which extend beyond the program specification.

The Customer should exercise care to assure that use of the software will be in full compliance with laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change. Revisions may be issued from time to time to advise of changes and/or additions.

This revision incorporates information released under the following:

PCN 1057197-001 (3/8/76)

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
	FOREWORD	xv
	Burroughs Extensions to ANSI 68 COBOL	xv
	Acknowledgement	xv
1	INTRODUCTION	1-1
	Advantages of COBOL	1-1
	Program Organization	1-2
2	LANGUAGE FORMATION	2-1
	General	2-1
	Character Set	2-1
	Characters Used for Words	2-1
	Punctuation Characters	2-2
	Characters Used in Editing	2-2
	Characters Used in Formulas	2-2
	Characters Used in Relations	2-2
	Definition of Words	2-3
	Types of Words	2-3
	Nouns	2-3
	Verbs	2-8
	Reserved Words	2-9
	Language Description Notation	2-10
	Key Words	2-10
	Optional Words	2-10
	Generic Terms	2-10
	Braces	2-10
	Brackets	2-10
	Ellipsis	2-11
	Period	2-11
3	CODING FORM	3-1
	General	3-1
	Sequence Field (Card Columns 1-6)	3-1

TABLE OF CONTENTS (Cont)

<u>Section</u>	<u>Page</u>
3	CODING FORM (Cont)
	Continuation Indicator (Column 7) 3-1
	Margin A (Columns 8 thru 11) 3-3
	Margin B (Columns 12 thru 72) 3-3
	Right Margin (Column 72) 3-3
	Identification (Columns 73 thru 80) 3-3
	Punctuation 3-4
	Sample Coding 3-4
4	IDENTIFICATION DIVISION 4-1
	General 4-1
	IDENTIFICATION DIVISION Structure 4-1
	MONITOR 4-2
	Coding the IDENTIFICATION DIVISION 4-3
5	ENVIRONMENT DIVISION 5-1
	General 5-1
	ENVIRONMENT DIVISION Organization 5-1
	ENVIRONMENT DIVISION Structure 5-1
	CONFIGURATION SECTION 5-2
	SOURCE-COMPUTER 5-3
	OBJECT-COMPUTER 5-4
	SPECIAL-NAMES 5-6
	INPUT-OUTPUT SECTION 5-8
	FILE-CONTROL..... 5-9
	I-O-CONTROL 5-15
	Coding the ENVIRONMENT DIVISION 5-17
6	DATA DIVISION 6-1
	General 6-1
	DATA DIVISION Organization 6-1
	DATA DIVISION Structure 6-2
	File and Record Concepts 6-3
	Physical Aspects of a File..... 6-3
	Conceptual Characteristics of a File 6-3
	Record Concepts 6-4

TABLE OF CONTENTS (Cont)

<u>Section</u>	<u>Page</u>
6 DATA DIVISION (Cont)	
Level Numbers Concept	6-5
Qualification	6-8
Tables	6-11
Subscripting	6-12
Indexing	6-14
Identifier	6-15
FILE SECTION	6-16
FILE DESCRIPTION	6-16
BLOCK	6-19
DATA RECORDS	6-21
FILE CONTAINS	6-22
LABEL	6-24
RECORD	6-25
RECORDING MODE	6-26
VALUE OF ID	6-27
VALUE OF Q-MAX-MESSAGES	6-31
RECORD DESCRIPTION	6-32
BLANK WHEN ZERO	6-35
CONDITION-NAME	6-36
DATA-NAME	6-39
JUSTIFIED	6-40
LEVEL-NUMBER	6-42
OCCURS	6-43
PICTURE	6-48
Categories of Data	6-48
Classes of Data	6-49
Function of the Editing Symbols	6-50
Editing Rules	6-54
Precedence of Symbols	6-58

TABLE OF CONTENTS (Cont)

<u>Section</u>		<u>Page</u>
6	DATA DIVISION (Cont)	
	REDEFINES	6-62
	RENAMES	6-64
	USAGE	6-66
	VALUE	6-69
	WORKING-STORAGE SECTION	6-71
	Organization	6-71
	Non-Contiguous WORKING-STORAGE	6-71
	WORKING-STORAGE Records	6-72
	Initial Values	6-72
	Condition-Names	6-72
	Coding The WORKING-STORAGE SECTION	6-72
7	PROCEDURE DIVISION	7-1
	General	7-1
	Rules of Procedure Formation	7-1
	Execution of PROCEDURE DIVISION	7-2
	Statements	7-3
	Imperative Statements	7-3
	Conditional Statements	7-3
	Compiler-Directing Statements	7-3
	Sentences	7-4
	Imperative Sentences	7-4
	Conditional Sentences	7-4
	Compiler-Directing Sentences	7-4
	Sentence Punctuation	7-5
	Execution of Imperative Sentences	7-5
	Execution of Conditional Sentences	7-5
	Execution of Compiler-Directing Sentences	7-6
	CONTROL Relationship Between Procedures	7-7
	Paragraphs	7-7
	Sections	7-7
	Segmentation	7-9
	Program Segments	7-9
	Segment Classification	7-9
	Priority Numbers	7-10

TABLE OF CONTENTS (Cont)

<u>Section</u>		<u>Page</u>
7	PROCEDURE DIVISION (Cont)	
	Declaratives	7-13
	USE Declarative	7-13
	COPY Statement as a Declarative	7-13
	Arithmetic Expressions	7-14
	Arithmetic Operators	7-14
	Formation and Evaluation Rules	7-15
	Conditions	7-17
	Logical Operators	7-17
	Relation Condition	7-17
	Relational Operators	7-19
	Comparison of Operands	7-19
	Sign Condition	7-20
	Class Condition	7-20
	Condition-Name Condition	7-21
	Evaluation Rules	7-21
	Simple Conditions	7-22
	Compound Conditions	7-22
	Abbreviated Compound Conditions	7-24
	Internal Program Switches	7-26
	Statement Options	7-27
	Rounded Option	7-27
	Size Error Option	7-27
	Corresponding Option	7-28
	Verbs	7-29
	Specific Verb Formats	7-30
	ACCEPT	7-31
	ADD	7-32
	ALTER	7-35
	CLOSE	7-36
	COMPUTE	7-41
	COPY	7-42
	DISPLAY	7-46
	DIVIDE	7-47
	DUMP	7-49
	EXAMINE	7-50
	EXIT	7-52

TABLE OF CONTENTS (Cont)

<u>Section</u>		<u>Page</u>
7	PROCEDURE DIVISION (Cont)	
	GO TO	7-53
	IF	7-55
	MERGE	7-56
	Syntax Rules	7-56
	General Rules	7-57
	MOVE	7-61
	Elementary MOVES	7-62
	Group MOVES	7-63
	Translation	7-64
	Index Data Items	7-64
	Valid MOVE Combinations	7-64
	MULTIPLY	7-66
	NOTE	7-67
	OPEN	7-68
	PERFORM	7-72
	READ	7-78
	RELEASE	7-81
	RETURN	7-82
	SEARCH	7-83
	SEEK	7-87
	SET	7-88
	SORT	7-90
	STOP	7-95
	SUBTRACT	7-96
	TRACE	7-97
	USE	7-98
	WAIT	7-100
	WRITE	7-103
	ZIP	7-106
	Coding the PROCEDURE DIVISION	7-106
8	B 1800/B 1700 COBOL READER-SORTER	8-1
	General	8-1
	Independent Functions	8-2
	COBOL Provisions for Reader-Sorter Handling	8-4
	ENVIRONMENT DIVISION REQUIREMENTS	8-4

TABLE OF CONTENTS (Cont)

<u>Section</u>		<u>Page</u>
8	B 1800/B 1700 COBOL READER-SORTER (Cont)	
	DATA DIVISION Requirements	8-5
	PROCEDURE DIVISION Requirements	8-6
	CLOSE	8-7
	CONTROL	8-8
	FORMAT	8-10
	MICR-EDIT	8-12
	OPEN	8-13
	READ	8-14
	USE	8-15
	Exception Condition Handling	8-16
	Reader-Sorter Characters	8-21
	Reader Sorter Programming Considerations	8-22
	Example Program	8-22
9	DATA COMMUNICATIONS	9-1
	General	9-1
	Specific Verb Formats	9-1
10	INTER-PROGRAM COMMUNICATION	10-1
	General	10-1
	Message Queues	10-1
11	COBOL COMPILER CONTROL	11-1
	General	11-1
	Compilation Card Deck	11-1
	?Compile Card	11-2
	MCP Label Card	11-2
	\$Option Control Card	11-3
	Source Data Card	11-6
	Label Equation Card	11-7
12	DATA MANAGEMENT	12-1
	COBOL Source Program Syntax	12-1
	IDENTIFICATION DIVISION and ENVIRONMENT DIVISION	12-1

TABLE OF CONTENTS (Cont)

<u>Section</u>		<u>Page</u>
12	DATA MANAGEMENT (Cont)	
	DATA DIVISION	12-1
	General	12-1
	Data-Base Section	12-1
	PROCEDURE DIVISION	12-2
13	INDEXED I-O	13-1
	Introduction	13-1
	Language Concepts	13-1
	Organization	13-1
	Access Modes	13-1
	Current Record Pointer	13-2
	I-O Status	13-2
	Status Key 1	13-2
	Status Key 2	13-3
	Valid Combinations of Status Keys	13-4
	The Invalid Key Condition	13-4
	The AT END Condition	13-5
	General Implementation Information	13-5
	Physical Files	13-5
	Tag Files	13-5
	RPG and COBOL Compatibility	13-6
	Rough Tables	13-7
	Searching the Tag File	13-7
	Adding Records to the Files	13-8
	Environment Division for Indexed I-O	13-8
	The File-Control Paragraph	13-8
	Function	13-8
	General Format	13-8
	Syntax Rules	13-9
	General Rules	13-9
	Data Division for Indexed I-O	13-10
	The File Description	13-10
	Function	13-10
	General Format	13-10
	Syntax Rules	13-10

TABLE OF CONTENTS (Cont)

<u>Section</u>	<u>Page</u>
13 INDEXED I-O (Cont)	
The VALUE OF Clause	13-10
Function	13-10
General Format	13-11
Syntax Rule	13-11
General Rules	13-11
Procedure Division for Indexed I-O	13-11
The CLOSE Statement	13-11
Function	13-11
General Format	13-12
Syntax Rule	13-12
General Rules	13-12
The OPEN Statement	13-13
Function	13-13
General Format	13-14
Syntax Rule	13-14
General Rules	13-14
The READ Statement	13-16
Function	13-16
General Format	13-16
Syntax Rules	13-16
General Rules	13-17
The REWRITE Statement	13-19
Function	13-19
General Format	13-19
Syntax Rules	13-19
General Rules	13-19
The START Statement	13-21
Function	13-21
General Format	13-21
Syntax Rules	13-21
General Rules	13-21
The USE Statement	13-22
Function	13-22
General Format	13-22
Syntax Rules	13-23
General Rules	13-23

TABLE OF CONTENTS (Cont)

<u>Section</u>	<u>Page</u>
13 INDEXED I-O (Cont)	
The WRITE Statement	13-23
Function	13-23
General Format	13-24
Syntax Rules	13-24
General Rules	13-24
Appendix A - Reserved Words	A-1
Appendix B - COBOL Syntax Summary	B-1
Appendix C - Compiler Error Messages	C-1
Appendix D - S-Language Programs	D-1
Table of Contents	D-i
Introduction	D-1
Instruction Set	D-10
Arithmetic Operands and Instructions	D-12
Data Movement Operands and Instructions	D-23
Branching Operands and Instructions	D-45
Conditional Branch Operands and Instructions	D-56
Miscellaneous Instructions	D-63
Alphabetic Index	D-68
Appendix E - COBOL Graphics	E-1
Index	Index-1

LIST OF ILLUSTRATIONS

<u>Figure</u>		<u>Page</u>
3-1	COBOL Coding Form	3-2
3-2	Example of Continuation of Words and Literals	3-5
4-1	IDENTIFICATION DIVISION Coding	4-4
5-1	ENVIRONMENT DIVISION Coding	5-18
6-1	Level Number Construction	6-6
6-2	Concept of Level Numbers	6-7
6-3	Coding of Multi-Dimensioned Table	6-13
6-4	Coding of FD and DATA RECORDS	6-18
6-5	Coding of Condition-Name	6-38
6-6	Relationship of Class and Category	6-50
6-7	Permissible Editing Types	6-54
6-8	Examples of RENAMES	6-65
6-9	WORKING-STORAGE SECTION Coding	6-73
7-1	Valid MOVE Statement Combinations	7-65
7-2	PERFORM Statement Varying One Identifier	7-76
7-3	PERFORM Statement Varying Two Identifiers	7-76
7-4	Example of SEARCH Operation Relating to Option 1	7-86
7-5	SET Statement Operand Combinations	7-89
7-6	Coding of PROCEDURE DIVISION	7-107
8-1	Approved Character Fonts	8-22
11-1	Compilation Card Deck	11-1

LIST OF TABLES

<u>Table</u>		<u>Page</u>
6-1	Maximum Value of Integers	6-20
6-2	Recording Modes for Peripheral Devices	6-26
6-3	Editing Symbols and Results	6-55
6-4	Order of Precedence	6-59
6-5	Editing Application of the PICTURE Clause	6-61
7-1	Combination of Symbols in Arithmetic Expressions	7-14
7-2	Relationship of Conditions, Logical Operators, and Truth Values	7-18

List of Tables (Cont)

<u>Tables</u>		<u>Page</u>
7-3	Combinations of Conditions and Logical Operators	7-18
13-1	Status Key Combinations	13-4
13-2	Tag File and Data File Names	13-6
13-3	File Close Type and Disposition	13-12
13-4	Permissible Statements	13-15
D-1	COBOL Program Layout	D-2
D-2	Special Registers	D-2
D-3	Editing Constants and Toggles	D-37
E-1	Graphics and Related Codes	E-2

FOREWORD

BURROUGHS EXTENSIONS TO ANSI 68 COBOL

Programming applications are written in the COBOL language as specified in this B 1800/B 1700 Systems COBOL Reference Manual. The source language herein described is the USA Standard COBOL, X3.23-1968, to which Burroughs extensions have been added. These extensions are marked in this manual by shading of those portions of the COBOL language that are designated as Burroughs extensions.

ACKNOWLEDGEMENT

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

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

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

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

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

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

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

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

SECTION 1

INTRODUCTION

This manual provides a complete description of COBOL (COMMON BUSINESS ORIENTED LANGUAGE) as implemented for use on the Burroughs B 1800/B 1700 system. This concept of COBOL embraces the adoption of the American National Standards Institute (ANSI) 1968.

ADVANTAGES OF COBOL

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

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

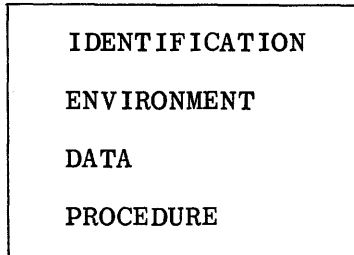
- a. Expeditious means of program implementation.
- b. Accelerated programmer training and simplified retraining requirements.
- c. Reduced conversion costs when changing from a computer of one manufacturer to that of another.
- d. Significant ease of program modification.
- e. Standardized documentation.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

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

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

PROGRAM ORGANIZATION

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



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

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

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

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

A program written in COBOL, called a source program, is accepted as input by the COBOL compiler. The compiler verifies that rules outlined in this manual are satisfied, and translates the source program language into an object program language capable of communicating with the computer and directing it to operate on the desired data. Should source corrections become necessary, appropriate changes can be made and the program recompiled. Thus, the source file always reflects the object program being operationally executed.

SECTION 2

LANGUAGE FORMATION

GENERAL

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

CHARACTER SET

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

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

Characters Used for Words

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

0 - 9
A - Z
- (hyphen)

Punctuation Characters

The following characters may be used for program punctuation:

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

NOTE

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

Characters Used in Editing

The COBOL compiler accepts the following characters in editing:

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

Characters Used in Formulas

The COBOL compiler accepts the following characters in arithmetic expressions:

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

Characters Used in Relations

The COBOL compiler accepts the following characters in conditional relations:

=	equal sign
<	less than symbol
>	greater than symbol

DEFINITION OF WORDS

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

A through Z
0 through 9
- hyphen

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

Types of Words

COBOL contains the following word types:

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

Nouns

Nouns are divided into ten special categories:

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

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

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

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

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

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

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

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

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

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

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

Numeric Literal

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

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

NOTES

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

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

- b. There must be at least one digit in a numeric literal.
- c. The sign of a numeric literal must appear as the leftmost character. If no sign is present, the literal is defined as a positive value.
- d. The decimal point may appear anywhere within the literal except for the rightmost character of a numeric literal. A decimal point within a numeric literal is treated as an implied decimal point. Absence of a decimal point denotes an integer quantity. (An integer is a numeric literal which contains no decimal point.)
- e. A numeric literal used for arithmetic manipulations cannot exceed 160 digits. The following are examples of numeric literals.

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

Non-Numeric Literal

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

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

Literal on Source Program Level

"ACTUAL SALES FIGURE"
"-1234.567"
""LIMITATIONS""
"ANNUAL DUES"
""
"A"B"

Literal Stored by Compiler

ACTUAL SALES FIGURE
-1234.567
"LIMITATIONS"
ANNUAL DUES
"
A"B

NOTE

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

Undigit Literals

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

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

- ZERO Represents the value 0, or one or more of the
- ZEROS character 0, depending on the context.
- ZEROES

- SPACE Represents one or more spaces (blanks).
- SPACES

- HIGH-VALUE Represents the highest internal coding sequence
- HIGH-VALUES (i.e., 999) value. When HIGH-VALUES are moved to a
- signed numeric computational field, the sign will
- be changed to a plus sign.

- LOW-VALUE Represents the lowest internal coding sequence (blanks)
- LOW-VALUES value. When LOW VALUES are moved to a signed numeric
- computational field, zeros will be moved into the field
- and the sign will be changed to a plus.

QUOTE
QUOTES

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

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

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

NOTE

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

Special Registers. The B 1800/B 1700 COBOL compiler provides the following five special PROCEDURE DIVISION register names:

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

Tally

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

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

Today's-Date (Calendar)

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

Today's-Name (Day of Week)

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

Date (Julian)

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

Time

Access to an internal clocking register reflecting the time of day is programmatically available whenever TIME is requested as the sending field of a MOVE statement. The contents of the TIME register will be maintained in hours, minutes, seconds and 10th of seconds. Its format is HHMMSSST. For example, 10:30:51.8 would be stored as 1030518.

Verbs

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

Reserved Words

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

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

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

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

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

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

LANGUAGE DESCRIPTION NOTATION

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

Key Words

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

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

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

Optional Words

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

Generic Terms

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

```
FILE-LIMIT IS integer-1 THRU integer-2
```

Braces

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

Brackets

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

Ellipsis

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

Period

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

SECTION 3

CODING FORM

GENERAL

The format of the COBOL coding form (figure 3-1) has been defined by CODASYL, by ANSI, and by common usage. The B 1800/B 1700 COBOL compiler accepts this standard format. Should program interchange be a major consideration, the user is directed to the ANSI standard.

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

SEQUENCE FIELD (CARD COLUMNS 1-6)

The sequence field may be used to sequence the source program. Normally a numeric sequence is used; however, the B 1800/B 1700 compiler allows any combinations of characters. A warning message is given if there is a sequence error. The B 1800/B 1700 compiler provides for insertion or replacement of card images during compilation, controlled by the sequence field. (See section on "COBOL COMPILER CONTROL," section 11.)

CONTINUATION INDICATOR (COLUMN 7)

Column 7 has several functions as follows:

- a. A \$ symbol in column 7 is used for cards which specify option for compiler operation. (See section 11.)
- b. If column 7 contains an asterisk (*), the rest of the card is considered to be a comment and, hence, is not "compiled" to produce object code.
- c. If column 7 contains a slash (/), the listing, if any, is advanced to channel 1 before printing, and the card is considered to be a comment card.
- d. The letter L followed by a "library-name" entry causes all succeeding source card data to be placed into the COBOL Library File during compilation. Termination of the action takes place when an L card is encountered followed by spaces.

- e. The presence of a hyphen (-) indicates that the last word or literal on the previous card was not complete, and is continued on this card beginning beyond margin A.

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

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

MARGIN A (COLUMNS 8 THRU 11)

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

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

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

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

MARGIN B (COLUMNS 12 THRU 72)

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

RIGHT MARGIN (COLUMN 72)

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

IDENTIFICATION (COLUMNS 73 THRU 80)

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

PUNCTUATION

The following rules of punctuation apply to the writing of COBOL programs for the B 1800/B 1700.

- a. A sentence is terminated by a period followed by a space. A period may not appear within a sentence unless it is within a non-numeric literal or is a decimal point in a numeric literal or PICTURE string.
- b. Two or more names in a series may be separated by a space or by a comma. If used, commas can appear only where allowed.
- c. Semicolons (;) are used only for readability and are never required.
- d. A space must never be embedded in a name; hyphens should be used instead. (A hyphen may not start or terminate a name.) For example:

NET-PAY

SAMPLE CODING

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

SECTION 4

IDENTIFICATION DIVISION

GENERAL

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

IDENTIFICATION DIVISION STRUCTURE

The structure of this division is as follows:

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

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

- a. The IDENTIFICATION DIVISION must begin with the reserved words IDENTIFICATION DIVISION followed by a period.
- b. All paragraph-names within this division must begin in Area A of the coding form.
- c. An entry following a paragraph-name cannot contain periods, with the exception that a period must be present to denote the end of that entry.

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

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

MONITOR

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

The format of this statement is:

```
[ MONITOR [DEPENDING] file-name ([data-name] ... :  
  { ALL  
  {procedure-name...} ) ] ]
```

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

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

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

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

Whenever a MONITORED elementary data-name is encountered as the receiving field in a MOVE or arithmetic statement, the data-name and its current value are listed. Refer to the REF option described under \$OPTION CONTROL CARD in Section 11 of this manual.

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

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

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

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

CODING THE IDENTIFICATION DIVISION

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

ENVIRONMENT DIVISION

GENERAL

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

ENVIRONMENT DIVISION ORGANIZATION

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

ENVIRONMENT DIVISION STRUCTURE

The structure of this division is as follows:

<u>ENVIRONMENT DIVISION.</u>
[<u>CONFIGURATION SECTION.</u>]
[<u>SOURCE-COMPUTER . . .</u>]
[<u>OBJECT-COMPUTER . . .</u>]
[<u>SPECIAL-NAMES . . .</u>]
[<u>INPUT-OUTPUT SECTION.</u>]
[<u>FILE-CONTROL . . .</u>]
[<u>I-O-CONTROL . . .</u>]

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

- a. The ENVIRONMENT DIVISION must begin with the reserved words ENVIRONMENT DIVISION followed by a period.
- b. All entries other than the ENVIRONMENT DIVISION source line are optional but, when used, they must begin in Area A of the coding form.

CONFIGURATION SECTION

CONFIGURATION SECTION

The CONFIGURATION SECTION contains information concerning the system to be used for program compilation (SOURCE-COMPUTER), the system to be used for program execution (OBJECT-COMPUTER), and the special-names paragraph, which relates hardware names used by the B 1800/B 1700 COBOL compiler to the mnemonic-names in the source program.

SOURCE-COMPUTER

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

The format of this paragraph has the following two options:

Option 1:

```
SOURCE-COMPUTER. COPY library-name
```

```
[ , REPLACING word-1 BY word-2  
  [ , word-3 BY word-4 ] ... ] .
```

Option 2:

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

This paragraph is for documentation only.

OBJECT-COMPUTER

OBJECT-COMPUTER

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

The format of this paragraph has the following two options:

Option 1:

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

Option 2:

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

Priority numbers used in the SEGMENT-LIMIT clause must be positive integers with a value from 1 to 49. The SEGMENT-LIMIT clause specifies the limit of the fixed segment for sections numbered from 0 to 49. See SEGMENT CLASSIFICATION, PROGRAM SEGMENTS, and PRIORITY NUMBERS.

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

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

When the MEMORY SIZE clause is used without the SORT option, the compiler will assign the amount of memory between the base and limit register to reflect the size of integer-1.

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

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

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

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

When data segmentation is specified:

- a. Each file record is placed in a separate segment.
- b. The compiler assigns as the overlayable data area (dynamic memory) the largest memory size required for all of the overlayable data segments referenced in any one operation. This overlayable data area may be expanded by using the MEMORY SIZE clause without the SORT option, or by using a MEMORY control card at execution time.
- c. All 77 level entries are placed in data segment 0 (zero).
- d. A record (01 level) that is greater in length than the DATA SEGMENT-LIMIT is placed in a segment by itself, and is not split between segments. If DATA SEGMENT-LIMIT has been declared larger than the defined record size, the record will reside in the declared amount of memory, as well as succeeding records to the limit of the defined segment.

SPECIAL-NAMES

SPECIAL-NAMES

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

The format of this paragraph has the following two options:

Option 1:

```
SPECIAL-NAMES. COPY library-name
```

```
[REPLACING word-1 BY word-2  
[, word-3 BY word-4] ... ] .
```

Option 2:

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

```
[[, implementor-name IS mnemonic-name] . . . ]  
[, DECIMAL-POINT IS COMMA] .
```

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

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

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

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

If implementor-name is specified, it must be one of the allowable B 1800/ B 1700 COBOL hardware-names which may be specified in FILE-CONTROL paragraph For example:

```
PUNCH IS CARD-PUNCH-EBCDIC
```

SPECIAL-NAMES

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

The SPECIAL-NAMES paragraph statement ends with a period as a delimiter.

Periods between clauses are not allowed.

INPUT-OUTPUT SECTION

INPUT-OUTPUT SECTION

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

FILE-CONTROL

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

The format of this paragraph has the following three options:

Option 1:

```

FILE-CONTROL.    COPY    library-name

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

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

```

Option 2:

```

FILE-CONTROL.

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

[ BY [integer-1 [INC-EU]] [WORK] [ALL-AT-OPEN] [SINGLE] [SAVE]
  [ [ [NO] BACKUP [TAPE] ] ] [FORM] [FOR MULTIPLE REEL]
  [ [OR] BACKUP [DISK] ] ] ]

[ ,RESERVE { NO integer-2 } [ALTERNATE [ [AREA ] ] ] ]

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

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

[ ,ACCESS MODE IS { RANDOM
  { SEQUENTIAL } ]

[ ,ACTUAL KEY IS data-name-3 ]

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

```

FILE-CONTROL

Option 3:

```
FILE-CONTROL.  
SELECT sort-file-name ASSIGN TO [ integer-3  
data-name-4 ] SORT { TAPES  
                  TAPE  
                  DISK } .
```

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

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

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

The ASSIGN clause must be used in order for the MCP to associate the file with a hardware peripheral device. The allowable entries for hardware-name-1 are:

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

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

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

Use of the FORM option with printer or punch files will cause the program to halt and an MCP message to be displayed declaring the need for special forms to be loaded in the Line Printer or Card Punch, as applicable. When the file is CLOSED, the device will be SAVED by the MCP to prevent another job from printing on that same printer before the operator has an opportunity to remove the special forms.

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

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

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

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

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

The SAVE option will cause the file to be CLOSED WITH LOCK by the MCP if the file is still OPENED OUTPUT at the time the program is terminated by other than a CLEAR/START (I.E., normal EOJ or DS/DP). If a disk file of the same name already exists, the MCP will display the "Duplicate file on disk" message, and operator action will be required. If the SAVE option is omitted, the standard MCP action will be invoked; that is, the file will automatically be CLOSED (but not LOCKed) if it is still OPEN at program termination.

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

The RESERVE clause has an additional function when used for QUEUE files. In this case, it is used to specify the number of messages to be contained in memory. If more messages are allowed in the QUEUE than are allowed in memory, the remaining messages will be placed on disk. (See the discussion of QUEUES in Section 10).

FILE-CONTROL

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

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

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

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

The FILE-LIMIT clause specifies the following:

- a. For SEQUENTIAL access, logical records are obtained from, or placed sequentially in, the disk storage file by the implicit progression from segment to segment. The AT END imperative statement of a READ statement is executed when the logical end of the last segment of the file is reached and an attempt is made to READ another record. The INVALID KEY clause of a WRITE statement is executed when the end of the last segment is reached and an attempt is made to WRITE another record. The END option specifies that the compiler is to determine the upper limit of an existing file. No ACTUAL KEY entry is necessary for the SEQUENTIAL mode.
- b. For RANDOM access, logical records are obtained from, or placed randomly in, the disk storage file within the specified FILE-LIMIT. The contents of ACTUAL KEY not within the specified limit will cause the execution of the INVALID KEY branch in the READ and the WRITE statements. The ACTUAL KEY entry must be specified.

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

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

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

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

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

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

Values of the ACTUAL KEY data-name-3 are controlled by the programmer, including any execution of the USE FOR KEY CONVERSION statement. The value may range from 1 to n, where n equals the number of records in the file or as reflected by the FILE-LIMITS clause. The ACTUAL KEY signifies the relative position of a record within the file and is equated to a data-name at any level which is defined with a PICTURE OF 9(8) COMPUTATIONAL. ACTUAL KEY is not used for ACCESS MODE SEQUENTIAL files.

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

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

```

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

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

MSG-TYPE must be set by the application program to a value of 0 for a WRITE. For a READ, MSG-TYPE is set to a value of 1.

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

The PROCESSING MODE IS SEQUENTIAL clause is for documentation only.

All integers must be of positive values.

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

The sort-file-name in Option 3 is the SD level file-name to be used by the SORT or MERGE verbs. Option 3 may also be used to specify the number of tapes to be used as work files for TAPE SORT.

If data-name-4 is specified, it must be defined as an integer. If this statement specifies tape work files and integer-3/data-name-4 is omitted, the default number of work tapes, three (3), will be used. Three is also the minimum allowable number of work tapes. The maximum number of work tapes which may be specified is (8).

The BY clause, when used with integer-1, directs the file to the drive or EU specified by integer-1. The drive must be a systems disk. Integer-1 must be a positive value from 0 through 15. The INC-EU option causes each area of the file to begin on the next system drive.

I-O-CONTROL

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

The construct of this paragraph is:

Option 1:

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

Option 2:

```
I-O-CONTROL.

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

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

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

I-O-CONTROL

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

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

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

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

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

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

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

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

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

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

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

The POSITION clause is for documentation only.

CODING THE ENVIRONMENT DIVISION

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

BURROUGHS COBOL CODING FORM
 ADDITIONS, DELETIONS AND CHANGES

5-1-68

PROGRAM		ENVIRONMENT DIVISION Coding										COBOL DIVISION		PAGE	OF	
PROGRAMMER		KEVIN										DATE		IDENT	73	80
PAGE NO.	LINE NO.	A	B													Z
1	3 4 6 7	8	11 12	22		32		42		52		62		72		
		ENVIRONMENT DIVISION.														
		CONFIGURATION SECTION.														
		SOURCE-COMPUTER. B-1700.														
		OBJECT-COMPUTER. B-1700 SEGMENT-LIMIT IS 110.														
		SPECIAL-NAMES. DECIMAL-POINT IS COMMA.														
		INPUT-OUTPUT SECTION.														
		FILE-CONTROL. SELECT OPTIONAL DAILY-TAPE ASSIGN TO TAPE.														
		SELECT MASTER-FILE ASSIGN TO DISK														
		FILE-LIMIT IS 1 THRU 1000														
		ACCESS MODE RANDOM ACTUAL KEY DISK-CONTROL.														
		SELECT MASTER-TAPE ASSIGN TO TAPE.														
		SELECT ERROR-TAPE ASSIGN TO TAPE RESERVE NO ALTERNATE AREAS.														
		SELECT DETAIL-CHANGES-FILE ASSIGN TO DISK.														
		SELECT SUMMARY-FILE ASSIGN TO DISK.														
		I-O-CONTROL-														
		SAME RECORD AREA FOR DAILY-TAPE, ERROR-TAPE														
		MULTIPLE FILE DISKPACK "MULTIPAK" CONTAINS														
		MASTER-FILE DETAIL-CHANGES-FILE SUMMARY-FILE.														

Figure 5-1. ENVIRONMENT DIVISION Coding

SECTION 6

DATA DIVISION

GENERAL

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

- a. Data which is contained in files and which enters or leaves the internal memory of the computer from a specified area or areas.
- b. Data which is developed internally and placed into intermediate storage, or placed into a specific format for output purposes.
- c. Constants which are defined by the programmer.

DATA DIVISION ORGANIZATION

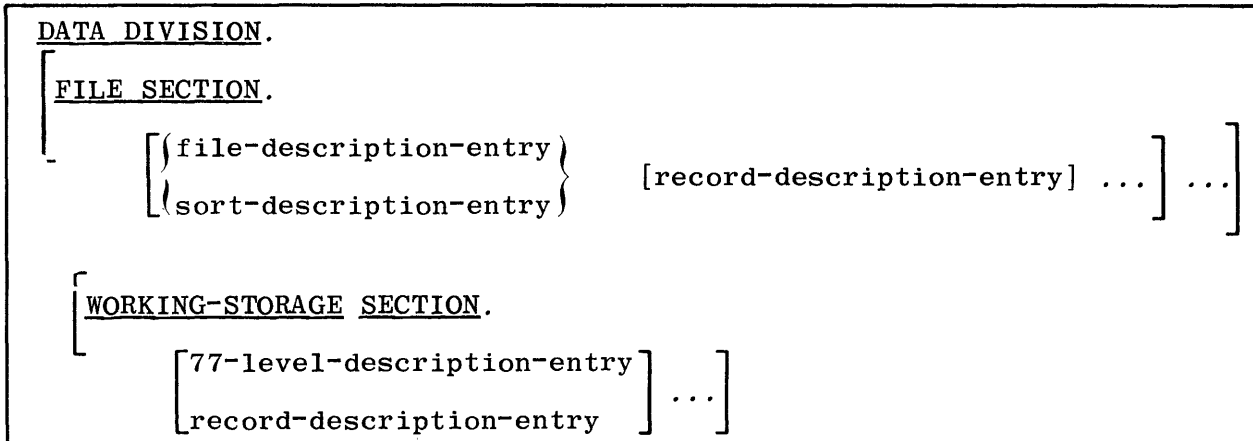
The DATA DIVISION is subdivided into two sections:

- a. The FILE SECTION defines the contents of data files which are to be created or used by an external medium. Each file is defined by a file description, followed by a record description or a series of file-related record descriptions.
- b. The WORKING-STORAGE SECTION describes records, constants, and non-contiguous data items which are not part of an external data field, but which are developed and processed internally.

DATA DIVISION STRUCTURE

DATA DIVISION STRUCTURE

The general structure of the DATA DIVISION is as follows:



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

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

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

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

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

FILE AND RECORD CONCEPTS

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

Physical Aspects of a File

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

- a. The mode in which the data file is recorded on the external medium.
- b. The grouping of logical records within the physical limitations of the file medium.
- c. The means by which the file can be identified.

Conceptual Characteristics of a File

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

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

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

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

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

FILE AND RECORD CONCEPTS

Record Concepts

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

Example:

01 ITEM-ONE PICTURE IS X(6).

The maximum size of a record description (i.e., the sum of the maximum sizes of all the items subordinate to an 01 level item) is restricted to 65,535 bits (8K bytes).

LEVEL NUMBERS CONCEPT

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

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

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

In COBOL, the item relationship is specified by the use of a series of level numbers. These numbers must not contain more than two characters, and may range from 1 thru 49. (Special level numbers of 66, 77, and 88 are discussed later.)

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

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

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

BURROUGHS COBOL CODING FORM
ADDITIONS, DELETIONS AND CHANGES

LEVEL NUMBERS CONCEPT

PROGRAM		LEVEL NUMBER CONSTRUCTION										COBOL DIVISION		PAGE 1 OF 1	
PROGRAMMER		BARBRA										DATE		IDENT 73 80	
PAGE NO.	LINE NO.	A	B									Z			
1	3 4 6 7	8	11 12	22	32	42	52	62	72	82					
		01		EMPLOYEE-INFO.											
		03		EMP-NO			PIC 9(5).								
		03		EMP-COST-CENTER			PIC 99.								
		03		EMP-NAME.											
		05		EMP-LAST-NAME			PIC X(13).								
		05		EMP-FIRST-INITIAL			PIC X.								
		05		EMP-M-INITIAL			PIC X.								
		03		EMP-ANNUAL-SALARY			PIC 9(6)V99.								
		03		EMP-DT-HIRED.											
		05		EMP-H-MONTH			PIC 99.								
		05		EMP-H-DAY			PIC 99.								
		05		EMP-H-YEAR			PIC 99.								
		03		EMP-PAY-DATA.											
		05		EMP-GROSS			PIC 9(6)V99.								
		05		EMP-DDEDUCTIONS.											
		09		EMP-INSURANCE.											
		11		EMP-HOSPITAL			PIC 9(4)V99.								
		11		EMP-LIFE			PIC 9(4)V99.								
		09		EMP-TAXES.											
		11		EMP-FICA			PIC 9(4)V99.								
		11		EMP-STATE-TAX			PIC 9(4)V99.								
		11		EMP-WITHHOLDING			PIC 9999V9(2).								
		03		EMP-LAST-REVIEW.											
		04		EMP-L-MONTH			PIC 99.								
		04		EMP-L-DAY			PIC 99.								

Figure 6-1. Level Number Construction

LEVEL NUMBERS CONCEPT

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

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

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

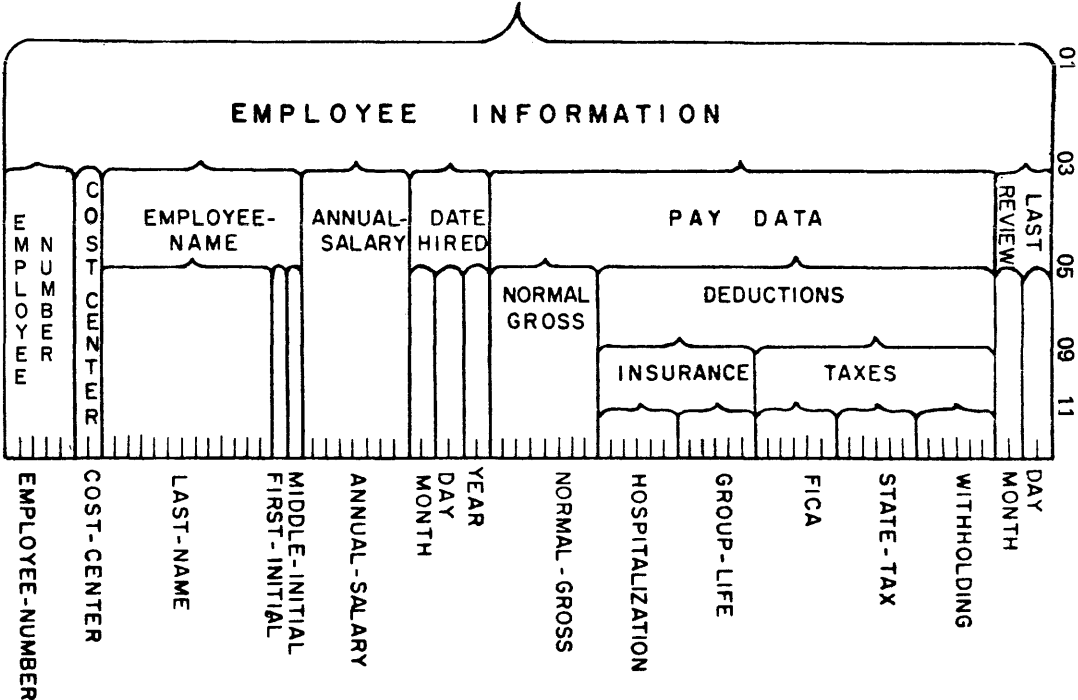


Figure 6-2. Concept of Level Numbers

QUALIFICATION

QUALIFICATION

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

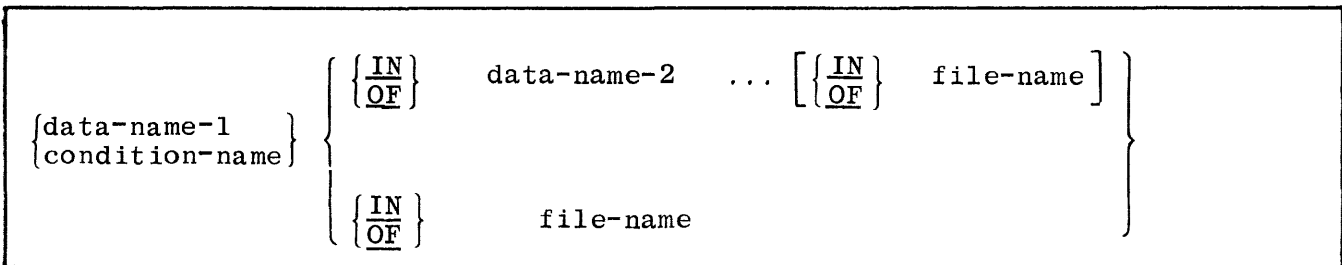
The hierarchy of qualification is as follows: names associated with a level indicator are the most significant; then names associated with level-number 01, then those names associated with level-number 02, ... , 49. A section-name is the highest (and the only) qualifier available for a paragraph-name.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

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

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

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

Option 1:



Option 2:

paragraph-name	$\left\{ \begin{array}{c} \text{OF} \\ \text{IN} \end{array} \right\}$	section-name
----------------	--	--------------

The rules for qualification are as follows:

- a. Each qualifier must be of a successively higher level and within the same hierarchy as the name it qualifies.
- b. The same name must not appear at two levels in a hierarchy so that the name would appear to qualify itself.
- c. If a data-name or a condition-name is assigned to more than one data item in a source program, the data-name or condition-name must be qualified each time it is referred to in the PROCEDURE DIVISION, ENVIRONMENT DIVISION, and DATA DIVISION (except REDEFINES where, by definition, qualification is unnecessary).
- d. A paragraph-name must not be duplicated within a section. When a paragraph-name is qualified by a section-name, the word SECTION must not appear. A paragraph-name need not be qualified when referenced within its own section.
- e. A data-name cannot be subscripted or indexed when it is being used as a qualifier.
- f. A name can be qualified, even though it does not need qualification; if there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used.

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

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

QUALIFICATION

Example:

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

TABLES

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

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

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

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

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

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

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

Example:

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

SUBSCRIPTING

SUBSCRIPTING

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

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

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

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

The general construct for subscripting is:

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

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

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

BURROUGHS COBOL CODING FORM
ADDITIONS, DELETIONS AND CHANGES

PROGRAM MULTI-DIMENSIONED TABLE							COBOL DIVISION			PAGE 1 OF 1			
PROGRAMMER BEV							DATE			IDENT 73 80			
PAGE NO.	LINE NO.	A	B								Z		
1	3	4	6	7	8	11	12	22	32	42	52	62	72
					01								
					05								
					10								
					10								
					15								
					15								
					20								

SUBSCRIBING

Figure 6-3. Coding of Multi-Dimensioned Table

INDEXING

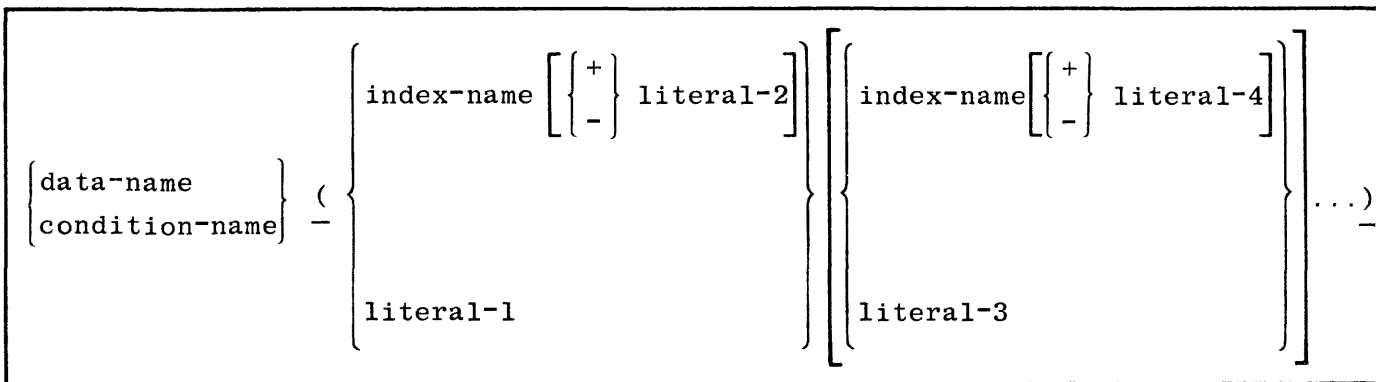
INDEXING

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

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

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

The general construct for indexing is:



FILE SECTION

FILE SECTION

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

File Description

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

The construct of this sentence contains four options:

Option 1:

```

FD      file-name-1  COPY      library-name

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

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

```

Option 2:

```

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

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

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

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

[ ; { VA
      VALUE } OF { ID
                   IDENTIFICATION } IS { [literal-1/] literal-2 [ / [literal-3] ]
                                           data-name-1 }
      [ SAVE-FACTOR IS integer-7 ] ]

[ ; { VA
      VALUE } OF Q-MAX-MESSAGES IS integer-8 ]

[ ; LABEL { RECORD IS
             RECORDS ARE } { OMITTED
                               STANDARD } [ data-name-2 [, data-name-3... ] ] ]

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

```


Option 3:

```

SD sort-file-name COPY library-name

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

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

```

Option 4;

```

SD sort-file-name

[ ; FILE CONTAINS integer-1 [ BY integer-2 ] RECORDS ]
[ ; BLOCK CONTAINS [ integer-3 TO ] integer-4 [ RECORDS
                                                    CHARACTERS ] ]
[ ; RECORD CONTAINS [ integer-5 TO ] integer-6 CHARACTERS ]
[ ; DATA { RECORD IS
            RECORDS ARE } data-name-1 [ data-name-2 ] ... ]

```

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

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

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

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

NOTE

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

BLOCK

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

The construct of this clause is:

BLOCK CONTAINS [integer-1 TO] integer-2 [RECORDS
CHARACTERS]

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

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

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

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

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

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

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

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

NOTE

If the BLOCK clause is omitted for an input file, and the file is assigned to a disk file or a labeled tape that was created on a B 1800/B 1700, the DEFAULT option will be set. This causes the blocking factor specified in the disk file header or tape label to be used.

BLOCK

Table 6-1. Maximum Value of Integers

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

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

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

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

This clause may be omitted for unblocked files.

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

DATA RECORDS

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

The construct of this clause is:

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

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

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

FILE CONTAINS

FILE CONTAINS

The function of this clause is to indicate the number of logical records in a file.

The construct of this clause is:

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

The indicated integers must be positive values.

Integer-1 may not exceed 105 when present.

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

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

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

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

The WITH HEADERS option may be used with REMOTE files. This option specifies that the Message Header for each message should be appended to the beginning of the message by the Network Controller. When a program OPENS a file with this option, that program is identified as an MCS. The information in the

Message Header may be used by the programmer to do message switching, or other functions of a Message Control System. When HEADERS are requested, the user must include the additional space in the records to contain the Message Headers. Additional information regarding the formats of the Message Headers may be found in the NDL Reference Manual.

NOTE

If the FILE CONTAINS clause is omitted,
the default value will be 1 area of
500 blocks.

LABEL

LABEL

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

The construct for this clause is:

$$\underline{\text{LABEL}} \left\{ \begin{array}{l} \underline{\text{RECORD IS}} \\ \underline{\text{RECORDS ARE}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{OMITTED}} \\ \underline{\text{STANDARD}} \\ \underline{\text{data-name-1}} [, \underline{\text{data-name-2}}] \dots \end{array} \right\}$$

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

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

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

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

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

The B 1800/B 1700 standard label for tape is in a format compatible with the proposed USASI standard label for information exchange.

RECORD

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

The construct of this clause is:

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

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

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

If only integer-2 is specified the compiler will treat the clause as being documentation only. The record size will be determined by the structure of the largest record description. If the largest record description is not equal to the value of integer-2, a warning message will be produced.

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

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

Variable-length records may only be contained on disk or magnetic tape files. Variable-length files are also limited to sequential access, and may only be OPENed INPUT or OUTPUT.

RECORDING MODE

RECORDING MODE

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

The construct for this clause is:

RECORDING MODE IS { STANDARD
NON-STANDARD
ASCII }

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

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

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

The records of a file which is specified RECORDING MODE ASCII are USAGE ASCII by default.

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

Table 6-2. Recording Modes for Peripheral Devices

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

VALUE OF ID

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

The construct of this clause is:

$$\left\{ \begin{array}{l} \text{VALUE} \\ \text{VA} \end{array} \right\} \text{ OF } \left\{ \begin{array}{l} \text{IDENTIFICATION} \\ \text{ID} \end{array} \right\} \text{ IS } \left\{ \begin{array}{l} \text{[literal-1/]} \text{ literal-2 } \text{[/[literal-3]]} \\ \text{data-name-1} \end{array} \right\}$$

[SAVE-FACTOR IS integer-1]

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

Example:

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

To make them unique:

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

The multiple file tape name will be taken from either the MULTIPLE FILE clause in the I-O-CONTROL paragraph; the value of literal-2 when literal-2 is followed by a slash and literal-3; or the second 10 characters of data-name-1. The file name of a multiple file tape file will be taken from the contents of literal-2 when only one literal is specified; from literal-3 if both literal-2 and literal-3 are specified; or from the last 10 characters of data-name-1. The file name for a single file tape will be taken from the contents of literal-2, or from the second 10 characters of data-name-1.

If literals are used, and contain non-blank characters, then the first character in the literal must be a non-blank character.

All non-disk files, are limited to a maximum of two names.

VALUE OF ID

The pack-id name of a disk file will be taken either from the multi-file clause in the I-O-CONTROL paragraph, or from the value of literal-1. The main directory (family) name will be taken from literal-2 (in the case of systems disk or if I-O-CONTROL is used to specify user disk), from literal-2 (in the case of user disk without I-O-CONTROL or if literal-2 is followed by a slash (/)). The sub-directory entry (file-name) will be taken from the value of literal-3. Literal-3 cannot be used when literal-1 and literal-2 are both blank. When using the literal option, if three literals are used, they represent pack-id, main directory (family), and sub-directory (file-name), respectively. If two literals are used they represent main directory and sub-directory. If only one literal is used it represents the main directory entry.

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

Examples:

```

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

```

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

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

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

Examples:

01 FILE-IDENTIFICATION.

03	PACK-ID	PC	X(10)	VA	"USER10000000".
03	MAIN-DIRECTORY	PC	X(10)	VA	"PAYROLL0000".
03	SUB-DIRECTORY	PC	X(10)	VA	"DEDUCTS0000".

01 DATA-NAME-1.

03	PACK-ID	PC	X(10)	VA	"WORKPACK10".
03	MAIN-DIRECTORY	PC	X(10)	VA	"TRANS000000".
03	SUB-DIRECTORY	PC	X(10)	VA	SPACES.

01 FILE-ID.

03	PACK-ID	PC	X(10)	VA	SPACES.
03	MAIN-DIRECTORY	PC	X(10)	VA	"PAYROLL0000".
03	SUB-DIRECTORY	PC	X(10)	VA	"MASTER0000".

01 VA-NAME.

03	PACK-ID	PC	X(10)	VA	SPACES.
03	MAIN-DIRECTORY	PC	X(10)	VA	"ITEMS000000".
03	SUB-DIRECTORY	PC	X(10)	VA	SPACES.

01 SOME-DATA-NAME.

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

NOTE

Names must be moved in prior to OPEN.

VALUE OF ID

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

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

If SAVE-FACTOR is not specified, tapes are automatically assigned a SAVE-FACTOR of zero days.

VALUE OF Q-MAX-MESSAGES

The function of this clause is to specify the maximum number of messages allowed in a QUEUE.

The construct of this clause is:

$\left. \begin{array}{c} \text{VA} \\ \hline \text{VALUE} \end{array} \right\}$ OF Q-MAX-MESSAGES IS integer-8

Integer-8 is limited to three digits in length with values ranging from 001 through 255 inclusive.

If this clause is omitted, the value of Q-MAX-MESSAGES is set to 2 by the MCP when the file is OPENed.

RECORD DESCRIPTION

RECORD DESCRIPTION

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

The construct of these entries contain the following four options:

Option 1:

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

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

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

Option 2:

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

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

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

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


RECORD DESCRIPTION

```

    [ { ASCENDING }      KEY IS data-name-4  [ ,data-name-5 ] ... ]
    [ { DESCENDING }
    [ INDEXED BY index-name-1  [ ,index-name-2 ] ... ]
    [ ; { SY                { LEFT } ]
    [ { SYNC                { RIGHT } ]
    [ { SYNCHRONIZED }
    [ ; { IS                RIGHT ]
    [ { JUST                ]
    [ { JUSTIFIED }
    [ ; { BZ                ]
    [ { BLANK WHEN ZERO } ]
    [ ; { VA                { IS } literal-1 ]
    [ { VALUE } { ARE } ] .
  
```

Option 3:

```

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

Option 4:

```

    88 condition-name { VA } { IS } literal-1 [ { THRU } literal-2 ]
    [ ; { VALUE } { ARE }
    [ ,literal-3 [ { THRU } literal-4 ] ] ... .
    [ { THROUGH }
  
```

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

The record description must be terminated by a period.

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

RECORD DESCRIPTION

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

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

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

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

The SYNCHRONIZED clause is for documentation only.

BLANK WHEN ZERO

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

The construct of this clause is:

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

BLANK WHEN ZERO may be abbreviated BZ.

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

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

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

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

CONDITION-NAME

CONDITION-NAME

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

The construct of this clause is:

88 condition-name { VA / VALUE } [IS / ARE] literal-1 [{ THRU / THROUGH } literal-2]
[,literal-3 [{ THRU / THROUGH } literal-4]] ...

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

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

- a. Another condition-name.
- b. A level 66 item.
- c. A group containing items with descriptions including JUSTIFIED, or USAGE (other than USAGE IS DISPLAY).
- d. An index data-item.
- e. A data-item with the name FILLER.

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

- a. If reference to a conditional variable requires subscripting, then references to its condition-names also require subscripting.
- b. A conditional variable may be used as a qualifier for any of its condition-names.
- c. Condition-names can only appear in conditional statements.
- d. Whenever the THRU phrase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.

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

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

IF JUNE GO TO

which is logically equivalent to the statement:

IF THIS-YEAR IS EQUAL TO 06 GO TO

DATA-NAME

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

The construct of this clause is:

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

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

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

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

JUSTIFIED

JUSTIFIED

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

The format for the JUSTIFIED clause is as follows:

$\left. \begin{array}{l} \text{JUSTIFIED} \\ \text{JUST} \\ \text{JS} \end{array} \right\} \text{RIGHT}$

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

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

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

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

Example:

<u>SENDING</u>	<u>RECEIVING</u>												
PIC X(5) <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 5px;">A</td><td style="padding: 2px 5px;">1</td><td style="padding: 2px 5px;">2</td><td style="padding: 2px 5px;">3</td><td style="padding: 2px 5px;">C</td></tr></table>	A	1	2	3	C	PIC X(7) <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 5px;"> </td><td style="padding: 2px 5px;"> </td><td style="padding: 2px 5px;">A</td><td style="padding: 2px 5px;">1</td><td style="padding: 2px 5px;">2</td><td style="padding: 2px 5px;">3</td><td style="padding: 2px 5px;">C</td></tr></table>			A	1	2	3	C
A	1	2	3	C									
		A	1	2	3	C							

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

Example:

<u>SENDING</u>	<u>RECEIVING</u>												
PIC X(7) <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 5px;">A</td><td style="padding: 2px 5px;">1</td><td style="padding: 2px 5px;">2</td><td style="padding: 2px 5px;">3</td><td style="padding: 2px 5px;">C</td><td style="padding: 2px 5px;">D</td><td style="padding: 2px 5px;">E</td></tr></table>	A	1	2	3	C	D	E	PIC X(5) <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 5px;">2</td><td style="padding: 2px 5px;">3</td><td style="padding: 2px 5px;">C</td><td style="padding: 2px 5px;">D</td><td style="padding: 2px 5px;">E</td></tr></table>	2	3	C	D	E
A	1	2	3	C	D	E							
2	3	C	D	E									

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

JUSTIFIED

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

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

LEVEL-NUMBER

LEVEL-NUMBER

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

The construct of this clause is:

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

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

Level-numbers may be as follows:

- a. 01 to 49 - record description and WORKING-STORAGE entries.
- b. 66 - RENAMES clause used as a record description or WORKING-STORAGE entry.
- c. 77 - applicable to WORKING-STORAGE only as non-contiguous items and must precede all other level-numbers.
- d. 88 - condition-name clause used as a record description or WORKING-STORAGE entry.

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

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

OCCURS

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

The construct for this clause has the following two options:

Option 1:

```

{OC
OCCURS} integer-2 TIMES

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

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

Option 2:

```

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

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

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

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

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

Data-name-2 must either be the name of the entry containing the OCCURS clause or the name of an entry subordinate to the entry containing the OCCURS clause.

Data-name-3, etc., must be the name of an entry subordinate to the group item which is the subject of this entry.

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

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

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

OCCURS

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

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

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

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

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

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

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

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

OCCURS

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

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

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

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

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

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

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

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

OCCURS

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

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

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

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

The ASCENDING/DESCENDING KEY option is for documentation only.

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

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

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

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

- a. The indexes should be used only when referencing the associated row.
- b. All "assumed" references are to the first index-name in a series. Others in the series are affected only during an explicit reference.

OCCURS

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

Example:

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

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

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

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

The COBOL compiler will assign the construction of a PC S9(6) COMPUTATIONAL area for each index-data-name specified.

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

PICTURE

PICTURE

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

The general construct for the PICTURE clause is as follows:

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

The following are rules for the PICTURE clause:

- a. A PICTURE clause can only be used at the elementary item level.
- b. A character-string consists of certain allowable combinations of characters in the COBOL character set used as symbols. The allowable combinations determine the category of the elementary item.
- c. The maximum number of symbols allowed in the character-string is 30. When an unsigned integer enclosed in parentheses immediately follows a symbol, the integer specifies the number of consecutive occurrences of that symbol. This may not be used for those symbols limited to one occurrence per picture.
- d. A PICTURE clause must appear in every elementary item except those items whose USAGE is declared as INDEX.

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

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

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

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

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

Categories of Data

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

These categories are described as follows:

ALPHABETIC

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

NUMERIC

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

ALPHANUMERIC

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

ALPHANUMERIC EDITED

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

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

NUMERIC EDITED

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

Classes of Data

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

PICTURE

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

LEVEL OF ITEM	CLASS	CATEGORY
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric-edited Alphanumeric-edited Alphanumeric
Group	Alphanumeric	Alphanumeric

Figure 6-6. Relationship of Class and Category

Function of the Editing Symbols

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

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

- A The symbol A in the character-string represents a character position which can contain only a letter of the alphabet or a space.
- B Each symbol B in the character-string represents a character position into which the space character will be inserted.
- J The symbol J indicates that an operational sign appears in the zone digit of the least-significant byte for DISPLAY items. If USAGE has been indicated as CMP, then J takes on the same function as an S. J is not allowed for CMP-3. J is not counted in the size for DISPLAY but is counted in CMP. Only one operational sign may be present in each PICTURE. J and S are mutually exclusive. See the S sign discussion for the exact bit configuration of signs.

NOTE

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

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

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

NOTE

Any value other than D will be assumed positive.

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

PICTURE

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

- V The letter V is used in a character-string to indicate the location of the assumed decimal point and may only appear once in a character-string. The V does not represent a character position and, therefore, is not counted in the length of the elementary item. When the assumed decimal point is to the right of the rightmost symbol in the string, the V is redundant.
- X Each letter X in the character-string is used to represent a character position which contains any allowable character from the computer's character set.
- Z Each letter Z in a character-string may only be used to represent the leftmost leading numeric character positions which will be replaced by a space character when the contents of the character position is zero. Each Z is counted in the length of the item. Zero suppression is terminated with the first non-zero numeric character in the data. Insertion characters are also replaced by spaces while suppression is in effect. Z can also appear to the right of J, when the J symbol is used to reinitiate zero suppression. For additional information on zero suppression, see the BLANK WHEN ZERO clause.
- 9 Each 9 in the character-string represents a character position which contains a numeral and is counted in the length of the item. If USAGE is explicitly or implicitly DISPLAY, the data will be operated on as 8-bit (BYTE) characters. If USAGE is CMP, it will be operated on as 4-bit digits.
- 0 Each 0 (zero) in the character-string represents a character position into which the numeral zero will be inserted. When that item is receiving field, the 0 is counted in the length of the item.

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

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

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

- + } The symbols +, -, CR, and DB are used as editing sign control symbols.
 - } When used, they represent the character position(s) into which the
 - CR } editing sign control symbol will be placed. The symbols are mutually
 - DB } exclusive in any one character-string, and each character used in the
- symbol is counted in determining the length of the data-item. (Note that the symbols CR and DB are two character symbols, and any other use of C or D constitutes an error.)
- * Each * symbol in the character-string represents a leading numeric character position into which an asterisk will be placed when the contents of that position is zero. Each * is counted in the length of the item. Asterisk replacement is disabled when the first non-zero character is encountered, or when the decimal point (implicit or explicit) is reached. When the PICTURE character string specifies only asterisks (*), and the value of the item is zero, the entire output item will consist of asterisks and the decimal point, if present. BLANK WHEN ZERO does not override the insertion of asterisks.
 - \$ The currency symbol (\$) in the character-string represents a character position into which a currency symbol is to be placed. The currency symbol in a character-string is represented by either the dollar sign (\$) symbol or by the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the length of the item. (Copy deleted.)

NOTE

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

Example

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

PICTURE

Editing Rules

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

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

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

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

Figure 6-7. Permissible Editing Types

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

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

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

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

PICTURE

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

Fixed Insertion Editing. The currency sign (\$) and the editing sign control symbols "+", "-", CR, and DB are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE character-string. When the symbols CR or DB are used, they represent two character positions in determining the length of the item, and they must represent the rightmost character positions that are counted in the size of the item.

The symbol "+" or "-", when used, must be the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of the item except that it can be preceded by either a + or a - symbol. Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character-string. Depending upon the value of the data item, editing sign control symbols produce the results indicated in table 6-3.

Table 6-3. Editing Symbols and Results

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

PICTURE

Floating Insertion Editing. The currency symbol and editing sign control symbols + or - are the insertion characters, and they are mutually exclusive as floating insertion characters in a given PICTURE character-string.

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

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

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

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

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

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

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

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

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates at the first 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, the entire data item will be spaces if the suppression symbol is Z or all asterisks (*), except for the actual decimal point, if the suppression symbol is *.

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

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

PICTURE

Precedence of Symbols

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

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

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

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

Table 6-4. Order of Precedence

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

PICTURE

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

ALPHABETIC ITEMS;

AA

A(25)

ALPHANUMERIC ITEMS:

XX

X(15)

A(5)9(4)

99A99XX

NUMERIC ITEMS:

9

99999

9V99

S99V99

999PPP

J99

EDITED NUMERIC ITEMS (CLASS IS ALPHANUMERIC):

9.99

ZZZZZ

\$\$.99CR

B(4)9

\$** , ** .99

-----9 ("-" IS A MINUS SIGN)

++ , ++ .999

\$** , ** .99DB

999,999

(Copy deleted.)

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

Table 6-5. Editing Application of the PICTURE Clause

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

REDEFINES

REDEFINES

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

The construct of the REDEFINES clause is:

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

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

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

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

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

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

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

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

Data-name-2 need not be qualified.

REDEFINES

An example of REDEFINES entries follows:

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

RENAMES

RENAMES

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

The construct of this clause is:

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

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

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

When data-name-3 is specified, data-name-1 is a group item which includes all elementary items starting with data-name-2 (if data-name-2 is an elementary item) or the first elementary item in data-name-2 (if data-name-2 is a group item), and concluding with data-name-3 (if data-name-3 is an elementary item) or the last elementary item in data-name-3 (if data-name-3 is a group item).

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

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

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

RENAMES

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

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

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

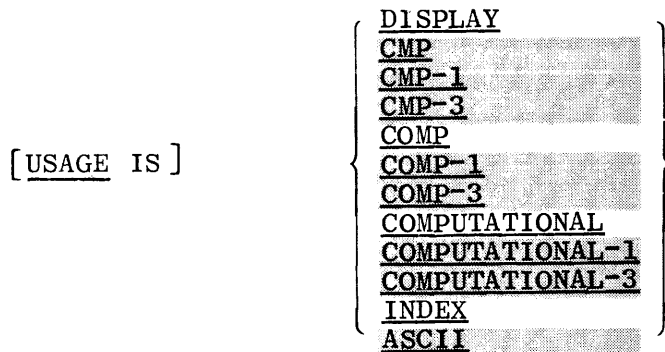
Figure 6-8. Examples of RENAMES

USAGE

USAGE

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

The construct of this clause is:



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

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

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

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

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

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

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

- b. When USAGE IS COMPUTATIONAL, the data item consists of 4-bit coded digits and must be numeric. If a group item is described as computational, the elementary items in the group are computational.
- c. When USAGE IS INDEX, a PICTURE may not be specified. For example, "77 ABC USAGE IS INDEX." An elementary item described with the USAGE IS INDEX clause is called an index data item. An index data item can be referred to directly only in a PERFORM, SEARCH, or SET statement or in a relational condition, and is used for temporary storage of an index-name.

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

For example: PC S9999 CMP-3 VALUE +1234 IN MEMORY = 01234C
 PC S9999 CMP-3 VALUE -1234 IN MEMORY = 01234D
 PC 9999 CMP-3 VALUE +1234 IN MEMORY = 012340
 PC 9999 CMP-3 VALUE -1234 IN MEMORY = 012340

0 indicates that one digit of filler has been added.

- e. When USAGE IS ASCII is used, it specifies that the data item consists of ASCII coded data. ASCII may only be specified for 77 or 01 level entries in the WORKING-STORAGE SECTION. The file records of files described with the RECORDING MODE IS ASCII clause are USAGE IS ASCII by default. A DISPLAY or COMPUTATIONAL item is automatically converted to its ASCII equivalent whenever the receiving item is defined as ASCII. An ASCII item is automatically converted to its numeric or EBCDIC equivalent when the receiving field is COMPUTATIONAL or DISPLAY.

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

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

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

USAGE

whenever the receiving area is declared DISPLAY, except when the sending CMP item is a group item.

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

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

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

VALUE

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

The construct of this clause is:

$$\left\{ \begin{array}{l} \underline{VA} \\ \underline{VALUE} \end{array} \right\} \text{ IS literal-1} \quad \left[\left\{ \begin{array}{l} \underline{THRU} \\ \underline{THROUGH} \end{array} \right\} \text{ literal-2} \right]$$

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

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

Abbreviation VA can be used in lieu of VALUE.

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

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

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

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

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

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

- a. If a category of an item is numeric, all literals in the VALUE clause must be numeric literals; e.g., VA 1, 3 THRU 9, 12, 16 THRU 20, 25 THRU 50, 51, 56.
- b. If the category of the item is alphabetic, all literals in the VALUE clause must be specifically stated as non-numeric literals; e.g., VA IS "A", "B", "C", "F", "M", "N", "O", "P", "Q", "Z".

VALUE

- c. All literals in a VALUE clause of an item must have a value which requires no editing to place that value in the item as indicated by the PICTURE clause.
- d. The function of any editing clause or editing characters in a PICTURE clause is ignored in determining the initial appearance of the item described. However, editing characters are included in determining the length of the item.

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

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

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

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

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

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

WORKING-STORAGE SECTION

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

Organization

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

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

Non-Contiguous WORKING-STORAGE

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

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

WORKING-STORAGE SECTION

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

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

WORKING-STORAGE Records

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

Initial Values

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

Condition-Names

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

Coding the WORKING-STORAGE SECTION

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

PROCEDURE DIVISION**GENERAL**

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

RULES OF PROCEDURE FORMATION

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

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

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

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

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

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

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

EXECUTION OF PROCEDURE DIVISION

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

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

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

STATEMENTS

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

Imperative Statements

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

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

Conditional Statements

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

```
IF conditional {statement-1 } [ ;ELSE {statement-2 } ]
                {NEXT SENTENCE} {NEXT SENTENCE}
```

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

Compiler-Directing Statements

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

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

SENTENCES

SENTENCES

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

Imperative Sentences

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

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

or

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

Conditional Sentences

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

Examples:

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

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

Compiler-Directing Sentences

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

Example:

```
SCAN. COPY SCANNER.
```


SENTENCES

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

Execution of Compiler-Directing Sentences

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

CONTROL RELATIONSHIP BETWEEN PROCEDURES

CONTROL RELATIONSHIP BETWEEN PROCEDURES

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

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

Paragraphs

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

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

Sections

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

CONTROL RELATIONSHIP BETWEEN PROCEDURES

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

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

SEGMENTATION

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

Program Segments

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

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

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

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

Segment Classification

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

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

SEGMENTATION

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

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

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

Priority Numbers

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

```
section-name  SECTION  priority-number.
```

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

SEGMENTATION

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

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

Program A: SEGMENT-LIMIT equals 17.

Non-Gathered

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

Gathered

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

SEGMENTATION

Results of Gathering

<u>Segment</u>	<u>Description</u>	<u>Size in Digits</u>
00-16	Main body of the program	4,000
17	Used frequently	3,500
18	Used infrequently	5,000
19	Used infrequently	5,000
20	Used infrequently	5,000

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

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

DECLARATIVES

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

USE Declarative

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

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

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

COPY Statement as a Declarative

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

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

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

ARITHMETIC EXPRESSIONS

ARITHMETIC EXPRESSIONS

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

- a. An identifier of a numeric elementary item.
- b. A numeric literal.
- c. Such identifiers and literals separated by arithmetic operators.
- d. Two arithmetic expressions separated by an arithmetic operator.
- e. An arithmetic expression enclosed in parentheses.

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

Table 7-1. Combination of Symbols in Arithmetic Expressions

First Symbol	Section Symbol				
	Variable	*/**+-	unary + or -	()
Variable	-	P	-	-	P
*/** + -	P	-	P	P	-
unary + or -	P	-	-	P	-
(P	-	P	P	-
)	-	P	-	-	P

NOTE

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

Arithmetic Operators

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

<u>Character</u>	<u>Meaning</u>
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Formation and Evaluation Rules

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

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

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

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

would be interpreted as:

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

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

ARITHMETIC EXPRESSIONS

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

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

CONDITIONS

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

- a. Relation condition.
- b. Class condition.
- c. Condition-name condition.
- d. Sign condition.
- e. NOT condition.
- f. Condition $\left\{ \begin{array}{c} \underline{\text{AND}} \\ \underline{\text{OR}} \end{array} \right\}$ condition.

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

Logical Operators

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

<u>Logical Operator</u>	<u>Meaning</u>
OR	Logical Inclusive OR
AND	Logical Conjunction
NOT	Logical Negation

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

Relation Condition

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

CONDITIONS

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

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

Table 7-3. Combinations of Conditions and Logical Operators

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

NOTE

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

The general format for a relation condition is as follows:

$$\left\{ \begin{array}{l} \text{data-name-1} \\ \text{literal-1} \\ \text{arithmetic-expression-1} \end{array} \right\} \text{ relational-operator } \left\{ \begin{array}{l} \text{data-name-2} \\ \text{literal-2} \\ \text{arithmetic-expression-2} \end{array} \right\}$$

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

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

Relational Operators

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

IS [NOT] GREATER THAN.

IS [NOT] LESS THAN.

IS [NOT] EQUAL TO.

IS [NOT] >.

IS [NOT] <.

IS [NOT] =.

EQUALS.

Comparison of Operands

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

- a. If the operands are of equal size, characters in corresponding character positions of the two operands are compared starting from the high-order end through the low-order end. If all pairs of characters compare equally through the last pair, the operands are considered equal when the low-order end is reached. The first pair of unequal characters to be encountered is compared to determine their respective relationship. The operand that contains the character that is positioned higher in the internal collating sequence is considered to be the greater operand.
- b. If the operands are of unequal size, the comparison of characters proceeds from high-order to low-order positions until a pair of unequal characters is encountered, or until one of the operands has no more characters to compare. If the end of the shorter operand is reached and the remaining characters in the longer operand are spaces, the two operands are considered to be equal.

Numeric. For operands that are numeric, a comparison results in the determination that one of them is less than, equal to, or greater than the other with respect to the algebraic value of the operands. The length of the oper-

CONDITIONS

ands, in terms of number of digits, is not significant. Zero is considered a unique value regardless of the sign. Comparison of these operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparisons.

The signs of signed numeric operands will be compared as to their algebraic value of being plus (highest) or minus (lowest).

Sign Condition

The sign condition determines whether or not the algebraic value of a numeric operand is less than, greater than, or equal to 0. The general construct for a sign condition is as follows:

arithmetic-expression IS [NOT] { POSITIVE
NEGATIVE
ZERO }

An operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero.

Class Condition

The class condition determines whether the operand is numeric; that is, consists entirely of the characters 0, 1, 2, 3, ..., 9, with or without an operational sign, or alphabetic, that is, consists entirely of the characters A, B, C, ..., Z, and space. The general construct for the class condition is as follows:

identifier IS [NOT] { NUMERIC
ALPHABETIC }

The usage of the operand being tested must be described, implicitly or explicitly, as DISPLAY.

The NUMERIC test cannot be used with an item whose record description describes the item as alphabetic. If the record description of the item being tested does not contain an operational sign, the item being tested is determined to be numeric only if the contents are numeric and an operational sign is not present.

The ALPHABETIC test cannot be used with an item whose record description describes the item as numeric. The item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters A thru Z and the space.

Condition-Name Condition

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name. The general construct for the condition-name condition is as follows:

[NOT] condition-name

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

The result of the test is TRUE if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

Evaluation Rules

The evaluation rules for conditions are analogous to those given for arithmetic expressions, except that the following hierarchy applies:

- a. Arithmetic expressions (formulas).
- b. All relational operators.
- c. NOT.
- d. AND.
- e. OR.

CONDITIONS

Simple Conditions

Simple conditions, as distinguished from compound conditions, are subdivided into four general families of conditional tests: Relation Tests, Relative Value Tests, Class Tests, and the Conditional Variable Tests. A detailed explanation of each of these can be found at the beginning of this subsection (title: Conditions).

Compound Conditions

The most common construct of a compound condition is:

$$\text{simple-condition-1} \quad \left\{ \begin{array}{c} \text{AND} \\ \text{OR} \end{array} \right\} \quad \text{simple-condition-2}$$

$$\left[\left\{ \begin{array}{c} \text{AND} \\ \text{OR} \end{array} \right\} \quad \dots \quad \left\{ \begin{array}{c} \text{AND} \\ \text{OR} \end{array} \right\} \quad \text{simple-condition-n} \right]$$

Simple conditions can be combined with logical operators, according to specified rules, to form compound conditions. The logical operators AND, OR, and NOT are shown in table 7-2, where A and B represent simple conditions. Thus, if A is TRUE and B is FALSE, then the expression A AND B is FALSE, while the expression A OR B is TRUE.

The following are illustrations of compound conditions:

- a. AGE IS LESS THAN MAX-AGE AND AGE IS GREATER THAN 20.
- b. AGE IS GREATER THAN 24 OR MARRIED.
- c. STOCK-ON-HAND IS LESS THAN DEMAND OR STK-SUPPLY IS GREATER THAN DEMAND + INVENTORY.
- d. A IS EQUAL TO B, AND C IS NOT EQUAL TO D, OR E IS NOT EQUAL TO F, AND G IS POSITIVE, OR H IS LESS THAN I * J.
- e. STK-ACCT IS GREATER THAN 72 AND (STK-NUMBER IS LESS THAN 100 OR STK-NUMBER EQUAL TO 76920).

Note that it is not necessary to use the same logical connective throughout. The rules for determining the logical (i.e., truth) value of a compound condition are as follows:

- a. If AND is the only logical connective used, then the compound condition is TRUE if, and only if, each of the simple conditions is TRUE.
- b. If OR is the only logical connective used, then the compound condition is TRUE if, and only if, one or more of the simple conditions is TRUE.

- c. If both logical connectives are used, then the conditions are grouped first according to AND, proceeding from left to right, and then by OR, proceeding from left to right.

Parentheses may be used to indicate grouping as specified in the examples below. Parentheses must always be paired the same as in algebra, i.e., the expressions within the parentheses will be evaluated first. In the event that nested parenthetical expressions are employed, the innermost expressions within parentheses are handled first. Examples of using parentheses to indicate grouping are:

- a. To evaluate $C1 \text{ AND } (C2 \text{ OR NOT } (C3 \text{ OR } C4))$, use the first part of rule c above and successively reduce this by substituting as follows:

Let $C5$ equal " $C3 \text{ OR } C4$ ", resulting in
 $C1 \text{ AND } (C2 \text{ OR NOT } C5)$

Let $C6$ equal " $C2 \text{ OR NOT } C5$ ", resulting
in $C1 \text{ AND } C6$

This can be evaluated by referencing table 7-2.

- b. To evaluate $C1 \text{ OR } C2 \text{ AND } C3$, use the second part of rule c and reduce this to $C1 \text{ OR } (C2 \text{ AND } C3)$, which can now be reduced as in example a.
- c. To evaluate $C1 \text{ AND } C2 \text{ OR NOT } C3 \text{ AND } C4$, group first by AND from left to right, resulting in:

$(C1 \text{ AND } C2) \text{ OR } (\text{NOT } C3 \text{ AND } C4)$

which can now be evaluated as in example a.

- d. To evaluate $C1 \text{ AND } C2 \text{ AND } C3 \text{ OR } C4 \text{ OR } C5 \text{ AND } C6 \text{ AND } C7 \text{ OR } C8$, group from the left by AND to produce:

$((C1 \text{ AND } C2) \text{ AND } C3) \text{ OR } C4 \text{ OR } ((C5 \text{ AND } C6) \text{ AND } C7) \text{ OR } C8$

which can now be evaluated as in example a.

- e. The following uses a condition-name as part of the statement.

IF CURRENT-MONTH AND DAY = 15 OR 30... would
be treated as:

IF (CURRENT-MONTH AND DAY = 15) OR 30... the
actual test desired is:

IF CURRENT-MONTH AND (DAY = 15 OR 30)...

CONDITIONS

The required result is that CURRENT-MONTH must be true and DAY must contain either 15 or 30.

Without the parentheses as shown, the conditions are:

1. DAY = 30 or
2. CURRENT-MONTH is true AND DAY = 15.

Abbreviated Compound Conditions

Any relation condition other than the first that appears in a compound conditional statement may be abbreviated as follows:

- a. The subject, or the subject and relational operator, may be omitted. In these cases, the effect of the abbreviated relation condition is the same as if the omitted parts had been taken from the nearest preceding complete relation condition within the same condition; that is, the first relation is a condition and must be complete.

- b. If, in a consecutive sequence of relation conditions (separated by logical operators) the subjects are identical, the relational operators are identical and the logical connectors are identical, the sequence may be abbreviated as follows:

1. Abbreviation 1: when identical subjects are omitted in a consecutive sequence of relation conditions. An example of abbreviation 1 would be:

IF A = B AND = C.

This is equivalent to IF A = B AND A = C.

2. Abbreviation 2: when identical subjects and relational operators are omitted in a consecutive sequence of relation conditions. An example of abbreviation 2 is:

IF A = B AND C.

This is equivalent to IF A = B AND A = C.

- c. As indicated in the previous paragraphs, compound conditions can be abbreviated by having implied subjects, or implied subjects and relational operators, providing the first simple condition is a full relation. The missing term is obtained from the last stated relation

CONDITIONS

in the sentence. The following examples further illustrate the abbreviated compound conditions:

1. IF $A = B$ OR C is equivalent to IF $A = B$ OR $A = C$.
2. IF $A < B$ OR $= C$ OR D is equivalent to IF $A < B$ OR $A = C$ OR $A = D$.

INTERNAL PROGRAM SWITCHES

INTERNAL PROGRAM SWITCHES

Every compiled object program contains eight automatically provided programmatic switches. Switches SW1 through SW8 are composed of one unsigned digit in length and are located in memory locations 0 through 7 of data segment 0.

These switches can be referred to in the PROCEDURE DIVISION by the use of the reserved words SW1, SW2...SW8. During execution, each individual switch setting can be changed by a MOVE, ADD, SUBTRACT, etc.. For example:

```
MOVE 0 TO SW1.  
ADD 1 TO SW2.  
SUBTRACT 1 FROM SW3.
```

Note that SW6 has an effect on the MONITOR DEPENDING....requirement if the statement is present.

The switch memory locations are reserved and operate identically to those of the reserved TALLY locations.

STATEMENT OPTIONS

In the statement descriptions that follow, several options appear frequently: the `ROUNDED` option, the `SIZE ERROR` option, and the `CORRESPONDING` option.

In the discussion below, a resultant-identifier is that identifier associated with a result of an arithmetic operation.

Rounded Option

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 value returned is $(X + 0.5)$, where X is the original argument.

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

Size Error Option

If, after decimal point alignment, the value of a result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. The size error condition applies only to the final results of an arithmetic operation and does not apply to intermediate results, except in the `MULTIPLY` and `DIVIDE` statements, in which case the size error condition applies to the intermediate results as well. Arithmetic faults such as divide by zero will cause program termination if the `SIZE ERROR` clause is not used.

If the `ROUNDED` option 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` option is specified:

- a. If the `SIZE ERROR` option is not specified and a size error condition occurs, the values of the resultant-identifier(s) affected will be unpredictable. 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.
- b. If the `SIZE ERROR` option is specified and 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

STATEMENT OPTIONS

operation. After completion of the execution of this operation, the statement in the SIZE ERROR option is executed.

For ADD and SUBTRACT CORRESPONDING, the SIZE ERROR clause is not executed until all of the individual additions or subtractions are completed.

Corresponding Option

In this discussion, identifier-1 and identifier-2 are identifiers specified in a statement containing the CORRESPONDING phrase.

- a. Rules for valid identifiers are:
 1. All identifiers must refer to group items.
 2. Identifiers may be described with or be subordinate to an item described with a REDEFINES or OCCURS clause.
 3. No identifier may have a USAGE of INDEX.
- b. Data items subordinate to identifier-1 correspond with data items subordinate to identifier-2, if the following rules apply:
 1. Both data items must have the same data-name.
 2. All possible qualifiers for the sending item, up to but not including identifier-1, must be identical to all possible qualifiers for the receiving item up to but not including identifier-2.
 3. In an ADD or SUBTRACT statement, only elementary numeric data items will be considered.
 4. In a MOVE statement, the corresponding sending and/or receiving data items must be elementary. The class may differ.
 5. Any item with a level-number of 66 or 88 or with a Data Description entry containing a REDEFINES, OCCURS, or INDEX clause is not considered. Any item subordinate to an item not eligible for correspondence will also be ignored.
 6. FILLER data items are ignored.

VERBS

The verbs available for use with the COBOL Compiler are categorized below. Although the word IF is not a verb in the English language, it is utilized as such in the COBOL language. Its occurrence is a vital feature in the PROCEDURE DIVISION.

- a. Arithmetic:
 - ADD
 - COMPUTE
 - DIVIDE
 - MULTIPLY
 - SUBTRACT
- b. Compiler Directing:
 - COPY
 - MONITOR
 - NOTE
 - USE
- c. Data Manipulations:
 - EXAMINE
 - FORMAT
 - MICR-EDIT
 - MOVE
- d. Ending:
 - STOP
- e. Input-Output:
 - ACCEPT
 - CLOSE
 - CONTROL
 - DISPLAY
 - OPEN
 - READ
 - SEEK
 - WRITE
 - ZIP
- f. Logical Control:
 - IF
 - WAIT

VERBS

g. Procedure Branching:

ALTER
EXIT
GO
PERFORM

h. Sort:

MERGE
RELEASE
RETURN
SORT

i. Table Manipulation:

SEARCH
SET

j. Debugging:

DUMP
TRACE

Specific Verb Formats

The specific verb formats, together with a detailed discussion of the restrictions and limitations associated with each, appear on the following pages in alphabetic sequence.

ACCEPT

The function of this verb is to permit the entry of low-volume data from the console typewriter.

The construct of this verb is:

ACCEPT identifier [FROM { SPO
mnemonic-name }]

This statement causes the operating object program to halt and wait for appropriate data to be entered on the console printer (SPO). The SPO entry will replace the contents of memory specified by the identifier. The systems operator answers an ACCEPT halt by keying in the following message:

mix-index AXdata-required

If a blank appears between the AX and data-required, the blank character will be included in the data-stream.

The MCP will space fill to the right if the number of characters entered is less, or truncate to right if the number of characters entered is more.

If mnemonic-name is used, it must appear in the SPECIAL-NAMES paragraph and be equated to the hardware-name SPO.

The receiving identifier may be a group level entry and cannot be subscripted.

The maximum number of characters per ACCEPT statement is unlimited.

Because of the inefficiency of entering data through the keyboard, this technique of data transmission should be restricted solely to low-volume input data.

NOTE

The "<" is a backspace character and is not passed by the MCP.

ADD

ADD

The function of this verb is to add two or more numeric data items and adjust the value of the receiving field(s) accordingly.

The construct of this verb has three options.

Option 1:

```
ADD { literal-1 } [ { literal-2 } ... ]  
TO identifier-m [ROUNDED] [ identifier-n [ROUNDED]... ]  
[;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

Option 2:

```
ADD { literal-1 } { literal-2 } [ { literal-3 } ... ]  
GIVING identifier-m [ROUNDED] [ ,identifier-n [ROUNDED] ] ...  
[;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

Option 3:

```
ADD { CORR } identifier-1 TO identifier-2  
[CORRESPONDING]  
[ROUNDED] [;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

With Option 1, the value(s) of the operand(s) preceding the word TO will be added together and the sum will be added to the existing value(s) of operand(s) following the word TO. A resummation does not occur if the value of one of the identifiers changes in the process.

For example, the result of the statement

```
ADD A, B, C TO C, D(C), E
```

is equivalent to

```
ADD A, B, C GIVING TEMP  
ADD TEMP TO C  
ADD TEMP TO D(C)  
ADD TEMP TO E
```

where TEMP is an intermediate result item provided by the compiler.

In Option 2, the sum of the operands preceding the word GIVING will be inserted as a replacement value of identifier(s) following the word GIVING.

In Options 1 and 2, the identifiers must refer to elementary numeric items only, except that identifiers appearing only to the right of the word GIVING may refer to elementary numeric-edited items.

An ADD statement must have at least two operands.

That composite of operands, which is the super-imposition of all operands, excluding the data item that follows the word GIVING, aligned on their decimal points, must not contain more than 125 digits or characters.

The internal format of operands referred to in an ADD statement may differ among each other. Any necessary format transformation and decimal point alignment are automatically supplied throughout the calculation.

Each literal must be a numeric literal.

If, after decimal point alignment with the receiving data item, the calculated result extends to the right of the receiving data item (i.e., an identifier whose value is to be set equal to the sum), truncation will occur. Truncation is always in accordance with the size associated with the resultant identifier. When the ROUNDED option is specified, it causes the resultant identifier to have its absolute value increased by 1 whenever the most-significant digit of the truncated portion is greater than or equal to 5.

Whenever the magnitude of the calculated result exceeds the largest magnitude that can be contained in a resultant data-name, a size error condition arises. In the event of a size error condition, one of two possibilities will occur, depending on whether or not the ON SIZE ERROR option has been specified. The testing for the size error condition occurs only when the ON SIZE ERROR option has been specified.

- a. In the event that ON SIZE ERROR is not specified and size error conditions arise, the value of the resultant identifier is unpredictable.
- b. If the ON SIZE ERROR option has been specified and size error conditions arise, then the value of the resultant identifier will not be altered. After it has been determined that there is a size error condition, the statement-1 associated with the ON SIZE ERROR option will be executed.

If Option 3 is used multiple operations are performed. The operations are executed by the pairing of identical data-names of numeric elementary items subor-

ADD

dinate in hierarchy to identifier-1 and identifier-2. Data-names match if they, and all their possible qualifiers up to, but not including identifier-1 and identifier-2, are the same. All general rules pertaining to the ADD verb apply to each individual ADD operation. For instance, if the size of matched data-names does not correspond, in that the decimal point is out of alignment or the sizes differ, the decimal point alignment or truncation takes place according to the rules previously discussed.

In the process of pairing identical data-names, any data-name with the REDEFINES clause is ignored. Similarly, data-names which are subordinate to the subordinate data-names with the REDEFINES clause are ignored.

NOTE

This restriction does not preclude identifier-1 or identifier-2 from having REDEFINES clauses or from being subordinate to data-names with REDEFINES clauses.

If the CORR or CORRESPONDING option is used, any item in the group referred to which contains an OCCURS clause will be ignored. Any items subordinate to such an item will also be ignored.

In Option 3, if either identifier-1 or identifier-2 is a group item which contains RENAMES entries, the entries are not considered in the matching of names.

In Option 3, identifier-1 and identifier-2 must not have a level number of 66, 77, or 88.

If corresponding data-names are not elementary numeric items, the ADD operation will be ignored.

In Option 3, CORR is an acceptable substitute for CORRESPONDING.

ALTER

The function of this verb is to modify a predetermined sequence of operations by changing the operand of a labeled GO TO paragraph.

The construct of this verb is:

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2  
[procedure-name-3 TO [PROCEED TO] procedure-name-4 ...]
```

Procedure-name-1, procedure-name-3, ... are names of paragraphs, each of which contains a single sentence consisting of only a GO TO statement as defined under Option 1 of the GO TO verb. Procedure-name-2, procedure-name-4, ... are not subject to the same restrictions and they may be either paragraph-names or section-names.

When control passes to procedure-name-1, control is immediately passed to procedure-name-2 rather than to the procedure-name referred to by the GO TO statement in procedure-name-1. Procedure-name-1 is therefore a "gate" which remains set until again referenced by another ALTER statement.

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

All other uses of the ALTER statement are valid and are performed even if the GO TO which the ALTER refers to is in an overlayable section, as long as the section priority number is less than 50.

CLOSE

CLOSE

The function of this verb is to communicate to the MCP that the designated file-name being operated on or created is programmatically completed, and also to fulfill the stated action requirements.

The construct of this verb is:

CLOSE file-name-1 [REEL] [WITH { LOCK
PURGE
RELEASE
NO REWIND
REMOVE } [WITH CRUNCH]]
[file-name-2...]

File-names must not be those defined as being SORT files. A file must have been OPENed previously before a CLOSE statement can be executed for the file. File space in memory will not be allocated until the file has been OPENed. When a file is programmatically CLOSEd and the assigned unit is released, the memory allocated for that file will be returned to the MCP. The MCP I/O assignment table reflects any unit which remains assigned to the program after the file on that unit has been CLOSEd.

The above statement applies to the following categories of input and output files.

- a. Files whose input and output media involve print files, card files, etc.
- b. Files which are contained entirely on one reel of magnetic tape and are the only files on that reel.
- c. Files which may be contained on more than one physical reel of magnetic tape. Furthermore, the number of reels might possibly be higher than the number of physical tape units provided on the system.
- d. Disk files.

To show the effects of the CLOSE options, each type of file will be discussed separately.

- a. Card Input.
 1. CLOSE - does not release the input memory areas or the reader.
 2. CLOSE WITH NO REWIND - same as CLOSE.

3. CLOSE WITH RELEASE - releases the input memory areas and returns the reader to the MCP.

4. CLOSE WITH LOCK - same as CLOSE WITH RELEASE.

5. CLOSE WITH PURGE - same as CLOSE WITH RELEASE.

6. CLOSE WITH REMOVE - same as CLOSE.

7. CLOSE WITH CRUNCH - same as CLOSE.

b. Card Output.

1. CLOSE - punches the trailer label (if any) and does not release the output memory areas or the punch.

2. CLOSE WITH NO REWIND - same as CLOSE.

3. CLOSE WITH RELEASE - releases the output memory areas and returns the punch to the MCP.

4. CLOSE WITH LOCK - same as CLOSE WITH RELEASE.

5. CLOSE WITH PURGE - same as CLOSE WITH RELEASE.

6. CLOSE WITH REMOVE - same as CLOSE.

7. CLOSE WITH CRUNCH - same as CLOSE.

c. Magnetic Tape Input.

1. CLOSE - rewinds the tape and does not release the input memory areas. The unit remains assigned to the program.

2. CLOSE WITH NO REWIND - same as CLOSE except the tape is not rewound.

3. CLOSE WITH LOCK - releases the input memory areas, rewinds the tape, and the MCP marks the unit not ready.

4. CLOSE WITH RELEASE - releases the memory input areas, rewinds the tape, and returns the unit to the MCP.

5. CLOSE WITH PURGE - releases the input memory areas, rewinds the tape, and if a write ring is in the reel, over-writes the label, making the tape a scratch tape which becomes a candidate for use by the MCP. The unit is returned to the MCP.

6. CLOSE WITH REMOVE - same as CLOSE.

7. CLOSE WITH CRUNCH - same as CLOSE.

d. Magnetic Tape Output.

CLOSE

1. CLOSE - does not release the output memory areas, writes the trailer label (if any), and rewinds the tape. The unit remains assigned to the program.
2. CLOSE WITH NO REWIND - does not release the output memory areas, writes the trailer label (if any). The tape remains positioned beyond the trailer label (or tape mark if there is no trailer label). The unit remains assigned to the program.
3. CLOSE WITH LOCK - releases the output memory areas, writes the trailer label (if any), rewinds the tape, and the MCP marks the unit not ready.

4. CLOSE WITH RELEASE - releases the output memory areas, writes the trailer label (if any), rewinds the tape, and returns the unit to the MCP.
5. CLOSE WITH PURGE - releases the output memory areas, writes the trailer label (if any), rewinds the tape, returns the unit to the MCP, and the MCP overwrites the label, making it a scratch tape and a candidate for use by the MCP.

6. CLOSE WITH REMOVE - same as CLOSE.

7. CLOSE WITH CRUNCH - same as CLOSE.

e. Printer Output.

1. CLOSE - prints the trailer label (if any) and does not release the the output memory areas or the printer.

2. CLOSE WITH NO REWIND - same as CLOSE.

3. CLOSE WITH RELEASE - releases the output memory areas and returns the printer to the MCP.

4. CLOSE WITH LOCK - same as CLOSE WITH RELEASE.

5. CLOSE WITH PURGE - same as CLOSE WITH RELEASE.

6. CLOSE WITH REMOVE - same as CLOSE.

7. CLOSE WITH CRUNCH - same as CLOSE.

f. Disk Files. The actions taken on files ASSIGNED to DISK will be discussed in terms of old files and new files. An old file is one that already exists on disk and appears in the MCP Disk Directory. A new file is one created by the program and does not appear in the Directory. A new file may only be referenced by the program which creates it.

1. CLOSE - does not release the input/output memory areas.

(a) For an old file, the file is left in the Directory and is available to other programs.

(b) For a new file, the file is not entered in the Directory; however, it remains on the disk and may be OPENed again by this program.

2. CLOSE WITH NO REWIND - not permitted on disk files.

3. CLOSE WITH RELEASE - releases the input/output memory areas.

(a) For an old file, the file is left in the Directory and is available to other programs.

(b) For a new file, the file is entered in the Directory and is available to other programs.

4. CLOSE WITH LOCK - releases the input/output memory areas.

(a) For an old file, the file remains in the Directory and is made available.

(b) For a new file, the file is entered in the Directory and is available to other programs.

5. CLOSE WITH PURGE - releases the input/output memory areas.

(a) An old file is immediately removed from the disk and deleted from the Directory.

(b) A new file will be immediately removed from the disk.

6. CLOSE WITH REMOVE - releases the input/output memory areas.

This option will cause the MCP to REMOVE a file from the Disk Directory that has the same file-id as the file being closed.

This action will take place prior to entering the closing files file-id in the Disk Directory. Use of this option will eliminate the DUPLICATE FILE condition and reduce operator intervention.

If the REMOVE option is not used, the "RM" SPO input message will accomplish the same results.

CLOSE

7. CLOSE WITH CRUNCH - releases the input/output memory areas.
 - (a) For an old file when used with the RELEASE or LOCK option the file remains in the Directory and is made available.
 - (b) For a new file, when used with the RELEASE or LOCK options, the file is entered in the Directory and is available to other programs. If the MCP has opened only one area for the file, the portion of that area which has not been used is returned to available disk. Also, the disk file header is changed so that no records may be added to this file. If more than one area is in use, no disk space is returned to available disk.

If a file has been specified as being OPTIONAL, the standard end-of-file processing is permitted whenever the file is not present.

If a CLOSE statement without the REEL option has been executed for a file, a READ, WRITE, or SEEK statement for that file must not be executed unless an intervening OPEN statement for that file is executed.

The CLOSE REEL option signifies that the file-name being CLOSED is a multi-reel magnetic tape input or output file. The reel will be CLOSED when the CLOSE REEL statement is encountered and an automatic OPEN of the next sequential reel of the multi-reel file is then performed by the MCP.

COMPUTE

The function of this verb is to assign to a data item the value of a numeric data item, literal, or arithmetic expression.

The construct of this verb is:

$$\text{COMPUTE identifier-1 [ROUNDED] = } \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal} \\ \text{arithmetic expression} \end{array} \right\}$$

`[;ON SIZE ERROR statement-1 [;ELSE statement-2]]`

The literal must be a numeric literal.

Identifier-2 must refer to an elementary numeric item. Identifier-1 may describe an elementary numeric edited item.

The arithmetic expression option permits the use of any meaningful combination of identifiers, numeric literals, arithmetic operators, and parenthesization, as required.

All rules regarding ON SIZE ERROR, ROUNDED options, truncation and editing are the same as for ADD.

If numeric-literal exponents are used, the results are accurate up to 18 digits in length and to as many decimal places.

COPY

COPY

The function of this verb is to allow library routines contained on a source language library file to be incorporated into the program.

The construct of this verb contains the following two options:

Option 1:

COPY library-name .

Option 2:

COPY library-name

[<u>REPLACING</u>	{ word-1 data-name-1 }	<u>BY</u>	{ word-2 identifier-2 literal-1 }	...]	.
		{ word-3 data-name-3 }	<u>BY</u>	{ word-4 identifier-4 literal-2 }			

The COPY statement may refer only to one library entry in the library. Library-name is the value placed in a library entry bounded by quotes or a procedure-name type word. The library entry can contain up to three 10-character non-numeric literals each separated by a slash (/), following normal naming conventions for disk files.

The library file is inserted in the source program immediately after the COPY statement at compilation time. The result is the same as if the library data were actually a part of the source program.

Library data can encompass an entire procedure, which may be any number of statements, paragraphs, or entire source program divisions (or parts thereof).

Library files may not contain COPY statements.

No statement may appear to the right of the COPY statement on the same source card.

COPY during the PROCEDURE or ENVIRONMENT divisions must follow a SECTION or paragraph-name, and all information contained in the library file is included and can be fully referenced.

On a COPY during the DATA DIVISION, the FD file-name, or the 01 level data-name preceding the COPY is saved and the relative constructs from the library file are discarded. For example, the statement

```
FD MASTER-INPUT COPY "MASTER".
```

will cause the library file titled MASTER to be inserted into the source program immediately following the COPY statement. The source program must refer to the FD file-name as MASTER-INPUT, not as MASTER. The library FD file-name will appear on the output listing, but cannot be referenced in the source program.

Library texts copied from the library are flagged on the output listing by an "L" preceding the sequence number.

In Option 2, a word is defined as being any COBOL word that is not a COBOL reserved word. For example, the following statement reflects non-reserved COBOL words AAA,BBB and 1234, where AAA and BBB are data-names and 1234 is a COBOL word:

```
MULTIPLY AAA BY BBB, THEN GO TO 1234.
```

If the COPY REPLACING option is specified, each word-1 or data-name-1 stipulated will be replaced by the word-2 or identifier-2 entries specified in the option. Data-names may not be subscripted, indexed, or qualified.

Use of the COPY REPLACING option requires that the "library-name" COBOL source image file be present on disk, prior to compilation of the source program containing the COPY REPLACING option. The use of this option will not cause alteration of the library file residing on disk.

In Option 2, literals contained in a library file cannot be replaced by literals, words, or identifiers. If a numeric literal is the last entry in a replacing list, it must be followed by a blank and then a period. For example:

```
COPY BERMAN REPLACING AAA BY HOURS,  
BBB BY PAY-SCALE, 1234 BY 58b.
```

The COPY REPLACING option is exceptionally useful for conversion of generalized COBOL source-language library routines into specific and well-named routines within a given program. For example, a generalized COBOL source-language library routine may use the following data-names for the purposes shown.

COPY

<u>Data-name</u>	<u>Purpose</u>
AAA	Monthly hours worked per employee.
BBB	Employee pay-rate.
CCC	Employee social security number.
DDD	Employee income tax rate.
EEE	Employee year to date gross income.
FFF	Employee year to date net income.
GGG	Employee gross pay for month. Employee net pay for the month.
.	.
.	.
.	.
1234	Specifies a GO TO exit from the routine.

A program calling upon the above generalized routine can replace the non-descriptive data-names with descriptive names as defined in the program's record description or WORKING-STORAGE area. For example:

```
COPY...REPLACING AAA BY HOURS-WORKED
COPY...REPLACING BBB BY RATE-OF-PAY
COPY...REPLACING CCC BY SOC-SEC-NR
COPY...REPLACING DDD BY INC-TAX-RATE
COPY...REPLACING EEE BY YR-TO-DATE-GROSS
COPY...REPLACING FFF BY YR-TO-DATE-NET
COPY...REPLACING GGG BY THIS-MONTHS-GROSS
COPY...REPLACING HHH BY THIS-MONTHS-NET
.
.
.
COPY...REPLACING 1234 BY WRITE-EMPLOYEE-DRAFT.
```

The specified source program data-names and exit points will be inserted into the library file routine at every occurrence of the assigned generalized names within the routine.

Library Creation. A library file will be created during a COBOL compilation each time a source statement is encountered that contains an "L" in column 7 followed by a library-name on that same image. A library file may contain up to a maximum of 20,000 card images.

Each library file in the source program will be terminated when a source image is encountered which contains an "L" in column 7 followed by all blanks or another library-name.

Once a file has been created, it may be COPYed by other programs, or by the creating program in succeeding FD, 01, or procedure COPY statements.

The source data used to create an original library file will also be compiled into the object program at the point of appearance.

All assigned library-names must be unique to other library-names contained in the library, to preserve the integrity of the COBOL library system.

Library files to be used with the COPY verb can be created by a user program which creates a card image file on disk. The compiler will automatically accept the blocking used when the file was created.

Library Maintenance. The COBOL compiler allows several methods for updating existing library files. Due to the fact that new source and/or library files are created during the initial pass of the compilation, source programs which create library files need not be complete COBOL programs. In this manner, source programs which create library files may be compiled with patch cards to be applied to the library files (as described in Section 11) by use of the dollar option MERGE.

Another method of updating library files is to use the library file as a source program rather than a library file. Using the dollar options MERGE and NEW to make patches and create a new source file, changes and updates may be made.

Another possible solution to the library file maintenance problem, is to create all library files in one source program. This would allow for centralized control of library files and also aid in documentation and training of new personnel.

DISPLAY

DISPLAY

The function of this verb is to provide for the printing of low-volume data, error messages, and operator instructions on the console typewriter.

The construct of this verb is:

$$\begin{array}{l} \underline{\text{DISPLAY}} \quad \left\{ \begin{array}{l} \text{literal-1} \\ \text{identifier-1} \end{array} \right\} \quad \left[\left\{ \begin{array}{l} \text{literal-2} \\ \text{identifier-2} \end{array} \right\} \quad \dots \right] \\ \left[\underline{\text{UPON}} \quad \left\{ \begin{array}{l} \underline{\text{SPO}} \\ \text{mnemonic-name} \end{array} \right\} \right] \end{array}$$

Each literal may be any figurative constant except ALL.

All special registers (DATE, TIME, TALLY, SW₁ ... SW_n, etc.) may be DISPLAYed.

The DISPLAY statement causes the contents of each operand to be written on the supervisory printer (SPO), from the MCP SPO queue, to ensure that a program is not operationally deterred while a message is printing.

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

The identifiers may be subscripted or indexed and can be COMPUTATIONAL or DISPLAY items.

An infinite number of characters may be displayed with one DISPLAY statement.

The DISPLAY series option will cause the literals or identifiers to be concatenated, printed on one line and, if required, the MCP will cause automatic carriage returns and line feeds for information extending to other lines of print.

When mnemonic-name is used, it must appear in the SPECIAL-NAMES paragraph equated to the hardware-name SPO.

DIVIDE

The function of this verb is to divide one numerical data-item into another and set the value of an item equal to the result.

The construct of this verb contains the following two options:

Option 1:

```
DIVIDE  [MOD]  {literal-1 } INTO  identifier-2  [ROUNDED ]
          {identifier-1}
[;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

Option 2:

```
DIVIDE  [MOD]  {literal-1 }    {BY }    {literal-2 }
          {identifier-1}    {INTO}    {identifier-2}
          GIVING identifier-3 [ROUNDED ]
          [REMAINDER identifier-4 [ROUNDED]]
[;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

Identifier-3 and identifier-4 of Option 2 may refer to elementary numeric-edited items.

Each literal must be a numeric literal.

Division by zero is not permissible and, if executed, will result in a size error indication. This can be handled programmatically, either by doing a zero test prior to the division or by the use of the SIZE ERROR clause. If SIZE ERROR is not written, an attempt to divide by zero will result in program termination.

All identifiers must refer to elementary numeric items.

In Option 1, the value of the operand preceding the word INTO will be divided into the operand following INTO and the resulting quotient stored as the new value of the latter.

The use of the BY option will cause literal-1/identifier-1 to be divided by literal-2/identifier-2, whereas the INTO option will cause literal-1/identifier-1 to be divided into literal-2/identifier-2.

In Option 2, the resulting quotient will be stored as the new value of identifier-3. The value of the operands immediately to the left of the word GIVING will remain unchanged.

DIVIDE

The ROUNDED option and ON SIZE ERROR clause and truncation are the same as those discussed for the ADD statement.

The size of the operands is determined by the sum of the divisor and the quotient. The sum of the two cannot exceed 99 digits.

The use of the MOD option will cause the remainder to be placed in identifier-2 of Option 1 and identifier-3 of Option 2. The remainder will be carried to the same degree of accuracy as defined in the PICTURE of the quotient, and all extra positions will be filled with zeros.

Literals cannot be used as dividends.

The use of the REMAINDER option will cause the remainder to be placed in identifier-4, and identifier-3 will contain the quotient.

Use of both the MOD and REMAINDER is prohibited.

DUMP

The DUMP statement causes messages to be displayed on the line printer instead of the console printer. The syntax is as follows:

```
DUMP [list]
```

where [list] is a list of data-names, literals, and blanks (for spacing). The DUMP statement must be used in conjunction with the MONITOR declaration because it uses the same WRITE routine and the printer file.

EXAMINE

EXAMINE

The function of this verb is to replace a specified character, and/or to count the number of occurrences of a particular character in a data item.

The construct of the verb contains the following two options:

Option 1:

EXAMINE identifier-1
TALLYING { ALL
LEADING
UNTIL FIRST } {literal-1
identifier-2} [REPLACING BY {literal-2
identifier-3}]

Option 2:

EXAMINE identifier-1 REPLACING { ALL
LEADING
[UNTIL] FIRST }
{literal-3
identifier-4} BY {literal-4
identifier-5}

The description of identifier-1 must be such that USAGE is DISPLAY explicitly or implicitly.

Each literal used in an EXAMINE statement must consist of a single DISPLAY character. Figurative constants will automatically represent a single DISPLAY character.

Examination proceeds as follows:

- a. For items that are not numeric, examination starts at the leftmost character and proceeds to the right. Each 8-bit character in the item specified by the data-name is examined in turn. Any reference to the first character means the left-most character.
- b. If an item referenced by the EXAMINE verb is numeric, it must consist of numeric characters and may possess an operational sign. Examination starts at the leftmost character (excluding the sign) and proceeds to the right. The low order digit of each character except the sign is examined in turn. Regardless of where the sign is physically located, it is completely ignored by the EXAMINE verb. Any reference to the first character means the leftmost numeric character.

The TALLYING option creates an integral count (i.e., a tally) which replaces the value of a special register called TALLY. The count represents the number of:

- a. Occurrences of literal-1 or identifier-2 when the ALL option is used.
- b. Occurrences of literal-1 or identifier-2 prior to encountering a character other than literal-1 or identifier-2 when the LEADING option is used.
- c. Characters not equal to literal-1 or identifier-2 encountered before the first occurrence of literal-1 or identifier-2 when the UNTIL FIRST option is used.

When either of the REPLACING options is used (i.e., with or without TALLYING), the replacement rules are as follows:

- a. When the ALL option is used, then literal-2 or identifier-3 or literal-4 or identifier-5 is substituted for each occurrence of literal-1 or identifier-2 or literal-3 or identifier-4.
- b. When the LEADING option is used, the substitution of literal-2 or identifier-3 or literal-4 or identifier-5 terminates as soon as a character other than literal-1 or identifier-2 or literal-3 or identifier-4 or the right-hand boundary of the data item is encountered.
- c. When the UNTIL FIRST option is used, the substitution of literal-2 or identifier-3 or literal-4 or identifier-5 terminates as soon as literal-1 or identifier-2 or literal-3 or identifier-4 or the right-hand boundary of the data item is encountered.
- d. When the FIRST option is used, the first occurrence of literal-3 or identifier-4 is replaced by literal-4 or identifier-5.

The field called TALLY is a 5-digit field provided by the compiler. Its usage is COMPUTATIONAL and will be reset to zero automatically when the EXAMINE... TALLY option is encountered.

EXIT

EXIT

The function of this verb is to provide a terminating point for a PERFORM loop, whenever required.

The construct of this verb is:

EXIT.

If the EXIT statement is used, it must be preceded by a paragraph-name and appear as a single one-word paragraph. EXIT is documentary only, but if used, must follow the rules of COBOL.

The EXIT is normally used in conjunction with conditional statements contained in procedures referenced by a PERFORM statement. This allows branch paths within the procedures to rejoin at a common return point.

If control reaches an EXIT paragraph and no associated PERFORM or USE statement is active, control passes through the EXIT point to the first sentence of the next paragraph.

GO TO

The function of this verb is to provide a means of interrupting out of the sequential, sentence by sentence, execution of code, and to permit continuation at some other location indicated by the procedure-name(s).

The construct of this verb has the following two options:

Option 1:

GO TO [procedure-name]

Option 2:

GO TO procedure-name-1 [, procedure-name-2] ... , procedure-name-3,
DEPENDING ON identifier

Each procedure-name is the name of a paragraph or section in the PROCEDURE DIVISION of the program.

Whenever a GO TO statement (represented by Option 1) is executed, control is unconditionally transferred to a procedure-name, or to another procedure-name if the GO TO statement has been changed by an ALTER statement.

A GO TO statement is unrestricted as to where it branches to in a segmented program. It can call upon any segment at either the section level or paragraph levels.

In Option 1, when the GO TO is referred to by an ALTER statement, the following rules apply, regardless of whether or not procedure-name is specified:

- a. The GO TO statement must be the only statement in the paragraph.
- b. If the procedure-name is omitted, and if the GO TO statement is not referenced by an ALTER statement prior to the first execution of the GO TO statement, the MCP will cause the job to be terminated.

If a GO TO statement represented by Option 1 appears in an imperative statement, it must appear as the only or the last statement in a sequence of imperative statements.

GO TO

In Option 2, GO TO... DEPENDING... may specify up to 1023 procedure-names in a single statement. The data-name in the format following the words DEPENDING ON must be a numeric elementary item described without any positions to the right of the assumed decimal point. Furthermore, the value must be positive in order to pass control to the procedure-names specified. Control will be transferred to procedure-name-1 if the value of the identifier is 1, to procedure-name-2 if the value is 2, etc. If the value of the identifier is anything other than a positive integer, or if its value is zero, or its value is higher than the number of procedure-names specified, control will be passed to the next statement in normal sequence. For example:

GO TO MFG, RE-SALE, STOCK, DEPENDING ON S-O.

VALUE OF S-O	GO TO PROCEDURE-NAME
-1	next statement
0	next statement
1	MFG
2	RE-SALE
3	STOCK
4	next statement

IF

The IF statement causes a condition to be evaluated. The subsequent action of the object program depends on whether the value of the condition is true or false.

The construct for the IF statement is as follows:

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

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

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

When an IF statement is executed, the following action is taken:

- a. If the condition is true, the statements immediately following the condition (represented by statement-1) are executed, and control then passes implicitly to the next sentence unless statement-1 causes some other transfer of control.
- b. If the condition is false, either the statements following ELSE are executed or, if the ELSE clause is omitted, the next sentence is executed.

When an IF statement is executed and the NEXT SENTENCE phrase is present, control passes explicitly to the next sentence, depending on the truth value of the condition and the placement of the NEXT SENTENCE phrase in the statement.

IF statements within IF statements may be considered as paired IF and ELSE combinations, proceeding from left to right; thus, any ELSE encountered is considered to apply to the immediately preceding IF that has not already been paired with an ELSE.

When control is transferred to the next sentence, either implicitly or explicitly, control passes to the next sentence as written or to a return mechanism of a PERFORM or a USE statement.

The method of evaluating conditional expressions allows early exit, once the truth value of the expression has been determined.

MERGE

MERGE

The function of this verb is to combine two or more files on a set of specified keys, and during the process, to make records available, in merge order, to an output procedure or to an output file.

The construct of this verb is:

```
MERGE file-name-1 [ { PURGE } ON ERROR ]
                        [ { END } ]
ON { ASCENDING } KEY data-name-1 [ data-name-2 ] ...
   { DESCENDING }
[ ON { ASCENDING } KEY data-name-3 [ data-name-4 ] ... ] ...
   { DESCENDING }
[ WITH file-name-6 ]
USING file-name-2 file-name-3 [ file-name-4 ] ...
{ OUTPUT PROCEDURE IS procedure-name-1 [ { THROUGH } procedure-name-2 ]
  { GIVING file-name-5 [ LOCK
                        [ RELEASE ] ] }
```

Syntax Rules

- a. File-name-1 must be described in a sort-merge file description (SD) entry in the DATA DIVISION.
- b. Procedure-name-1 represents the name of an output procedure.
- c. File-name-2, file-name-3, file-name-4, and file-name-5 must be described in a file description (FD) entry, not in a sort-merge file description (SD) 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.

- d. File-name-6 must be described in a file description (FD) entry in the DATA DIVISION.
- e. The words THRU and THROUGH are equivalent.
- f. Date-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names and are subject to the following rules:
 - 1. The data items identified by KEY data-names must be described in records associated with file-name-1.
 - 2. KEY data-names may be qualified.
 - 3. The data items identified by KEY data-name must not be variable length items.
 - 4. If file-name-1 has more than one record description, then the data items identified by KEY data-names may all be described within one of the record descriptions or in any combination of record descriptions. It is not necessary to redescribe the KEY data-names in each record description.
 - 5. 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.
- g. No more than one file-name from a multiple file reel can appear in the MERGE statement.
- h. File-names must not be repeated within the MERGE statement.
- i. MERGE statements may appear anywhere in the PROCEDURE DIVISION except in the DECLARATIVES portion or in an input or output procedure associated with a SORT or MERGE statement.
- j. As many as eight (8) file-names may be specified in the USING clause.

General Rules

- a. The MERGE statement will merge all records contained in file-name-2, file-name-3, and file-name-4. The files referenced in the MERGE statement must have been CLOSED with LOCK or RELEASE prior to 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.
- b. The data-names following the word KEY are listed from left to right in the MERGE statement in order of decreasing significance without

MERGE

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.

1. 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.
 2. 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 rules for comparison of operands in a relation condition.
- c. The collating sequence that applies to the comparison of the non-numeric key data items specified is determined in the following order of precedence:
1. When file-name-6 is specified it invokes the virtual collating sequence capability of the MERGE. The file description (FD) entry which describes file-name-6 must describe a file which exists on disk, and was created by the utility program SORT/COLLATE.
 2. When file-name-6 is not specified, the collating sequence used is the binary EBCDIC collating sequence.
- d. The output procedure must consist of one or more paragraphs or sections that appear contiguously in a source program and do not form a 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 merged order, from file-name-1.

The restrictions on procedural statements within the output procedure are as follows:

1. 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.

2. The output procedures must not contain any SORT or MERGE statements.
 3. 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.
- e. 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 paragraph or section in the output procedure. When control passes the last statement in the output procedure, the return mechanism provides for termination of the merge, and then passes control to the next executable statement after the MERGE statement. Before entering the output 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.
- f. If the GIVING phrase is specified, all of the merged records in file-name-1 are automatically written to file-name-5 as the implied output procedure for this MERGE statement.
- g. When, according to the rules for the comparison of operands in a relation condition, the contents of all the key data items of one data record are equal to the contents of the corresponding key data items of one or more other data records, the order of return of these records:
1. Follows the order of the associated input files as specified in the MERGE statement.
 2. Is such that all records associated with one input file are returned prior to the return of records from another input file.
- h. When the records in the files referenced by file-name-2, file-name-3...are not ordered as described in the ASCENDING or DESCENDING KEY clauses, the MERGE will take place as previously described but with all improperly ordered data records being placed on the output file or RETURNED to the output procedure immediately after they are read from their respective input files. As a result, when such a condition exists, the output from the MERGE statement will not be in a strict ASCENDING or DESCENDING KEY order.

MERGE

- i. The ON ERROR option is provided to allow programmers some control over irrecoverable parity errors when an OUTPUT PROCEDURE is not present in a program. PURGE will cause all records in a block containing an irrecoverable parity error to be dropped, and processing will be continued after a SPO message giving the relative position in the file of the bad block has been printed. END will cause a program termination and is assumed if no option is specified.
- j. The LOCK, and RELEASE options may be used to specify the type of file close for the GIVING file, file-name-5. (See the CLOSE statement). Default is LOCK.

MOVE

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

The construct for the MOVE statement consists of the following two options:

Option 1:

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

Option 2:

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

Identifier-1 and literal represent the sending field; identifier-2, identifier-3 represent the receiving fields. Literal may be any literal or figurative constant consistent with the class of the receiving field.

Option 1 provides for multiple receiving fields. The data designated by the literal or identifier-1 will be moved first to identifier-2, then to identifier-3, etc. Subscripting or indexing associated with identifier-1 is evaluated only once, immediately before data is moved to the first receiving field. The notes referencing identifier-2 also apply to the other areas.

The result of the statement:

MOVE A(SUB) TO SUB, B(SUB)

would produce the same result as:

MOVE A(SUB) TO TEMP.
MOVE TEMP TO SUB.
MOVE TEMP TO B(SUB).

When Option 2 is used, selected items within identifier-1 are moved, with any required editing, to selected areas within identifier-2. Identifier-1 and identifier-2 must be group items. Items are selected by matching the data-names of items defined within identifier-1 with like data-names of areas defined within identifier-2, according to the rules specified in the discussion of the corresponding option. The resulting operation on each of the sets of matched data items proceeds as if an Option 1 MOVE had been specified.

MOVE

Elementary Moves

Any move in which the sending and receiving items are both elementary items is an elementary move. All other moves are defined as group moves. Every elementary item belongs to one of these five categories:

- a. Numeric.
- b. Numeric Edited.
- c. Alphabetic.
- d. Alphanumeric.
- e. Alphanumeric Edited.

See the PICTURE clause description in section 6 for a detailed discussion of these categories. Group items, non-numeric literals, and all figurative constants, except ZEROS and SPACES, are classed as alphanumeric. Numeric literals and the figurative constant ZEROS are classed as numeric. The figurative constant SPACES is classed as alphabetic.

Illegal Elementary Moves. The rules governing illegal elementary moves are as follows:

1. A numeric-edited item, alphanumeric edited item, SPACES, or an alphabetic item cannot be moved to a numeric or numeric edited item.
2. A numeric literal, ZEROS, a numeric data item, or a numeric edited item cannot be moved to an alphabetic data item.
3. A non-integer numeric literal or a non-integer numeric data item cannot be moved to an alphanumeric or alphanumeric edited data item.

Legal Elementary Moves. The rules governing legal elementary moves are as follows:

4. When an alphanumeric or alphanumeric edited item is a receiving field, justification and any necessary space filling takes place as defined under the JUSTIFIED clause. If the size of the sending field is greater than the size of the receiving field, the excess characters are truncated on the right after the receiving item is filled.

If the sending field is described as being signed numeric, the operational sign will not be moved. If the sign occupies a separate character position (KSIGN), that character will not be moved and the size of the sending field will be considered to be one less than its actual size.

For example:

Given these data descriptions:

77 S PIC K9999.

77 R PIC X(6).

Then the statements:

MOVE -124 TOS.

MOVE S TO R.

will result in R being equal to "0124 "

5. When a numeric or numeric edited item is the receiving field in an elementary move and the data in the sending field is not numeric, zone bits will be stripped. Alignment by decimal point and any necessary zero-filling takes place as defined under the JUSTIFIED clause, except where zeros are replaced because of editing requirements.

When a signed numeric item is the receiving field, the sign of the sending field is placed in the receiving field. Conversion of the sign representation takes place as necessary. If the sending field is unsigned, a positive sign is generated for the receiving field.

When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.

When an alphanumeric item is the sending field, data is moved as if the sending item was described as an unsigned numeric integer.

6. When the receiving field is alphabetic, justification and any necessary space filling takes place as defined under the JUSTIFIED clause. If the size of the sending field is greater than the size of the receiving field, the excess characters are truncated on the right, after the receiving field is filled.

Group Moves

A group move is any move in which either the sending field or the receiving field is a group item. Group moves are handled as alphanumeric to alphanumeric moves, regardless of the class of the receiving field and without consideration for the individual elementary or group items contained within either the sending or receiving area.

MOVE

Translation

Any necessary translation of data from one form of internal representation to another, i.e., ASCII to EBCDIC, will be done for elementary or group moves depending on the usages of the sending and receiving data items.

Index Data Items

An index data item cannot be used as an operand in a MOVE statement. The SET statement must be used to move index data items.

Valid Move Combinations

Figure 7-1 shows the valid combinations of sending and receiving fields permitted in COBOL.

RECEIVING SENDING		ALPHABETIC	AN		AE	DISPLAY NUMERIC		CMP NUMERIC		NE
			GROUP	ELEM		INTEGER	REAL	INTEGER	REAL	
ALPHABETIC		①	①	①	⑥	*	*	*	*	*
AN	GROUP	①	①	①	①	①	①	③	③	①
	ELEM (a)	①	①	①	⑥	②	②	②	②	⑦
AE		①	①	①	⑥	*	*	*	*	*
DISPLAY NUMERIC (DN OR LIT)	INTEGER	*	①	①	⑥	②	②	②	②	⑦
	REAL	*	①	*	*	②	②	②	②	⑦
CMP NUMERIC	INTEGER (b)	*	⑤	⑤	④	②	②	②	②	⑦
	REAL	*	⑤	*	*	②	②	②	②	⑦
NE		*	①	①	⑥	*	*	*	*	*

a - also non-numeric literal

b - also undigit literal

	NON-NUMERIC MOVE	NUMERIC MOVE	LEFT JUST.	JUST. BY DECIMAL	SPACE FILL	ZERO FILL ON RIGHT	ZERO FILL ON LEFT	ANY NECESSARY TRANSLATION	SENDING ZONES STRIPPED	PROPER ZONES STRIPPED OR SUPPLIED BY INTERPRETER	EDITING PERFORMED
①	✓		✓		✓			✓			
②		✓		✓			✓			✓	
③	✓		✓			✓			✓		
④	✓		✓		✓			✓		✓	✓
⑤	✓		✓		✓			✓		✓	
⑥	✓		✓		✓			✓			✓
⑦		✓		✓			✓				✓
*ILLEGAL											

Figure 7-1. Valid MOVE Statement Combinations

MOVE

MULTIPLY

MULTIPLY

The function of this verb is to multiply two operands and store the results in the last-named field (which must be a numeric item).

The construct of this verb is:

```
MULTIPLY {literal-1  
            {identifier-1} BY {literal-2  
                               {identifier-2}  
  
            [GIVING identifier-3] [ROUNDED]  
  
            [;ON SIZE ERROR statement-1 [;ELSE statement-2]]
```

All rules specified under the ADD statement regarding the presence of editing symbols in operands, the ON SIZE ERROR option, the ROUNDED option, the GIVING option, truncation, and the editing results apply to the MULTIPLY statement, except the maximum operand size is 125 digits for the sum of two operands.

The identifiers must be elementary item references. If GIVING is used, identifier-3 may be an elementary edited numeric item. In all other cases, the identifiers used must refer to elementary numeric items only.

If the GIVING option is used, the result of the multiplication replaces the contents of identifier-3; otherwise, it replaces the contents of identifier-2. If GIVING is not used, literal-2 is not permitted, i.e., identifier-2 must appear.

NOTE

The function of this statement is to allow the programmer to write explanatory statements in his program which are to be produced on the source program listing for documentation purposes.

The constructs of this statement are:

Option 1: Paragraph NOTE:

paragraph-name. NOTE any comment.

Option 2: Paragraph NOTE:

NOTE any comment.

Option 3: Sentence NOTE:

NOTE any comment.

Any combination of the characters from the allowable character set may be included in the character string of a NOTE statement.

If a NOTE sentence is the first sentence of a paragraph, the entire paragraph is considered to be commentary. Either Option 1 or Option 2 may be used as NOTE statements on a paragraph level.

If a NOTE statement appears as other than the first sentence of a paragraph, only the sentence constitutes a commentary. After the word NOTE is encountered, the first period followed by a space will cause the compiler to resume compilation unless the new sentence commences with the word NOTE.

Refer to the paragraph entitled CONTINUATION INDICATOR (section 3) for an explanation of comments (* or / in column 7) appearing anywhere within the source program.

OPEN

OPEN

The function of this verb is to initiate the processing of both input and output files. The MCP performs checking or writing, or both, of labels and other input-output operations.

The construct of this verb is:

Option 1:

<u>OPEN</u>	
{	[<u>INPUT</u> file-name-1 [<u>WITH LOCK [ACCESS]</u>] [<u>REVERSED</u>] [<u>WITH NO REWIND</u>] [file-name-2...]] ...
	[<u>OUTPUT</u> file-name-3 [<u>WITH NO REWIND</u>] [file-name-4 ...]] ...
	[{ <u>INPUT-OUTPUT</u> } [<u>I-O</u>] file-name-5 [<u>LOCK [ACCESS]</u>] [file-name-6...]] ...
	[<u>O-I</u> file-name-7 [file-name-8...]] ...
	[<u>EXTEND</u> file-name-9 [<u>WITH LOCK [ACCESS]</u>] [file-name-10...]] ...

Option 2:

<u>OPEN</u>	{ <u>OUTPUT</u> } { <u>I-O</u> } { <u>INPUT-OUTPUT</u> }	file-name
	[<u>WITH PUNCH</u>]	[<u>WITH</u> { <u>PRINT128</u> }] [<u>WITH STACKERS</u>]

Option 2 of the OPEN verb is valid only for 80/96 column data recorders. More than one of the options in Option 2 may be specified in the OPEN statement; for example, OPEN OUTPUT file-name WITH PUNCH INTERPRET STACKERS.

When PRINT128 is specified for the 96 column data recorder, it will require an output area of 96 + 128, or 224 characters. The first 96 characters will be for punch output, and the remaining 128 will be for printing on the card. When PRINT128 is used for the 80 column data recorder, it will require an output area of 80 + 80, or 160 characters. The first 80 characters will be for punch output, and the remaining 80 will be for printing on the card. The maximum number of characters which will be printed is device dependent.

When INTERPRET is specified, it causes the punch data to be printed, therefore no additional output area for print data is required.

When the STACKERS option is used alone, it will cause a write with no data transfer to the stacker selected. If used in conjunction with the other options, data will be transferred also. When the STACKERS option is not used, the cards will be selected to the default stacker on a READ or WRITE.

File-names must not be those defined as being SORT files.

At least one of the options must be specified before a file can be read.

The I-O, INPUT-OUTPUT, and O-I options pertain to disk storage files.

The OPEN statement must be executed prior to the first SEEK, READ, or WRITE statement for that file.

A second OPEN statement for a file cannot be executed prior to the execution of a CLOSE statement for that file.

A file area will not exist in memory until an OPEN statement is executed, which in turn, causes the MCP to allocate memory for the file work area, and any alternate areas or buffers. The MCP will obtain the needed information from the File Parameter Block to determine the file's characteristics. Once the file has been OPENed, memory will remain allocated until the file is programmatically CLOSEd.

The OPEN statement does not obtain or release the first data record. A READ or WRITE statement must be executed to obtain or release, respectively, the first data record.

When the first label is to be checked or written, the user's beginning label subroutine is executed if it is specified by a USE statement.

The REVERSED and the NO REWIND options can only be used with sequential, single-reel tape files.

If the peripheral ASSIGNED to the file permits rewind action, the following rules apply:

- a. When neither the REVERSED nor the NO REWIND option is specified, execution of the OPEN statement for the file will cause the file to be positioned ready to read the first data-record.
- b. When either the REVERSED or the NO REWIND option is specified, execution of the OPEN statement does not cause the file to be positioned. When the REVERSED option is specified, the file must be positioned at its physical end. When the NO REWIND option is specified, the file must be positioned at its physical beginning.
- c. When the NO REWIND option is specified, it applies only to sequential, single-reel files stored on magnetic tape units.

OPEN

When the REVERSED option is specified, the subsequent READ statements for the file makes the data-records available in reverse record order starting with the last record. Each record will be read into its record-area, and will appear as if it has been read from a forward-moving file.

If an input file is designated with the OPTIONAL clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION, the object program causes an interrogation to the MCP, for the presence or absence of a pertinent file. If this file is not present, the first READ statement for this file causes the imperative statement in the AT END clause to be executed only when the operator has responded with an optional file "mix index OF" message.

The I-O or INPUT-OUTPUT option permits the OPENing of a disk file for input and/or output operations. This option demands the existence of the file to be on the disk and cannot be used if the file is being initially created; that is, the file to be OPENed must be present in the MCP disk directory, or has been previously created and CLOSED in the same run of the program.

When any input file option is used, the MCP immediately checks the MCP disk directory to see if the file is present, or if it has been created and CLOSED in the same program run. The system operator will be notified in its absence, and the file can then be loaded if it is available or the program can be DSed (discontinued). If the decision is to load the file, the operator does so and then notifies the MCP to proceed with the program, by means of a "mix-index OK" message.

The O-I option is identical to OPEN I-O, with the exception that with the O-I option the file is assumed to be a new file to the disk directory. The OPEN O-I option will short cut the usual method of initially creating I-O work files within a program, e.g., OPEN OUTPUT, WRITE record(s), CLOSE WITH RELEASE, OPEN I-O, etc. The O-I option does not, nor was it intended to, replace the OPEN I-O option, since the use of OPEN O-I assumes that a new file is to be created each time.

During processing of mass storage files for which the ACCESS MODE is SEQUENTIAL, the OPEN statement supplies the initial address of the first record to be accessed.

The contents of the data-names specified in the FILE-LIMIT clause of the FILE-CONTROL paragraph (at the time the file is OPENed) are used for all checking operations while that file is OPEN. The FILE-LIMIT clause is dynamic only to this extent.

When an OPEN OUTPUT statement is executed for a magnetic tape file, the MCP searches the assignment table for an available scratch tape, writes the label if specified by the program, and executes any USE declaratives for the file. If no scratch tape is available, a message to the operator is typed and the program is suspended until the operator mounts such a tape or one becomes available due to the termination of a multiprogramming program.

OPENing of subsequent reels of multi-reel tape files is handled automatically by the MCP and requires no special consideration by the programmer.

OPEN WITH LOCK on a permanent disk file contained in the Disk Directory denies the opening of that file to all other operating programs in the program mix. If another program attempts to OPEN INPUT on a file already opened with LOCK, the following MCP control message will appear on the console printer: job-specifier LOCKED DISK FILE file-identifier.

OPEN WITH LOCK ACCESS on a permanent file contained in the Disk Directory allows the subsequent opening of that file by other operating programs in the program mix, but only if they use the OPEN INPUT option.

The EXTEND phrase can only be used for SEQUENTIAL files. When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file add records to the file as though the file had been OPENed with the OUTPUT phrase.

PERFORM

PERFORM

The function of this verb is to depart from the normal sequence of execution in order to execute one or more procedures, either a specified number of times or until a specified condition is satisfied. Following this departure, control is automatically returned to the normal sequence.

The construct of this verb has the following four options:

Option 1:

```
PERFORM procedure-name-1 [ { THRU }  
                          { THROUGH } procedure-name-2 ]
```

Option 2:

```
PERFORM procedure-name-1 [ { THRU }  
                          { THROUGH } procedure-name-2 ]  
  
      { integer-1  
      { identifier-10 } TIMES
```

Option 3:

```
PERFORM procedure-name-1 [ { THRU }  
                          { THROUGH } procedure-name-2 ]  
  
      UNTIL condition-1
```

Option 4:

```
PERFORM procedure-name-1 [ { THRU }  
                          { THROUGH } procedure-name-2 ]  
  
VARYING { index-name-1 } FROM { index-name-2 }  
          { identifier-1 }      { identifier-2 }      BY  
                               { literal-2 }  
  
{ identifier-3 }      UNTIL condition-1 [ AFTER { index-name-4 }  
{ literal-3 }                                               { identifier-4 }  
  
FROM { index-name-5 }  
      { identifier-5 }      BY { identifier-6 }  
      { literal-5 }         { literal-6 }
```

```

UNTIL condition-2 ] [ AFTER { index-name-7 }
                        { identifier-7 } FROM
{ index-name-8 }
{ identifier-8 } BY { identifier-9 }
{ literal-8 }      { literal-9 }
UNTIL condition-3 ]

```

PERFORM is the means by which subroutines are executed in COBOL. The subroutines may be executed once, or a number of times, as determined by a variety of controls. A given paragraph may be PERFORMed by itself, in conjunction with another paragraph, control may pass through it in sequential operation, and it may be the object of a GO statement, all in the same program.

Each identifier represents a numeric elementary item. Identifier-10 must be described as an integer.

Each literal represents a numeric literal.

When the PERFORM statement is executed, control is transferred to the first statement of procedure-name-1. An automatic return to the statement following the PERFORM statement is established as follows:

- a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return occurs after the last statement of procedure-name-1.
- b. If procedure-name-1 is a section name and procedure-name-2 is not specified, then the return occurs after the last statement of the last paragraph in procedure-name-1.
- c. If the procedure-name-2 is specified and it is a paragraph name, then the return occurs after the last statement of the paragraph.
- d. If the procedure-name-2 is specified and it is a section name, then the return occurs after the last sentence of the last paragraph in the section.

There is no necessary relationship between procedure-name-1 and procedure-name-2, except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. In particular, GO TO and PERFORM statements may occur between procedure-name-1 and the end of

PERFORM

procedure-name-2. If there are two or more direct paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement, to which all of these paths must lead.

If control passes to these procedures by means other than a PERFORM statement, control passes thru the last statement of the procedure to the following statement, unless a PERFORM statement is executed during execution of these procedures.

If a statement within procedure-name-1 or procedure-name-2 contains a nested PERFORM, object program control will pass to the procedure-name contained in the nested statement, and the procedure will be accomplished. Program control will automatically return to the next sentence following the executed PERFORM statement. Nested PERFORM statements are allowed to any reasonable depth. However, the procedure named must return to the statement following the previously executed PERFORM and cannot contain a GO TO out of range of procedure-name-1 or procedure-name-2.

A PERFORM statement is not restricted by overlayable segment boundaries and may reference a procedure-name anywhere within the PROCEDURE DIVISION.

Option 1 is the basic PERFORM statement. A procedure referred to by this type of PERFORM statement is executed once, and then control passes to the statement following the PERFORM statement.

Option 2 is the TIMES option and, when used, the procedures are performed the number of times specified by identifier-10 or integer-1. The value of identifier-10 or integer-1 must be positive. Control is transferred to the statement following the PERFORM statement. If the value is zero, control passes immediately to the statement following the PERFORM sentence. Once the PERFORM statement has been initiated, any reference to, or manipulation of, identifier-10 will not affect the number of times the procedures are executed.

Option 3 is the UNTIL option. The specified procedures are performed until the condition specified by the UNTIL condition is TRUE. At this time, control is transferred to the statement following the PERFORM statement. If the condition is TRUE at the time that the PERFORM statement is encountered, the specified procedure is not executed.

In option 4, when one identifier is varied, identifier-1 is set equal to the current value of identifier-2, or literal-2. If the condition is false, the sequence of procedures, procedure-name-1 thru procedure-name-2, is executed once. The value of identifier-1 is augmented by the specified increment or decrement (identifier-3), and condition-1 is evaluated again. The

PERFORM

cycle continues until this expression is true; at this point, control passes to the statement following the PERFORM statement. If the condition is true at the beginning of execution of the PERFORM, control passes directly to the statement following the PERFORM statement. Figure 7-2 illustrates the logic of the PERFORM statement when one identifier is varied.

In option 4, when two identifiers are varied, identifier-1 and identifier-4 are set to the current value of identifier-2 and identifier-5, respectively. At the start of the PERFORM statement, condition-1 is evaluated; if true, control is passed to the statement following the PERFORM statement; if false, condition-2 is evaluated. If condition-2 is false, procedure-name-1 thru procedure-name-2 is executed once, after which identifier-4 is augmented by identifier-6, and condition-2 is evaluated again. The cycle of execution and augmentation continues until this condition is true. When condition-2 is true, identifier-4 is set to the current value of identifier-5; identifier-1 is augmented by identifier-3, and condition-1 is re-evaluated. The PERFORM statement is completed if condition-1 is true; if not, the cycles continue until condition-1 is true.

Figure 7-3 illustrates the logic of the PERFORM statement when two identifiers are varied.

During the execution of the procedures associated with the PERFORM statement, any change to the VARYING variable (identifier-1 and index-name-1), the BY variable (identifier-3), the AFTER variable (identifier-4 and index-name-4), of the FROM variable (identifier-2, index-name-2, identifier-5 and index-name-5) will be taken into consideration and will affect the operation of the PERFORM statement.

When two identifiers are varied, identifier-4 goes thru a complete cycle (FROM, BY, UNTIL) each time identifier-1 is varied.

At the termination of the PERFORM statement, identifier-4 contains the current value of identifier-5. Identifier-1 has a value that exceeds the last used setting by an increment or decrement, as the case may be, unless condition-1 was true when the PERFORM statement was entered, in which case identifier-1 contains the current value of identifier-2.

NOTE

Identifier-3, identifier-6, and identifier-9 cannot contain zeroes.

PERFORM

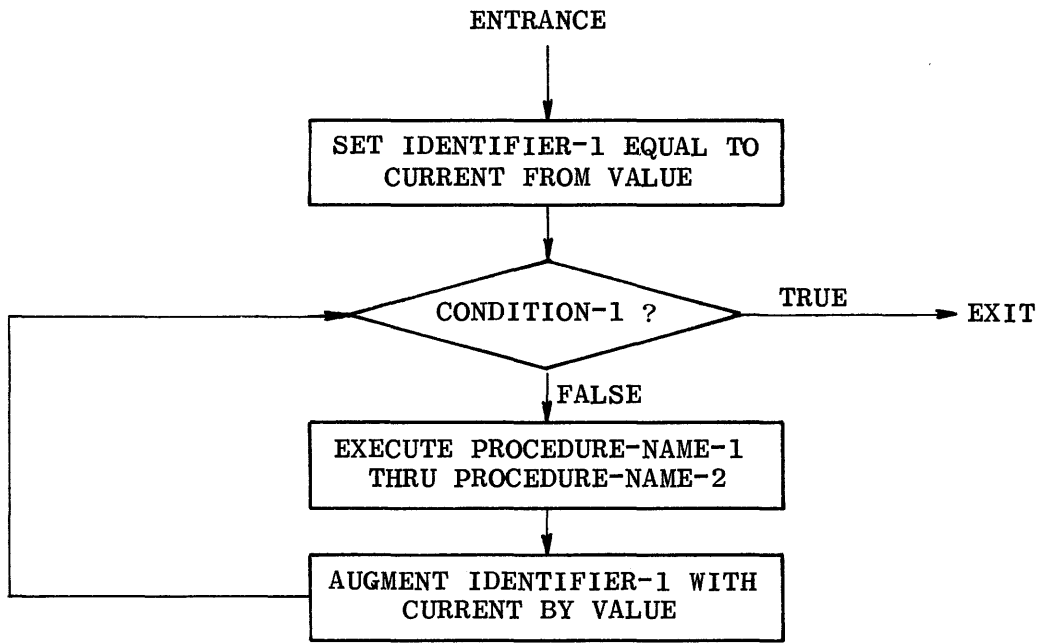


Figure 7-2. PERFORM Statement Varying One Identifier

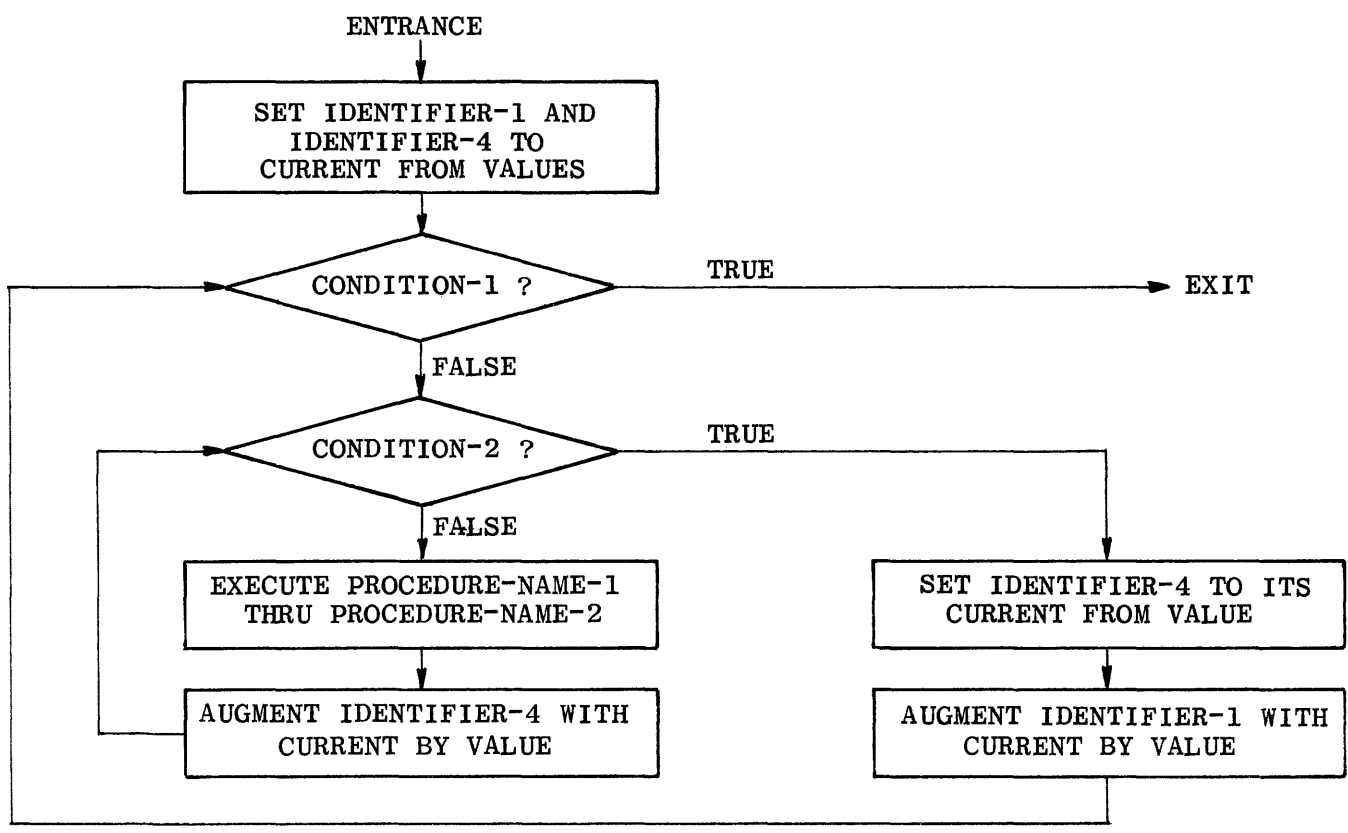
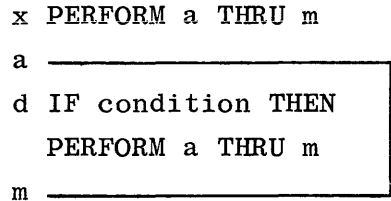
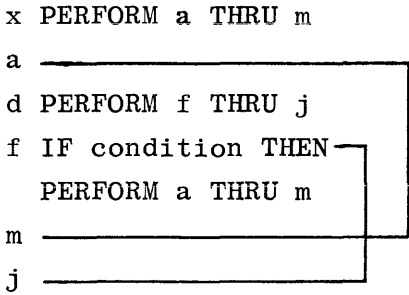
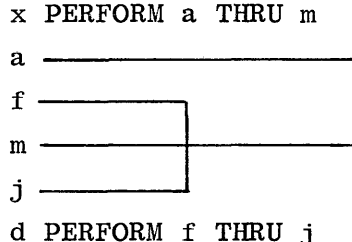
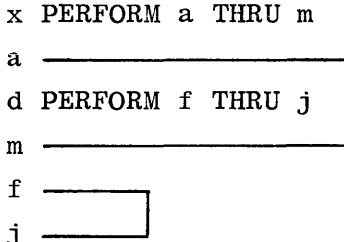
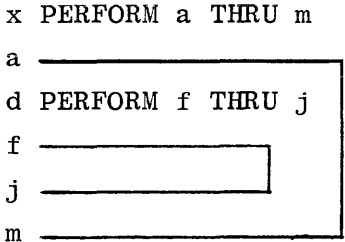


Figure 7-3. PERFORM Statement Varying Two Identifiers

In Option 4 where three conditions are required to control the number of iterations that a given procedure is to be PERFORMed, the mechanism is the same as for two-conditional control except that identifier-7 goes through a complete cycle each time that identifier-6 is added to identifier-4, which in turn goes through a complete cycle each time that identifier-1 is varied.

After the completion of option 4, identifier-4 and identifier-7 contain the current value of identifier-5 and identifier-8, respectively. Identifier-1 has a value that exceeds its last used setting by one increment or decrement value, unless condition-1 is true when the PERFORM statement is entered, in which case identifier-1 contains the current value of identifier-2.

Since the return control information is placed in the stack rather than being directed through instruction address modification, a PERFORM statement executed within the range of another PERFORM is not restricted in the range of paragraph names it may include. The examples shown below are permitted and will execute correctly.



READ

The functions of this verb are twofold, namely:

- a. During processing of sequential input files, a READ statement will cause the next sequential logical record to be moved from the input buffer area to the record work area, thus making the record available to the program.

All sequential records will be physically read into the buffer area of the file. Physical READs are performed as a function of the MCP. The READ statement permits the performance of a specified statement when an end-of-file condition is detected by the MCP.

- b. For random file processing, the READ statement communicates with the MCP to explicitly cause the reading of a physical record from a disk file, and also allows performance of a specified statement if the content of the associated ACTUAL KEY data item is found to be invalid.

The construct of this verb is:

```

READ file-name RECORD [INTO identifier] [; {AT END } statement-1
      [INVALID KEY }
      [ELSE statement-2] ]
  
```

The AT END clause is used for non-disk files or for disk files being processed in the sequential access mode. If no AT END or INVALID KEY clause is stated, and one of these conditions occurs, the program will be terminated with a DS or DP message.

If, during execution of a READ statement with AT END, the logical end-of-file is reached and an attempt is made to READ that file, the statement specified in the AT END phrase is executed. After the execution of the statement of the AT END phrase, a READ statement for that file must not be given without prior execution of a CLOSE statement and an OPEN statement for that file.

When the AT END clause is specified in a conditional sentence, all exits within the sentence are controlled by using the rules pertaining to the matching of IF...ELSE pairs. For example:

```

IF AAA = BBB THEN READ FILE-A, AT END
      GO TO WRAP-UP, ELSE NEXT SENTENCE, ELSE STOP RUN.
  
```

- a. When AAA does not equal BBB, control will be passed to STOP RUN.
- b. When AAA equals BBB, FILE-A is read, end-of-file is tested and if the result is TRUE program control will be transferred to the WRAP-UP procedure; however, a result of FALSE will cause program control to be transferred to the next sentence.

The INVALID KEY applies to files that are ASSIGNED to disk. The access of the file is controlled by the value contained in ACTUAL KEY.

An AT END or INVALID KEY clause must be specified when reading a file described as containing FILE-LIMITS.

An OPEN statement must be executed for a file prior to the execution of the first READ statement for that file.

When a file consists of more than one type of logical record, these records automatically share the same storage area and are equivalent to an implicit redefinition of the area. Only the information that is present in the current record is available.

If the INTO option is specified, the current record is MOVED from the input area to the area specified by identifier according to the rules for the MOVE statement without the CORRESPONDING option.

When the INTO option is used, the record being read is available in both the data area associated with identifier and the input record area.

If a file described with the OPTIONAL clause is not present, the imperative statement in the AT END phrase is executed on the first READ. The standard End-of-File procedures are not performed. (See the OPEN and USE statements, and the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION.)

If the end of a magnetic tape file is recognized during execution of a READ statement, the following operations are carried out:

- a. The standard ending reel label procedure and the user's ending reel label procedure, if specified by the USE statement, are performed. The order of execution of these two procedures is specified by the USE statement.
- b. A tape swap is performed.

READ

- c. The standard beginning reel label procedure and the user's beginning label procedure, if specified, are executed. The order of execution is again specified by the USE statement.
- d. The first data record on the new reel is made available.

READ with INVALID KEY is used for disk files in the random access mode. The READ statement implicitly performs the functions of the SEEK statement, except for the function of the KEY CONVERSION option for a specific disk file. If the content of the associated ACTUAL KEY data item is out of the range indicated by FILE-LIMITS or the FILE CONTAINS clause, the INVALID KEY phrase will be executed.

For random disk files, the sensing of an INVALID KEY does not preclude further READs on that file, nor must the file be closed and reopened before such READs are allowed.

RELEASE

The function of this verb is to cause records to be transferred to the initial phase of a SORT operation.

The construct of this verb is:

RELEASE record-name [FROM identifier]

A RELEASE statement may only be used within the range of an input procedure associated with a SORT statement.

Record-name and identifier must name different memory areas when specified.

The RELEASE statement causes the contents of record-name to be released to the initial phase of a sort. Record-name will be transferred to the specified sort-file (SD) and becomes controlled by the sort operation.

In the FROM option, the contents of identifier are MOVED to record-name, then the contents of record-name are released to the initial phase of a sort. Moving takes place according to the rules specified for the MOVE statement without the CORRESPONDING option.

When control passes from the input procedure, the SD file consists of all records placed in it by the execution of RELEASE statements.

RETURN

RETURN

The function of this verb is to obtain sorted records from the final phase of a SORT operation.

The construct of this verb is:

```
RETURN file-name RECORD [INTO identifier]  
[; AT END statement-1 [; ELSE statement-2]]
```

File-name must be a sort file with a Sort File Description (SD) entry in the DATA DIVISION.

A RETURN statement may only be used within the range of an output procedure associated with a SORT statement for file-name.

Records automatically share the same area when a file consists of more than one type record and only the information pertinent to the current record is available.

The execution of the RETURN statement causes the next record, in the order specified by the keys listed in the SORT statement, to be made available for processing in the record area associated with the SORT file (SD).

Moving is performed according to the rules specified for the MOVE statement without the CORRESPONDING option.

When the INTO option is specified, the sorted data is available in both the input-record area and the data-area specified by identifier.

RETURN statements may not be executed within the current SORT input procedure after the AT END clause has been executed.

SEARCH

The function of this verb is to cause a search of a table to locate a table-element that satisfies a specific condition and, in turn, to adjust the associated index-name to indicate that table-element.

The construct of this verb has the following two options:

Option 1:

```

SEARCH identifier-1 [ VARYING {index-name-1}
                    {identifier-2} ]
                    [;AT END imperative-statement-1]
                    ;WHEN condition-1 {statement-1}
                                        {NEXT SENTENCE}
                    [;WHEN condition-2 {statement-2}
                                        {NEXT SENTENCE}]

```

Option 2:

```

SEARCH ALL identifier-3 [;AT END imperative statement-2]
                    ;WHEN condition-3 {statement-3}
                                        {NEXT SENTENCE}

```

Identifier-1 and identifier-3 must not be subscripted or indexed, but its description in the DATA DIVISION must contain an OCCURS clause and an INDEXED BY clause.

When Option 2 is specified, the description of identifier-3 may optionally contain the ASCENDING/DESCENDING KEY clause.

When the VARYING option is used, identifier-2 must be described as USAGE IS INDEX, or as the name of a numeric elementary item described without any positions to the right of the assumed decimal point. Identifier-2 will be incremented at the same time as the occurrence number (and by the same amount) represented by the index-name associated with identifier-1.

When Option 1 is used, condition-1, condition-2, etc., may be comprised of any conditional as described by the IF verb.

When Option 2 is used, condition-3 may consist of a relational condition incorporating the relation EQUAL, or a condition-name condition where the VALUE clause that describes the condition-name contains only a single literal. Condition-3 may be a compound condition formed from simple conditions of the type just mentioned, with AND being the only acceptable connective.

SEARCH

When Option 2 is used, any data-name that appears in the KEY option of identifier-3 may appear as the subject or object of a test, or be the name of the conditional variable with which the tested condition-name is associated.

When Option 1 is used, a serial type search operation takes place, starting with the current index setting. The search is immediately terminated if, at the start of execution of the statement, the index-name associated with data-identifier-1 contains a value that corresponds to an occurrence number that is greater than the highest permissible occurrence number for identifier-1. Then, if the AT END option is specified, statement-1 is executed; if AT END is not specified, control passes to the NEXT SENTENCE.

When Option 1 is used, if at the start of execution of the SEARCH statement, the index-name associated with identifier-1 contains a value that corresponds to an occurrence number that is not greater than the highest permissible occurrence number for identifier-1, the SEARCH statement will begin evaluating the conditions in the order that they are written, making use of index settings wherever specified, to determine the occurrences of those items to be tested. If none of the conditions are satisfied, the index-name for identifier-1 is incremented to obtain a reference to the next occurrence. The process is repeated using the new index-name setting for identifier-1, which corresponds to a table element which exceeds the last setting by one more occurrence, until such time as the highest permissible occurrence number is exceeded, in which case the SEARCH terminates as indicated in the previous paragraph.

When Option 1 is used, if one of the conditions is satisfied upon its evaluation, the SEARCH terminates immediately and the statement associated with that condition is executed; the index-name remains set at the occurrence which caused the condition to be satisfied.

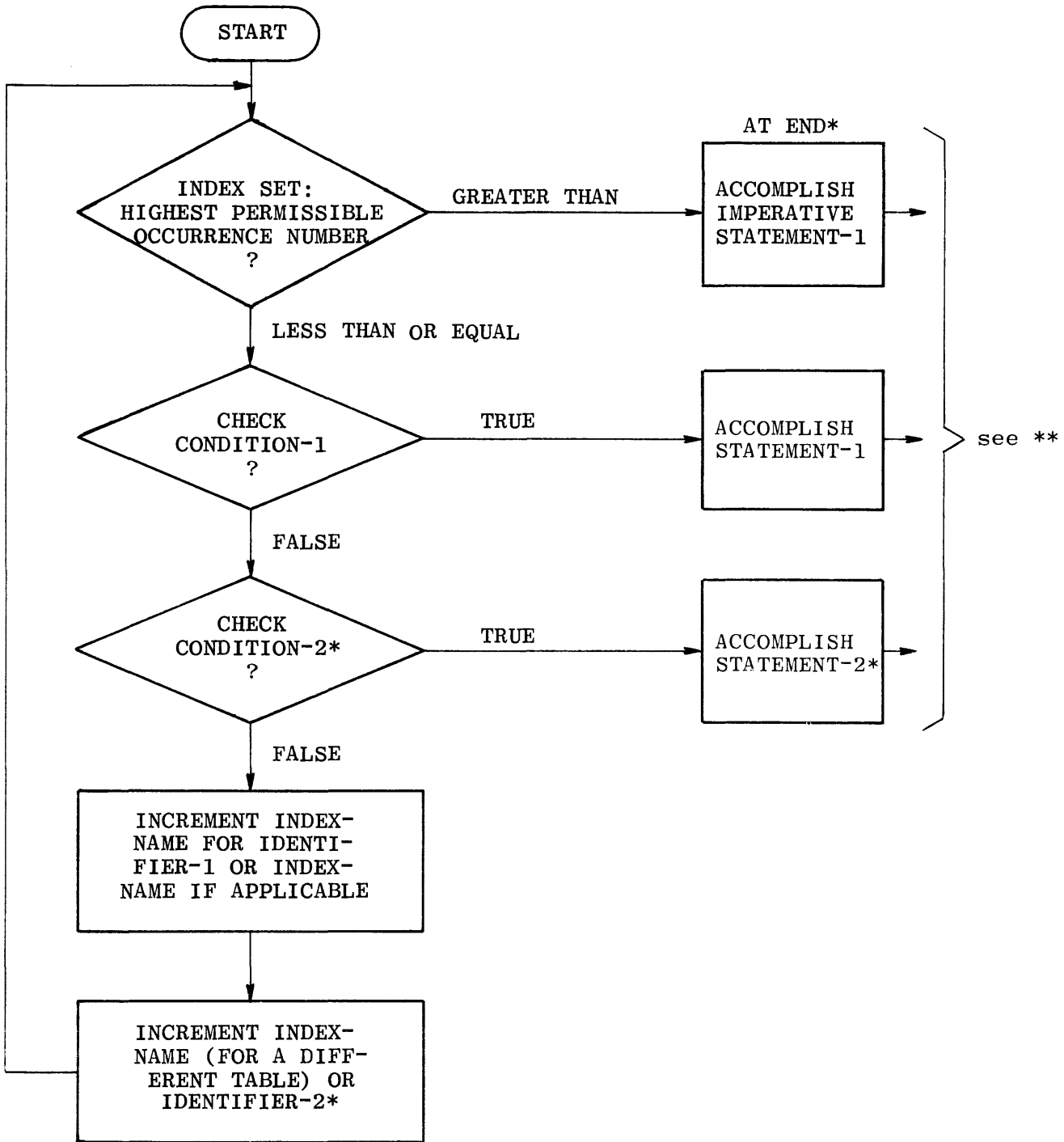
In Options 1 and 2, if the specified imperative statements do not terminate with a GO statement, then program control will pass to the next sentence, after the execution of the imperative statement.

In the VARYING option, if index-name-1 appears in the INDEXED BY option of identifier-1, then that index-name will be used for the SEARCH; otherwise, the first index-name given in the INDEXED BY option of identifier-1 will be used. If index-name-1 appears in the INDEXED BY clause of another table entry, the occurrence number represented by index-name-1 is incremented by the same amount as, and at the same time as, the occurrence number represented by the index-name associated with identifier-1 is incremented.

In Option 2, the initial setting of the index-name for identifier-3 is ignored, the effect being the same as if it were SET to 1.

In Options 1 and 2, if identifier-1 and identifier-3 constitute an item in a group, or a hierarchy of groups, whose description contains an OCCURS clause, then each of these groups must also have an index-name associated with it. The settings of these index-names are used throughout the execution of the SEARCH statement to refer to identifier-1 and identifier-3, or to items within its structure. These index settings are not modified by the execution of the SEARCH statement (unless stated as index-name-1), and only the index-name associated with identifier-1 and identifier-3 (and identifier-2 or index-name-1) is incremented by the SEARCH. Figure 7-4 provides an example of SEARCH operation as related to Option 1.

SEARCH



* These operations are only included when called for in the SEARCH statement.

** Each of the control transfers is to NEXT SENTENCE unless the statement ends with a GO statement.

Figure 7-4. Example of Option 1 SEARCH Statement

SEEK

The function of this verb is to initiate the accessing of a disk file record for subsequent reading and/or writing. The construct of this verb is:

SEEK file-name RECORD [WITH KEY CONVERSION]

The specification of the KEY CONVERSION clause indicates that the user-provided USE FOR KEY CONVERSION section in the DECLARATIVE SECTION is to be executed prior to the execution of the SEEK statement. If there are no DECLARATIVES for KEY CONVERSION in a SEEK statement, then the KEY CONVERSION clause will be ignored.

A SEEK statement pertains only to disk storage files in the random access mode and may be executed prior to the execution of each READ and WRITE statement.

The SEEK statement uses the contents of the data-name in the ACTUAL KEY clause as the location of the record to be accessed. At the time of execution, the determination is made as to the validity of the contents of the ACTUAL KEY data item for the particular disk storage file. If the key is invalid, the imperative statement in the INVALID KEY clause of the next executed READ or WRITE statement for the associated file is executed.

Two SEEK statements for a disk storage file may logically follow each other. Any validity check associated with the first SEEK statement is negated by the execution of a second implicit or implied SEEK statement.

An implied SEEK is executed by the MCP whenever an explicit SEEK is missing for the specified record. An implied SEEK never executes any USE KEY CONVERSION Declaratives.

If a READ/WRITE statement for a file ASSIGNED to DISK is executed, but an explicit SEEK has not been executed since the last previous READ or WRITE for the file, then the implied SEEK statement is executed as the first step of the READ/WRITE statement.

An explicit alteration of ACTUAL KEY after the execution of an explicit SEEK has been performed, but prior to a READ/WRITE, will cause the initiation of an implied SEEK of the initial record in the sequence. For example,

- a. If ACTUAL KEY is 10, then
- b. READ record 10, then
- c. MOVE 50 to ACTUAL KEY, then
- d. WRITE record 50.

An implied SEEK of record 50 will be performed between actions c and d, above.

SET

SET

The SET statement establishes reference points or offsets operations by setting index-names associated with table elements.

The construct of this verb has the following two options:

Option 1:

$$\underline{\text{SET}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{index-name-1} \end{array} \right\} \left[, \left\{ \begin{array}{l} \text{identifier-2} \\ \text{index-name-2} \end{array} \right\} \right] \dots \underline{\text{TO}} \left\{ \begin{array}{l} \text{identifier-3} \\ \text{index-name-3} \\ \text{integer-1} \end{array} \right\}$$

Option 2:

$$\underline{\text{SET}} \text{ index-name-4 } [, \text{ index-name-5}] \dots \left\{ \begin{array}{l} \underline{\text{UP BY}} \\ \underline{\text{DOWN BY}} \end{array} \right\} \left\{ \begin{array}{l} \text{identifier-4} \\ \text{integer-2} \end{array} \right\}$$

All references to identifier-1 and index-name-1 apply equally to identifier-2 and index-name-2, respectively.

All identifiers must name either index data items, or elementary items described as an integer, except that identifier-4 must not name an index data item. When integer-1 is used, it must be a positive integer. Index-names are considered related to a given table and are defined by being specified in the INDEXED BY phrase of the OCCURS clause.

If index-name-3 is specified, the value of the index before the execution of the SET statement must correspond to an occurrence number of an element in the associated table.

If index-name-1, index-name-2 is specified, the value of the index after the execution of the SET statement must correspond to an occurrence number of an element in the associated table. The value of the index associated with an index-name after the execution of a SEARCH or PERFORM statement may be undefined.

In option 1, the following action occurs:

- a. Index-name-1 is set to a value causing it to refer to the table element that corresponds in occurrence number to the table element referenced by index-name-3, identifier-3, or integer-1. If identifier-3 is an index data item, or if index-name-3 is related to the same table as index-name-1, no conversion takes place.

SET

- b. If identifier-1 is an index data item, it may be set equal to either the contents of index-name-3 or identifier-3 where identifier-3 is also an index data item; no conversion takes place in either case.
- c. If identifier-1 is not an index data item, it may be set only to an occurrence number that corresponds to the value of index-name-3. Neither identifier-3 nor integer-1 can be used in this case.
- d. The process is repeated for index-name-2, identifier-2, etc., if specified. Each time, the value of index-name-3 or identifier-3 is used as it was at the beginning of the execution of the statement. Any subscripting or indexing associated with identifier-1, etc., is evaluated immediately before the value of the respective data item is changed.

In option 2, the contents of index-name-4 are incremented (UP BY) or decremented (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of integer-2 or identifier-4; thereafter, the process is repeated for index-name-5, etc. Each time the value of identifier-4 is used as it was at the beginning of the execution of the statement.

Data in the figure 7-5 represents the validity of various operand combinations in the SET statement. The parenthetical comment references the lettered paragraphs above.

SENDING ITEM	RECEIVING ITEM		
	INTEGER DATA ITEM	INDEX-NAME	INDEX DATA ITEM
Integer Literal	No (c)	Valid (a)	No (b)
Integer Data Item	No (c)	Valid (a)	No (b)
Index-Name	Valid (c)	Valid (a)	Valid (b)*
Index Data Item	No (c)	Valid (a)*	Valid (b)*
*No conversion takes place.			

Figure 7-5. SET Statement Operand Combinations

SORT

SORT

The function of this verb is to sort an input file of records by transferring such data into sort work files and sorting those records on a set of specified keys. The final phase of the sort operation makes each record available from the sort-file, in sorted order, to an output procedure or to an output file.

The construct of this verb is:

```
SORT  [TAG-SEARCH]      [INPLACE]      file-name-1
      [TAG-KEY]
[WITH file-name-4]  [RESTART WITH data-name-5]
[ [PURGE] .ON ERROR ]
      ON {DESCENDING} KEY data-name-1 [,data-name-2] ...
      [ ON {DESCENDING} KEY data-name-3 [,data-name-4] ... ] ...
{ [INPUT PROCEDURE IS section-name-1  [ {THRU} section-name-2] ]
  [USING file-name-2  [LOCK
                     PURGE
                     RELEASE] ] }
{ [OUTPUT PROCEDURE IS section-name-3  [ {THRU} section-name-4] ]
  [GIVING file-name-3  [LOCK
                       RELEASE] ] }
```

When the TAG-KEY option is used, keys and relative record locations are extracted from the input file, in order for sorting to be performed using only the necessary data rather than manipulating the entire record. The relative record locations are written out in sorted order by the final phase of the sort. The record size of the output file must be 8 digits.

When the TAG-SEARCH option is used, a sort is performed as in the TAG-KEY option, with the exception that the final phase makes available the entire record rather than simply the relative record location.

When the INPLACE option is used, the amount of disk space required for sorting is minimized. The record sizes for file-name-1, file-name-2, and file-name-3 must be the same. The INPLACE option is invalid for sorts using tape work files.

When the RESTART option is used, the program is allowed to restart the sort from the last completed pass of the sort. Data-name-5 must be declared as PIC 9(6). Prior to the execution of a SORT statement containing the RESTART option, data-name-5 must contain the value zero, or a six digit job number. If data-name-5 contains zero the normal sort takes place, otherwise it is assumed to contain the job number of the discontinued sort, and a RESTART is attempted. The job number contained in data-name-5 must be the job number of the sort which failed, not the job number of the calling program. The job number is used to locate the sort work files which were created during the original execution of the SORT statement in order for the sort process to continue as if the discontinuation had not occurred. If a RESTART is required, it will be the responsibility of the program logic to transfer control to the SORT statement which failed.

File-name-4 is used to invoke the virtual collating sequence capability of the SORT. File-name-4 must describe a file which exists on disk, and was created by the utility program SORT/COLLATE.

File-name-1 must be described in a Sort File Description (SD) entry in the DATA DIVISION, and file-name-2, file-name-3, and file-name-4 must be described in a File Description (FD) entry.

The ON ERROR option is provided to allow programmers some control over irrecoverable I/O errors when input/output procedures are not present in a program. When PURGE ON ERROR is specified, all records in a block containing an irrecoverable I/O error are dropped, and processing is continued until the final phase of the sort. At that point, a console message will be printed and the choice to continue or discontinue the sort must be made by the system operator. If the ON ERROR option is not specified, or END ON ERROR is specified and an I/O error occurs, a DS or DP console printer message will be issued to the system operator.

Section-name-1 specifies the name of the input procedure to be used before each record is passed to the sort-file, and section-name-3 specifies the output procedure to be used to obtain each sorted record from the sort-file.

SORT

Each data-name must represent data-items described in records associated with file-name-1. Data-names following the word KEY are listed from left to right, in the order of decreasing significance, without regard to their division into optional KEY clauses.

The PROCEDURE DIVISION of a source program may contain more than one SORT statement appearing anywhere in the program, except in the DECLARATIVES portion or in the input/output procedures associated with a SORT statement.

The input procedure must consist of one or more sections that are written consecutively and which do not form a part of an output procedure. The input procedure must include at least one RELEASE statement in order to transfer records to the sort-file after the object program has accomplished the required input data manipulation specified in the procedure. Input procedures can select, create and/or modify records, one at a time, as specified by the programmer.

There are three restrictions placed on procedural statements within an input or output procedure:

- a. The procedure must not contain any SORT statements.
- b. The input or output procedures must not contain any transfers of program control outside the range of the procedure; ALTER, GO and PERFORM statements within the procedure are not permitted to refer to procedure-names outside of the input or output procedure.
- c. The remainder of the PROCEDURE DIVISION must not contain any transfers of program control to points within the input or output procedure; ALTER, GO, and PERFORM statements in the remainder of the PROCEDURE DIVISION must not refer to procedure-names within the range of the input or output procedure.

The output procedure must consist of one or more sections that are written consecutively and which do not form a part of an input procedure. The output procedure must include at least one RETURN statement in order to make each sorted record available for processing. Output procedures can select, create, and/or modify records, one at a time, as they are being returned from the sort-file.

When the ASCENDING clause is specified, the sorted sequence of the affected records is from the lowest to the highest value, according to the binary EBCDIC collating sequence (or as specified in the virtual collating sequence.)

When the DESCENDING clause is specified, the sorted sequence of the affected records is from the highest to the lowest value according to the binary EBCDIC collating sequence (or as specified in the virtual collating sequence).

The SD record description of the sort-file must contain fully defined data-name KEY items in the relative positions of the record, as applicable. A rule to follow when using these KEY items is that when a KEY item appears in more than one type of record, the data-names must be relatively equivalent in each record and may not contain, or be subordinate to, entries containing an OCCURS clause.

When an INPUT procedure is specified, object-program control will be passed to that procedure automatically as an implicit function of encountering the generated SORT verb object code compiled into the program. The compiler will insert a "return-to-the-sort" mechanism at the end of the last section in the input procedure, and when program control passes the last statement of the input procedure, the records that have been RELEASED to file-name-1 are sorted.

If the USING option is specified, all records residing in file-name-2 will be automatically transferred to file-name-1, upon encountering the generated SORT verb object code. At the time of execution of the SORT statement, file-name-2 must have been CLOSED WITH LOCK or RELEASE. The SORT statement automatically performs the function necessary to OPEN, READ, USE and CLOSE file-name-2. If file-name-2 is a disk file, it must be in the Disk Directory before the SORT intrinsic is called.

If an output procedure is specified, object-program control will be passed to that procedure automatically as an implicit function when all records have become sorted. The compiler will insert a "return-to-the-object program" mechanism at the end of the last section in the output procedure; and when program control passes the last statement of the output procedure, the object program will execute the next statement following the pertinent SORT statement.

If the GIVING option is specified, all sorted records residing in file-name-1 are automatically transferred to the OUTPUT file as specified in file-name-3. At the time of execution of the SORT statement, file-name-3 must have been CLOSED WITH LOCK or RELEASE. File-name-3 will be automatically OPENED before the sorted records are transferred from the sort-file and, in turn, will be automatically CLOSED, default is LOCK, after the last record in the sort-file has been transferred.

SORT

The CLOSE options LOCK or RELEASE may be used to specify the type of close for file-name-3. In addition to the LOCK or RELEASE, the PURGE option may also be used for file-name-2. Refer to description of CLOSE verb in this section. The options only apply to the USING/GIVING options.

Example:

```
SORT file-name-1 ASCENDING KEY data-name-1  
USING file-name-2 PURGE  
GIVING file-name-3 LOCK.
```

Beginning and ending label USE procedures are provided as follows when input/output procedures are present in the SORT statement:

- a. OPEN INPUT file-name.
USE. . . (The programmer's USE procedure will be invoked).
- b. OPEN OUTPUT file-name.
USE. . . (The programmer's USE procedure will be invoked).
- c. CLOSE INPUT file-name.
USE. . . (The programmer's USE procedure will be invoked; however, the contents of the ending input label will not be available to the USE procedure).
- d. CLOSE OUTPUT file-name.
USE. . . (The programmer's USE procedure will be invoked; however, the ending label will have been written prior to execution of the USE procedure).

NOTE

The above action provide label USE procedures at beginning and ending of files, but not during switching of reels of multi-reel files.

STOP

The function of this verb is to halt the object program temporarily or to terminate execution.

The construct of this verb is:

STOP { RUN }
 { literal }

If the word RUN is used, then all files which remain OPEN will be CLOSED automatically. New files ASSIGNED to DISK will be CLOSED WITH PURGE and all others will be CLOSED WITH RELEASE. See the SAVE option in the FILE-CONTROL paragraph. All storage areas for the object program are returned to the MCP and the job is then removed from the MCP mix.

The STOP RUN is not used for temporary stops within a program. STOP RUN must be the last statement of the program execution sequence.

A sequence of imperative statements may contain a STOP RUN statement which, if present, must appear as the last imperative statement of the sequence.

If the literal option is used, the literal will be DISPLAYed on the console printer and the program will be suspended. When the operator enters the MCP continuation message mix-index AX, program execution resumes with the next sequential operation. This option is normally used for operational halts to cause the system's operator to physically accomplish an external action.

SUBTRACT

SUBTRACT

The function of this verb is to subtract one data item, or the sum of two or more, numeric data items from another item, and set the value of an item equal to the result(s).

The construct of this verb has the following three options:

Option 1:

```
SUBTRACT {literal-1 } [ {literal-2 } ... ] FROM  
  {identifier-1} [ {identifier-2} ... ]  
  {identifier-m} [ROUNDED] [ {identifier-n} [ROUNDED] ... ]  
  [ ; ON SIZE ERROR statement-1 [ ; ELSE statement-2 ] ]
```

Option 2:

```
SUBTRACT {literal-1 } [ {literal-2 } ... ] FROM  
  {literal-m }  
  {identifier-m} GIVING {identifier-n} [ROUNDED] [ , {identifier-o} [ROUNDED] ] ...  
  [ ; ON SIZE ERROR statement-1 [ ; ELSE statement-2 ] ]
```

Option 3:

```
SUBTRACT { CORR  
  { CORRESPONDING } } {identifier-1} FROM {identifier-2}  
  [ROUNDED] [ ; ON SIZE ERROR statement-1 [ ; ELSE statement -2 ] ]
```

In Options 1 and 2, the identifiers used must refer only to elementary numeric items. If Option 2 is used, the data-description of identifier-n and identifier-o may be an elementary numeric edited item.

All rules specified under the ADD statement with respect to the operand size, presence of editing symbols in operands, the ON SIZE ERROR option, the ROUNDED option, the GIVING option, truncation, the editing results, the handling of intermediate results, and the CORR or CORRESPONDING option apply to the SUBTRACT statement.

When the GIVING option is not used, a literal may not be specified as the minuend

When dealing with multiple subtrahends, the effect of the subtraction will be as if the subtrahends were first summed, and then the sum subtracted from the minuends.

TRACE

The function of this verb is to create documentation of all normal and/or control mode processing events and to output this data on a line printer.

The construct of this verb is:

TRACE 20

When a TRACE statement is encountered during object-program execution, the following actions will take place at that point in the program.

The 20 option will cause a memory dump to be taken of locations that are base relative to the program's memory assignment. Processing will continue after the memory "snapshot."

Within a given format, a file-name must not be referred to implicitly or explicitly in more than one USE statement.

USE procedures will be executed by the MCP:

- a. After completion of the standard I/O error retry routine (this applies only to option 1), the record in error has been read; therefore, another READ cannot appear in the USE section, since the MCP is performing the section because of a previous READ which has been completed. Upon completion of the USE procedure, control is returned to the statement following the READ which detected the error condition.
- b. Prior to any SEEK WITH KEY CONVERSION statement on files named in the USE FOR KEY CONVERSION statement.
- c. When USE FOR Q-EMPTY is specified for a file and a READ is requested for that file, the procedure following this clause will be performed if the queue is empty.
- d. When USE FOR Q-FULL is specified for a file and a WRITE is requested for that file, the procedure following this clause will be performed if the addition of this message to the queue would cause the number of messages contained in the queue to exceed the value of Q-MAX-MESSAGES. (See the File Description information for a further description of Q-MAX-MESSAGES).

There must not be any reference to non-declarative procedures within a USE procedure. Conversely, in the non-declarative portion there must be no reference to procedure-names that appear in the declarative portion, except that a PERFORM statement may refer to a USE declarative or to the procedures associated with such USE declaratives.

NOTE

USE AFTER STANDARD ENDING and USE BEFORE
STANDARD BEGINNING are both illegal entries
in B 1800/B 1700 COBOL.

WAIT

WAIT

The function of this verb is to suspend an object program until a specified event occurs, and/or to suspend an object program for a specified number of tenths of a second.

The construct of this verb is:

WAIT UNTIL

{(identifier-1)}
{literal-1 }

[SPO-INPUT-PRESENT]

[WRITE-OK ON file-name-1 [(identifier-2)] [literal-2] ...]

[READ-OK ON file-name-2 [(identifier-3)] [literal-3] ...]

[USING identifier-4]

[GIVING identifier-5]

When the identifier-1/literal-1 option is used, identifier-1 or literal-1 will contain the number of tenths of seconds for the program to be suspended. The number must not be less than one or greater than 1048576. If the identifier-1/literal-1 option is used in conjunction with other options, it must be the first option in the statement.

The SPO-INPUT-PRESENT option causes the program execution to be suspended until an AX message is entered for that program.

The WRITE-OK option causes program execution to be suspended until there is space in file-name-1 for at least one more record. Identifier-2/literal-2 are used to specify queue number for queue families. For queue and remote files, this condition is true whenever the specified queue is not full. For serial I/O devices such as card punches or tape, the true condition means that there is space in the buffers for at least one record. For disk and backup files and files which are not OPENed, this event is always true.

The READ-OK option causes program execution to be suspended until there is at least one record in file-name-2 to be read. Identifier-3/literal-3 are used to specify queue number for queue families. For queue and remote files, this event is true when the queue specified is not empty. For serial I/O devices

such as card readers and tape, this event being true means that a record is present in the buffer. For disk and pseudo-readers and files which are not OPENed, this condition is always true.

When the USING option is specified, the value contained in identifier-4 is used to determine at which point in the specified list of events the WAIT should begin checking for a true condition. Identifier-4 is zero relative to the list of events coded in the WAIT statement; i.e. a value of zero in identifier-4 would cause the WAIT statement to begin checking with the first event in the list. From the beginning point in the list, events in the list are checked for a true condition until all events have been checked, or until the first true condition is found. If a true condition is found, the "WAIT" is terminated and control is passed to the next statement. If in this initial scan of the events, no events are found to be true, program execution is suspended until any event in the list becomes true, at which time, the "WAIT" statement is terminated and control is passed to the next statement.

When the GIVING option is specified, identifier-5 is set by the WAIT statement to reflect the event in the statement which terminated the WAIT. Identifier-5, like identifier-4 is zero relative; i.e., if the second event in the list of events is found to be true, identifier-5 will be set to a value of one. The maximum number of events which may be coded in a WAIT statement is nine.

WAIT

AN EXAMPLE OF THE "WAIT" STATEMENT.

```
IDENTIFICATION DIVISION.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT QUEUE-FILE ASSIGN TO QUEUE ACTUAL KEY QUEUE-KEY.  
DATA DIVISION.  
FILE SECTION.  
FD  QUEUE-FILE FILE CONTAINS 9 QUEUES  
    VALUE OF ID IS "A".  
01  QA-REC.  
    03  QA-NUM      PC 9(10).  
    03  QA-FILL    PC X(50).  
    03  FILLER     PC X(1360).  
WORKING-STORAGE SECTION.  
77  QUFUE-KEY     PC 9(8) COMP  VA 0.  
77  IDENTIFIER-1 PC 9(6) COMP  VA 200.  
77  IDENTIFIER-2 PC 99          VA 2.  
77  IDENTIFIER-3 PC 99          VA 3.  
77  IDENTIFIER-4 PC 9           VA 0.  
77  IDENTIFIER-5 PC 9           VA 0.  
PROCEDURE DIVISION.  
BEGIN.  
    OPEN I-O QUEUE-FILE.  
    WAIT UNTIL      IDENTIFIER-1  
                    SPO-INPUT-PRESENT  
                    WRITE-OK ON QUEUE-FILE ( IDENTIFIER-2 )  
                    READ-OK  ON QUEUE-FILE ( IDENTIFIER-3 )  
                    WRITE-OK ON QUEUE-FILE ( 8 )  
                    USING IDENTIFIER-4  
                    GIVING IDENTIFIER-5.  
    IF IDENTIFIER-5 = 0 DISPLAY "WAITED 20 SECONDS".  
    IF IDENTIFIER-5 = 1 DISPLAY "GOT SPO-INPUT-PRESENT".  
    IF IDENTIFIER-5 = 2 DISPLAY "2ND QUEUE HAS SPACE FOR A MSG".  
    IF IDENTIFIER-5 = 3 DISPLAY "3RD QUEUE HAS A MSG TO READ".  
    IF IDENTIFIER-5 = 4 DISPLAY "8TH QUEUE HAS SPACE FOR A MSG".  
    STOP RUN.  
END-OF-JOB.
```


WRITE

The function of this verb is to release a logical record for an output file. It is also used to vertically position forms in the printer. For mass storage files, the WRITE statement also allows the performance of a specified imperative statement if the contents of the associated ACTUAL KEY item are found to be invalid.

The construct of this verb has the following two options:

Option 1:

WRITE record-name [FROM identifier-1]

[{	<u>AFTER</u> <u>BEFORE</u>	}	ADVANCING	{	{ integer-1 identifier-2 }	LINES	}]
						TO <u>CHANNEL</u>			
[TO	{ <u>ERROR</u> <u>AUXILIARY</u> <u>STACKER</u> }		{	integer-3	}]]]
					identifier-4				
[;AT	{	<u>END-OF-PAGE</u> <u>EOP</u>	}	statement-1	[<u>ELSE</u>	statement-2]

Option 2:

WRITE record-name [FROM identifier-1]
[; INVALID KEY statement-1 [; ELSE statement-2]]

An OPEN statement for a file must be executed prior to execution of the first WRITE statement for that file.

The record-name must be defined in the DATA DIVISION by means of an 01 level entry under the FD entry for the file. The record-name and identifier-1 must not be the same name, or be in two files that have the same record area.

The ADVANCING option allows the control of vertical positioning of each record on the printed page. The options are as follows:

- a. When LINES is used, identifier-2 must be declared as PC 99 COMPUTATIONAL, or integer-1 must be a positive integral value of 00 thru 99.
- b. WRITE BEFORE ADVANCING is more efficient than AFTER ADVANCING.

WRITE

- c. When CHANNEL is used, identifier-3 or integer-2 must contain a positive integral value of 01 ... 11. Identifier-3 must be declared as PC 99 COMPUTATIONAL. The MCP will advance the line printer's carriage to the carriage control channel specified.
- d. If the ADVANCING option is not specified, single spacing is provided.

The END-OF-PAGE option applies to a file that has been assigned to a printer. When the END-OF-PAGE punch in the carriage control tape on the printer is detected, the statement following END-OF-PAGE will be executed.

Option 2 must be used for writing on disk files.

If the FROM option is specified, the data is moved from the area specified by identifier-1 to record-name according to the rules specified for the MOVE statement without the CORR or CORRESPONDING option. After execution of the WRITE statement is completed, the information in identifier-1 is available, even though that record-name is not available.

When the WRITE statement is executed at object time, the logical record is released for output and is no longer available for referencing by the object program. Instead, the record area is ready to receive items for the next record to be written. If blocking is called for by the COBOL program, the records will be automatically blocked by the MCP.

Short blocks of records which were written during EOF or EOJ will be of no programmatic concern to the user when using the file as input at a later time.

If a write error is detected during a magnetic tape write operation, the tape record in error will be erased and a rewrite will be attempted further down the tape until the record is finally written correctly. A punch or printer write error will result in a message to the operator. The COBOL programmer need not include any USE procedures to handle write errors.

The shortest allowable record which can be written on 7 and 9 channel magnetic tape units is 7 bytes.

If a CLOSE statement has been executed for a file, any attempt to WRITE on the file until it is OPENed again will result in an error termination.

For files which are being accessed in a SEQUENTIAL manner, the INVALID KEY clause is executed when the end of the last segment of the file (last record) has been reached and another attempt is made to WRITE into the file. The last segment of a file is specified in the FILE-LIMITS clause or the FILE CONTAINS clause. Similarly, for files being accessed in a RANDOM manner, the INVALID

KEY clause will be executed whenever the value of the ACTUAL KEY is outside the defined limits. An INVALID KEY entry must be specified when writing to a file described as containing FILE-LIMITS.

Records will be written onto DISK in either a SEQUENTIAL or RANDOM manner according to the rules given under ACCESS MODE. For RANDOM accessing, SEEK statements may be explicitly used for record determination as defined under ACCESS MODE, SEEK, and READ.

If the size and blocking of records being accessed in a RANDOM manner is such that a WRITE statement must place a record into the middle of a block without disturbing the other contents of the block, then an implicit SEEK will be given to load the block desired (provided that an explicit SEEK has not been given). If the file is being processed for INPUT-OUTPUT, then either an explicit or implicit SEEK for a READ statement will suffice to load the block between the READ and WRITE statements.

If the value of the ACTUAL KEY is changed after a SEEK statement has been given and prior to the WRITE statement, an implied SEEK will be performed and the WRITE will use the record area selected by the implied SEEK as the output record area. The value contained in the ACTUAL KEY will not be affected.

For RANDOM access, when records are unblocked, the use of a SEEK statement related exclusively to WRITE is unnecessary, and may result in an extra loading of the record from disk, because the compiler is, in general, unable to distinguish between SEEK statements that are intended to be related to a READ and those intended to be related to a WRITE.

The TO option allows the selection of stackers for punched card output. The options are:

- a. When TO STACKER is used, identifier-4 must be declared as PC 99 COMPUTATIONAL, or integer-3 must be a positive integral value of 0 through 6.
- b. When TO ERROR or TO AUXILIARY is used, the indicated stacker will be selected.

ZIP

ZIP

The function of this verb is to cause the MCP to execute a control instruction contained within the operating object program.

The construct of this verb is:

ZIP data-name

Data-name (any level) must be assigned a value equivalent to the information contained in the MCP control card. ZIP may be used for programmatic scheduling of subordinate object programs contained in the Systems Program Library or to accomplish any of the MCP control functions as performed through the console printer or card reader.

In the statement ZIP TO-CALL-PGM2, the DATA DIVISION of the source program could contain the following entry:

```
01 TO-CALL-PGM2      PIC X(12), VALUE IS "EXECUTE PGM2".
```

The MCP will be called upon when the object program encounters the ZIP statement and will reference data-name (TO-CALL-PGM2 in the above example) to find out which control function is being called for. Using the above example, the MCP will schedule PGM2. When the time comes and the priority for PGM2 is recognized and memory space becomes available, the MCP will retrieve PGM2 from the program library and place it in the MIX for subsequent operation. The program containing the ZIP verb will proceed to the next sequential instruction following the ZIP.

CODING THE PROCEDURE DIVISION

Figure 7-6 illustrates the manner in which the PROCEDURE DIVISION can be coded.

B 1800/B 1700 COBOL READER-SORTER

GENERAL

A reader-sorter is a device which reads and records specific information on checks, receipts, deposit slips, withdrawal records and similar documents and then sorts the documents. Reading is accomplished by magnetic ink character recognition (MICR) or by optical character recognition (OCR). Sorting is into "pockets" built into the reader-sorter. MICR is the more widely used of the two reading methods in the applications under consideration.

The banking industry, which has a very high daily volume of such documents, is a primary user of reader-sorters.

To deal with this high volume, reader-sorters are high-speed devices, handling 900 to 1625 documents per minute. The maximum rate is determined by the capabilities of the host computer as well as the reader-sorter mechanism and electronics. Operation is in real time; therefore, unlike any other B 1800/B 1700 peripheral device, the reader-sorter must be immediately serviced on request, interrupting all other MCP functions.

This on-demand availability is required by the two functions that must be completed in a very limited amount of time for each document. The first function is the Read; the second function is Pocket Select. Read is self-explanatory. Pocket Select refers to the pocket selection information that must be issued by the program during the time the document moves from the read station to the sort (pocket) area.

Pocket selection is done only in a Pocket-Select DECLARATIVE section of the program. This section is bound to the particular reader-sorter by a "USE" statement (covered later in this section). The purpose of this DECLARATIVE section is to make the decisions, based on data and/or exception conditions, necessary to route the documents into the desired pocket.

It is the programmer's responsibility to insure the quickest path through this Pocket-Select DECLARATIVE section. If the document reaches the entrance to

the pocket area and the pocket select information has not been received from the reader-sorter control by the device, that document and all documents in transit behind it will be routed to the Reject pocket, the feeder will stop, and the program will be informed of the "TOO-LATE-TO-POCKET-SELECT" in the program's non-DECLARATIVE section.

INDEPENDENT FUNCTIONS

In order to understand the interface between the normal-state program (COBOL user program) and the MCP, the independent functions of each must be shown. First, we consider the MCP's function. Once flow mode (documents moving through the reader-sorter) has been initiated by the program, the MCP and the Pocket-Select DECLARATIVE section of the program work independently of the non-DECLARATIVE section of the program.

While in the flow mode, the reader-sorter does not need continuous READ communicates from the program to keep it flowing. The MCP maintains a linked list of alternate areas (buffers) equal to the number declared in the program. It knows where in the list the current check's data is to be put and also knows from where in the list the next check data to be given to the non-DECLARATIVE section, is to be taken. Once this data has been given to the non-DECLARATIVE section, that area is made available to be read into. When the MCP detects that it has fewer than six areas left to read into, it halts the reader-sorter Feeder, allows the checks in the path to be processed by the Pocket-Select DECLARATIVE, and waits 350 milliseconds (timeout period) which is sufficient time for the non-DECLARATIVE section to process all items already tanked by the MCP. After the timeout period, the MCP automatically starts flow after receiving a READ from the program and the process starts over. Therefore, it is important for both the non-DECLARATIVE and the Pocket-Select DECLARATIVE sections to process the data as quickly and efficiently as possible.

Certain other exception conditions also cause the Reader-Sorter flow to stop.

- a. NOT READY: Caused by depressing the Start-Stop bar or a loss or reduction in power.
- b. TOO-LATE-TO-READ: Document reaches the read head before READ is issued by the MCP.
- c. JAM: Document flow is obstructed.
- d. MISSORT: Document did not physically go where the reader-sorter control said it should go.
- e. BLACK-BAND: Immediate Feeder halt, caused by sensing a BLACK-BAND document.

- f. DOUBLE-DOCUMENT: Caused by too long a document or one document overlapping another. (Note: This error is the only exception in this list that will not require a READ communicate to start the Feeder again.)

All the above exception conditions except for BLACK-BAND must use a value greater than 31 for their pocket selection. This is because the reader-sorter control has already routed them to the Reject pocket, if possible, and does not expect valid pocket select information. In the case of the BLACK-BAND exception, it should be given valid pocket information. Because of the independent function of the Pocket-Select DECLARATIVE section of the Cobol program, no communicates other than the Pocket-Select CONTROL communicate (covered later), are allowed.

Those COBOL verbs or conditions which cause the occurrence of communicates (communication between the MCP and the user program) are as follows:

- | | |
|----------------------|----------------|
| a. DATA SEGMENTATION | k. STOP |
| b. READ | l. TRACE |
| c. WRITE | m. DUMP |
| d. CLOSE | n. TODAYS-DATE |
| e. OPEN | o. TODAYS-NAME |
| f. SORT | p. DATE |
| g. SEEK | q. TIME |
| h. ZIP | r. RELEASE |
| i. ACCEPT | s. RETURN |
| j. DISPLAY | t. MERGE |

If issued such communicates will cause a DS or DP situation.

In the non-DECLARATIVE section, the programmer need not be concerned with the operation of the Pocket-Select DECLARATIVE section except in the following case:

When a particular document has not been pocket-selected before it reaches the pocket area, the TOO-LATE-TO-POCKET-SELECT exception condition is set and is seen only by the non-DECLARATIVE section. It is here that the programmer must deal with this exception. Otherwise, the same exception conditions and data which determined the pocket selected will be seen in the non-DECLARATIVE section where the data can be captured. Normally, pocket selection is transparent to the non-DECLARATIVE section.

Since the non-DECLARATIVE section may be interrupted to handle pocket selection between any S-INSTRUCTION (See APPENDIX D), the programmer should avoid long data transfers such as moving of large data areas, READ...INTO, or WRITE...FROM.

I/O operations which are in process at the time of the transfer of control are completed by the MCP in the normal manner and the non-DECLARATIVE section is reinstated at the next S-INSTRUCTION after pocket selection is complete.

Thus you have the independent functions of a Cobol Reader-Sorter program. The following defines the B 1800/B 1700 Cobol language facilities for handling Reader-Sorter files.

COBOL PROVISIONS FOR READER-SORTER HANDLING

Three of the divisions of a COBOL program require special considerations when the program is to process reader-sorter files. Those divisions are ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION.

The following paragraphs define the requirements of those three divisions.

ENVIRONMENT DIVISION REQUIREMENTS

Each reader-sorter to be serviced by the program must have a file assigned to it in the ENVIRONMENT DIVISION. This is accomplished in the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION by use of the SELECT statement.

The construct of this statement is:

```
SELECT file-name ASSIGN TO READER-SORTER;  
    RESERVE integer ALTERNATE AREAS;  
    ACTUAL KEY IS data-name.
```

The file-name must be described in a File Description (FD) entry in the DATA DIVISION.

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

Generally a minimum of nine alternate areas should be specified in order to provide the non-DECLARATIVE section sufficient time to process the documents while the Pocket-Select DECLARATIVE section performs the pocket selection function.

The data-name specified in the ACTUAL KEY clause must be defined in the WORKING-STORAGE SECTION with a length of 112 characters. The ACTUAL KEY data-name specifies the data area where document information is to be placed by the MCP for use by the Pocket-Select DECLARATIVE section in the decision as to which pocket the document is to be routed.

The high-order 24 digits (COMP) of data-name contain result descriptor information for the document which is to be pocket-selected. This result descriptor describes exception conditions found while reading the document. The remaining 100 characters contain the data read from the document, right justified, with blank-fill to the left.

The content of data-name is undefined during the execution of the non-DECLARATIVE section. A more detailed description of the result descriptor portion and the actions required of the program when exception conditions occur appears later in this section of the manual (See Exception Condition Handling).

The I-O CONTROL paragraph uses the APPLY clause to specify the specific reader-sorter read station(s) to be used. At present this clause is for documentation purposes only. When specified, the construct for this clause is:

APPLY { MICR } file-name
 OCR }

DATA DIVISION REQUIREMENTS

The FILE SECTION of the DATA DIVISION must contain a File Description (FD) entry for each reader-sorter file SELECTed.

The construct of the File Description sentence is:

FD file-name.

Other clauses of the FD entry are optional as described in Section 6. It should be noted that blocked reader-sorter files are not allowed.

The file record area for the reader-sorter file must be the same data type and length as the ACTUAL KEY data-name which was specified in the ENVIRONMENT DIVISION. The data which is available in the ACTUAL KEY data-name area during the Pocket-Select DECLARATIVE section is available to the non-DECLARATIVE section in the file record area. (See the TOO-LATE-TO-POCKET-SELECT discussion later in this section of the manual.)

PROCEDURE DIVISION REQUIREMENTS

The PROCEDURE DIVISION of a reader-sorter program must contain both a Pocket-Select DECLARATIVE section and a non-DECLARATIVE section. The following pages define the statements used in the PROCEDURE DIVISION to accommodate reader-sorter files.

CLOSE

The function of this verb is to communicate to the MCP that the designated file-name being operated on has been programmatically completed.

The construct of this verb is:

CLOSE file-name-1

File-name-1 must be described in a File Description (FD) entry. A file must have been OPENed previously before a CLOSE statement can be executed for the file. When a file is programmatically CLOSEd and the assigned unit is released, the memory allocated for that file is returned to the MCP.

The BATCH-COUNT option is used to advance the batch counter contained in the Endorser by one. The batch counter is a three-digit number which is printed by the Endorser as a part of the endorsement.

The POCKET-LIGHT option causes the reader-sorter to light the pocket light of the pocket indicated by literal-2 or identifier-2.

FORMAT

FORMAT

The function of this verb is to place the data received from the Reader-Sorter into 20-character subfields based on the occurrence of specified control characters.

The construct of this verb is:

FORMAT identifier-1 INTO identifier-2
[ON SIZE ERROR statement-1 [ELSE statement-2]]

Both identifier-1 and identifier-2 must be USAGE DISPLAY. Identifier-2 is to be composed of up to nine fields and each field must be 20 characters in length. Data movement by the FORMAT verb is from low to high order. For a more detailed description of the operation of the FORMAT verb, refer to APPENDIX D.

The FORMAT verb may be used in both the Pocket-Select DECLARATIVE and non-DECLARATIVE sections of the PROCEDURE DIVISION.

The ON SIZE ERROR condition will be considered true when:

- a. Any subfield in identifier-1 exceeds 20 characters in length, or
- b. If the total number of characters in identifier-1 exceeds the length of identifier-2, or
- c. The document being FORMATTed is unencoded.

If a CANNOT-READ character is encountered as a subfield is being filled, the most significant character of that subfield will be set to a "J". If no CANNOT-READ character is encountered, the most-significant character is set to a blank.

If the least-significant character is found to be a CANNOT-READ character, it is treated as a transfer character, and is placed into the subfield after a blank has been placed into the least-significant position.

Example 1: (b = blank, * = CANNOT-READ)

Identifier-1

' :654321:@8765-4321@:765-4321:97#0987654321#

Identifier -2

bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb'
bbbbbbbbbbbbbb:654321:
bbbbbbbbbb@8765-4321@
bbbbbbbbbb:765-4321:
bbbbbbbbbbbbbbbbbb97b
bbbbbbbbbb#0987654321#

Example 2: (b = blank, * = CANNOT-READ)

Identifier-1

' 654321:8765-4321@*765-4321*97#98765432**

Identifier-2

bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb'
bbbbbbbbbbbbbb654321:
bbbbbbbbbb8765-4321@
Jbbbbbb*765-4321*97#
Jbbbbbb*98765432**b

MICR-EDIT

MICR-EDIT

The function of this verb is to transfer data from a document subfield to a specified field.

The construct of this verb is:

MICR-EDIT identifier-1 INTO identifier-2

The MICR-EDIT verb will move the contents of identifier-1 to identifier-2. Only the numeric characters 0 thru 9, and the CANNOT-READ character are transferred. A count of the numeric characters moved is placed into the COBOL special register TALLY. Identifier-1 must be defined as alphanumeric, however, identifier-2 may be defined as DISPLAY or COMPUTATIONAL, but must not be specified as having a sign.

Identifier-2 is right-justified and left zero-filled.

The MICR-EDIT verb may appear in both the Pocket-Select DECLARATIVE section, and the non-DECLARATIVE section.

It should be noted that in the Pocket-Select DECLARATIVE section if verbs such as the MICR-EDIT or EXAMINE...TALLYING are used, the programmer should save the contents of TALLY during the execution of the Pocket-Select DECLARATIVE, then restore that value prior to the completion of the Pocket-Select DECLARATIVE. This action is necessary due to the interruptable nature of the non-DECLARATIVE section. In other words, if the program does a MICR-EDIT in the non-DECLARATIVE section, then before it has an opportunity to see the number of characters transferred, the Pocket-Select DECLARATIVE section is executed to perform a pocket-selection, the value contained in TALLY will reflect the results left in TALLY by the Pocket-Select DECLARATIVE section, not the results of the MICR-EDIT which was the last instruction completed before the non-DECLARATIVE section was interrupted.

Several examples of the contents of identifier-1, identifier-2 and TALLY after a MICR-EDIT are shown below:

<u>identifier-1</u>	<u>identifier-2</u>	<u>TALLY</u>
bbbbbbbbb@8765-4321@	00000000000087654321	8
Jbbbbbbb*****-773@	0000000000*****773	3
bbbbbbb#0000075025#	00000000000000075025	10

b = blank, * = CANNOT-READ

OPEN

The function of this verb is to initiate the processing of a reader-sorter file.

The construct of this verb is:

OPEN INPUT file-name-1

File-name-1 must be described in a File Description (FD) entry. The OPEN statement must be executed prior to the first READ statement for that file.

A second OPEN statement for a file cannot be executed prior to the execution of a CLOSE statement for that file.

A file area will not exist in memory until an OPEN statement is executed, which in turn, causes the MCP to allocate memory for the file work area, and alternate areas. The MCP will obtain the needed information from the File Parameter Block to determine the file's characteristics. Once the file has been OPENed, memory will remain allocated until the file is programmatically CLOSED.

READ

READ

The function of this verb is to retrieve document and result descriptor information from documents which have already been pocket selected by the Pocket-Select DECLARATIVE section.

The construct of this verb is:

READ file-name [INTO identifier]

File-name must be the name of a file which is ASSIGNED to READER-SORTER. The READ statement may only appear in the non-DECLARATIVE section of the program.

Result descriptor information is placed into the 24 high-order digit positions of the record area. Because documents are fed through the reader-sorter right-edge first, data from the document is stored in descending memory locations of the remaining record area.

If INTO is specified, data will be moved from the file record to the area specified by identifier according to the rules for the MOVE statement without the CORRESPONDING option. When the INTO option is used, the record being read is available in both the data area associated with identifier and the file record area.

USE

The function of this verb is to specify procedures for handling pocket selection for Reader-Sorter files.

The construct of this verb is:

USE FOR READER-SORTER POCKET ON file-name-1.

The USE statement must immediately follow a section header in the DECLARATIVES portion of the PROCEDURE DIVISION (see the discussion of DECLARATIVES in Section 7) and must be followed by a period followed by a space. The remainder of the section must consist of one or more procedural paragraphs that define the procedures to be used. The USE statement itself is never executed; rather, it defines the conditions which call for the execution of the USE procedures.

File-name-1 must be a file ASSIGNED to READER-SORTER. There must be a USE procedure specified for each Reader-Sorter file.

EXCEPTION CONDITION HANDLING

Information regarding possible exception conditions is given to the user program in a portion of the area set aside for the reader-sorter records. This result descriptor information accompanies each record (document) which is read by the program. In addition to the information regarding exception conditions, a portion of the result descriptor area is utilized to inform the non-DECLARATIVE section which pocket number was selected for that document by the Pocket-Select DECLARATIVE section. This result descriptor portion is contained in the high-order 12 bytes of each record. The format of the result descriptor area is as follows:

05 RESULT-DESCRIPTOR.

10	OPERATION-COMPLETE	PIC	9	COMP.
10	EXCEPTION-CONDITION	PIC	9	COMP.
10	NOT-READY	PIC	9	COMP.
10	UNENCODED-DOCUMENT	PIC	9	COMP.
10	DIGIT-5-RESERVED	PIC	9	COMP.
10	CANNOT-READ	PIC	9	COMP.
10	DIGITS-7-9-RESERVED	PIC	999	COMP.
10	DOUBLE-DOCUMENT	PIC	9	COMP.
10	TOO-LATE-TO-READ	PIC	9	COMP.
10	JAM	PIC	9	COMP.
10	MISSORT	PIC	9	COMP.
10	BLACK-BAND	PIC	9	COMP.
10	HALT-FLOW-STOPPED	PIC	9	COMP.
10	DIGIT-16-RESERVED	PIC	9	COMP.
10	TOO-LATE-TO-POCKET-SELECT	PIC	9	COMP.
10	DIGITS-18-20-RESERVED	PIC	999	COMP.
10	POCKET-NUMBER-SELECTED	PIC	99	DISPLAY.

These exception conditions and the program actions they may require are described in the paragraphs which follow. For conditions in the result descriptor, a value of one (1) represents a condition that is TRUE, and a value of zero (0) represents a condition that is FALSE.

OPERATION-COMPLETE - If this condition is TRUE, the remainder of the result descriptor contains valid results.

EXCEPTION-CONDITION - When this condition is TRUE, one or more of the following conditions must also be TRUE: NOT-READY, UNENCODED-DOCUMENT, CANNOT-READ, DOUBLE-DOCUMENT, TOO-LATE-TO-READ, JAM, MISSORT, BLACK-BAND, HALT-FLOW-STOPPED, TOO-LATE-TO-POCKET-SELECT.

NOT-READY - This condition is caused by one or more of the following:

- a. Power off or not yet fully up.
- b. Off line.
- c. Jam.
- d. Missort.
- e. Empty hopper.
- f. Full pocket.
- g. Start/Stop bar depressed while Ready.
- h. Pocket lighted.
- i. Feed check (no item fed within reasonable time while Feeder is on).
- j. Transport shutdown (after Feeder off for a number of minutes).
- k. Open interlock.

When a NOT-READY condition occurs, the reader-sorter transport continues to operate and documents already in motion will be read, pocket selected, and pocketed normally; however, the Feeder is turned off and no additional documents are fed. In the case of an open interlock or power down, the entire transport mechanism is immediately stopped, and several documents may have to be manually removed from the transport area. Since the reader-sorter control and the MCP cannot in general distinguish between these various cases, operation proceeds as though all documents in motion are to be processed in the normal manner. If documents in fact stop in the transport area, the reader-sorter operator must remove these and assist in the recovery procedure.

The data which is returned with a NOT-READY condition is INVALID, and must not be processed. The Pocket-Select DECLARATIVE section must select a pocket number which is greater than 31 when a NOT-READY condition is reported in the result descriptor.

UNENCODED-DOCUMENT - This condition is TRUE when the document related to this result descriptor contains no non-blank characters. A document with one or more characters (even a cannot-read character) is not reported as unencoded. The Pocket-Select DECLARATIVE section must select a pocket number less than 32, unless there is also an exception condition reported which requires a pocket number greater than 31, in which case that requirement takes precedence.

DIGIT-5-RESERVED - This digit is reserved.

CANNOT-READ - A CANNOT-READ condition indicates that one or more of the characters detected by the reader-sorter could not be correctly interpreted. Note that UNENCODED-DOCUMENT and CANNOT-READ are mutually exclusive.

A CANNOT-READ condition does not invalidate the entire data read from the document. Those characters which could not be correctly interpreted are so indicated in the data, and the remaining valid characters may be used. If the data required by the program is valid and no other exceptions cause the data to be incorrect, the program may ignore the CANNOT-READ condition.

The Pocket-Select DECLARATIVE section must select a pocket number less than 32, unless there is also an exception condition reported which requires a pocket number greater than 31, in which case, that requirement takes precedence.

DIGITS-7-9-RESERVED - These digits are reserved.

DOUBLE-DOCUMENT - This condition is caused by double-fed, under-spaced, over-length, or underlength documents occurring between the Feeder and the read station, as well as document slippage between the read head and the Reject pocket. When a DOUBLE-DOCUMENT occurs, the reader-sorter transport continues to operate and documents prior to the document(s) which caused the exception are processed normally. However, the Feeder is turned off, and the document(s) which caused the exception and succeeding documents which were already in motion are routed to the Reject pocket.

The data which is returned with a DOUBLE-DOCUMENT condition is INVALID and must not be processed. The Pocket-Select DECLARATIVE section must select a pocket number greater than 31 when a DOUBLE-DOCUMENT condition is reported in the result descriptor.

TOO-LATE-TO-READ - If when a document reaches the read station, the buffer in the control is still involved with the previous read operation, a TOO-LATE-TO-READ condition has occurred. The control immediately signals the Feeder to halt, and stops reading documents. The data which is returned with a TOO-LATE-TO-READ condition is INVALID and must not be processed. The document on which this condition occurs and subsequent documents which are already in motion, are routed to the Reject pocket by the control. The Pocket-Select DECLARATIVE section must select a pocket number greater than 31 when a TOO-LATE-TO-READ condition is reported in the result descriptor.

JAM - When a JAM is detected, the reader-sorter immediately goes not ready and stops the Feeder. If the control was waiting for pocket select information when the jam occurred, the pocket select information is accepted and sent to the reader-sorter. That document may or may not be properly pocketed depending on the location of the jam.

If the jam is located before the read heads, documents which have been pocket selected are correctly pocketed. If the jam is located in the transport or

pocket areas, the transport is halted immediately. In this case the operator may be required to extract documents from the transport area and manually pocket the items. It should be noted that there is no indication, except to the operator, as to where the jam occurred and where various documents have stopped. Also, a jam which occurs after the read result has been returned is not reported to the program, even though documents may be halted in the transport area.

The data which is returned with the JAM condition is INVALID and must not be processed. The Pocket-Select DECLARATIVE section must select a pocket number greater than 31 when a JAM condition is reported in the result descriptor.

MISSORT - When a missort is detected, the reader-sorter immediately goes not ready and stops the Feeder. The reader-sorter lights the pocket(s) into which the missorted document(s) was routed. If the control was waiting for pocket select information when the missort occurred, the pocket select information is accepted and sent to the reader-sorter, and that document is pocketed normally. A missort which occurs after the read result for the last item has been returned is not reported to the program.

If a document is directed to a nonexistent pocket on a reader-sorter with fewer than 32 pockets, then that document is a missort to the last pocket. MISSORT is then reported on a subsequent document.

The data which is returned with the MISSORT condition is INVALID and must not be processed. The document(s) missorted and succeeding documents which are already in motion are routed to the Reject pocket. The Pocket-Select DECLARATIVE section must select a pocket number greater than 31 when a MISSORT condition is reported in the result descriptor.

BLACK-BAND - A BLACK-BAND condition is caused by a large black area on a document which is detected photo-optically by the reader-sorter. Detection occurs close to the Feeder so that the Feeder can be stopped without feeding the following document.

A document which contains this large black area may be used to stop the feeding of documents, and to flag the end of a specific series of documents.

The data which is returned with a BLACK-BAND condition is VALID, and may be processed. The Pocket-Select DECLARATIVE section must select a pocket number less than 32, unless there is also an exception condition reported which requires a pocket number greater than 31, in which case, that requirement takes precedence. Other exceptions or conditions are reported as in a normal

read operation. It should be noted that no operation will be executed by the reader-sorter until a 350-ms timeout period has elapsed since the Feeder was turned off.

HALT-FLOW-STOPPED - This condition is used to inform the user program that there has been a programmatic halting of the Feeder, and that the document with which the HALT-FLOW-STOPPED condition is reported is the last document which was fed before the Feeder was stopped. This programmatic stopping of the Feeder may be done either by the user program executing the STOP-FLOW option of the CONTROL statement, or by the MCP.

The data which is returned with a HALT-FLOW-STOPPED condition is VALID, and may be processed. The Pocket-Select DECLARATIVE section must select a pocket number less than 32, unless there is also an exception condition reported which requires a pocket number greater than 31, in which case, that requirement takes precedence. Other exceptions or conditions are reported as in a normal read operation. It should be noted that no operation will be executed by the reader-sorter until a 350-ms timeout period has elapsed since the Feeder was turned off.

DIGIT-16-RESERVED - This digit is reserved.

TOO-LATE-TO-POCKET-SELECT - This condition is caused when the Pocket-Select DECLARATIVE section does not issue pocket select information before the document to be pocketed reaches the entrance to the pocket area. This condition is only reported in the result descriptor in the non-DECLARATIVE section.

When the TOO-LATE-TO-POCKET-SELECT condition occurs, the control turns off the Feeder and stops reading. The document which caused the condition and succeeding documents which were already in motion at the time the Feeder stopped are routed to the Reject pocket during the 350-ms timeout period which started when the last item was fed. Documents preceding the TOO-LATE-TO-POCKET-SELECT document are pocketed normally. Normal operation is resumed when an operation is received for the reader-sorter after the timeout period.

The data which is returned with a TOO-LATE-TO-POCKET-SELECT condition is valid and may be processed. However, it must be noted, that the document has been routed to the Reject pocket by the control regardless of what is contained in the POCKET-NUMBER-SELECTED field.

DIGITS-18-20-RESERVED - These digits are reserved.

POCKET-NUMBER-SELECTED - This field is available only in the non-DECLARATIVE section and contains the pocket number which was selected for this document by the Pocket-Select DECLARATIVE section. The pocket number contained in the **POCKET-NUMBER-SELECTED** field is not valid if the **TOO-LATE-TO-POCKET-SELECT** condition is true.

READER-SORTER CHARACTERS

The two most widely used character recognition systems on the reader-sorter are the MICR E13B font, and the OCR-A font.

The following table and figure present these two character sets.

READER-SORTER CHARACTER TRANSLATION

<u>E13B FONT</u>	<u>OCR-A FONT</u>	<u>INTERNAL EBCDIC</u>	<u>EBCDIC GRAPHIC</u>
0	0	F0	0
1	1	F1	1
2	2	F2	2
3	3	F3	3
4	4	F4	4
5	5	F5	5
6	6	F6	6
7	7	F7	7
8	8	F8	8
9	9	F9	9
AMOUNT	NOT USED	7B	#
TRANSIT	NOT USED	7C	@
ON-US	NOT USED	79	:
HYPHEN	NOT USED	60	-
CANNOT-READ	NOT USED	5C	*
NOT USED	HOOK	4C	<
NOT USED	FORK	7E	=
NOT USED	CHAIR	6E	>
NOT USED	VERTICAL BAR	4F	
NOT USED	BLANK	40	BLANK
NOT USED	CANNOT-READ	3F	SUB
The END-OF-DOCUMENT character		7D	'

is inserted by the reader-sorter control.

E13 B TYPE FONT MICR

OCR-A FONT

0 1 2 3 4

1 2 3 4

5 6 7 8 9

5 6 7

•• AMOUNT SYMBOL

8 9 0

••• ON US SYMBOL

∟

••• TRANSIT NUMBER SYMBOL

∟ VERTICAL BAR
∟ CHAIR
∟ FORK
∟ HOOK

••• DASH SYMBOL

Figure 8-1. Approved Character Fonts

READER-SORTER PROGRAMMING CONSIDERATIONS

- a. If the special register TALLY is used in both the Pocket-Select DECLARATIVE and non-DECLARATIVE sections, the value contained in TALLY at the time the Pocket-Select DECLARATIVE is entered must be stored by the programmer in a temporary area during the execution of the Pocket-Select DECLARATIVE. This value must be restored to TALLY just prior to executing the CONTROL verb.
- b. If FORMAT or MICR-EDIT statements appear in both the Pocket-Select DECLARATIVE and non-DECLARATIVE sections, the data areas used must not be the same in both sections.
- c. If the Pocket-Select DECLARATIVE section selects the pocket value @FF@, the value placed into the POCKET-NUMBER-SELECTED field in the non-DECLARATIVE will be a value greater than 31, but not @FFFF@.

EXAMPLE PROGRAM

The following program listing is that of a program which will read, pocket select, and print MICR documents. The program prints both result descriptor and document data.

005510 /				* MICR *	000,0000415
005600	05	RESULT-DES REDEFINES RESULT-DESCRIPTOR PC 9(20) CMP.		* MICR *	000,0000398
005700	05	PNT-NUMBER	PC 99.	* MICR *	[0005] 000,0000418
005800	05	DOCUMENT.		* MICR *	000,0000422
005900	07	FILLER	PC X(10).	* MICR *	000,0000422
006000	07	PRNT-ITM.		* MICR *	000,0000442
006100	09	MSG-DATA	PC X(20).	* MICR *	[0006] 000,0000442
006200	09	FILLER	PC X(70).	* MICR *	000,0000482
006400		WORKING-STORAGE SECTION.		* MICR *	
006500	77	PCK-NUM	PC 99 CMP VA 0.	* MICR *	[0007] 000,0000622
006600	77	TOT	PC 9(6) CMP VA 0.	* MICR *	[0008] 000,0000624
006700	77	TOT-EXC	PC 9(6) CMP VA 0.	* MICR *	[0009] 000,0000630
006800	77	E-PERCENT	PC 99999999 CMP VA 0.	* MICR *	[0010] 000,0000636
006900	77	STORE-TALLY	PC 9(5) CMP VA 0.	* MICR *	[0011] 000,0000642
007000	77	PCK-COUNT	PC 999 CMP VA 0.	* MICR *	[0012] 000,0000647
007100	88	PCKET-NOI-FULL VA 000 THRU 300.		* MICR *	
007200	77	LINE-CNT	PC 99 CMP VA 0.	* MICR *	[0013] 000,0000650
007300	77	EOJFLAG	PC 9 CMP VA 0.	* MICR *	[0014] 000,0000652
007400	77	FLOW-STP-FLAG	PC 9 CMP VA 0.	* MICR *	[0015] 000,0000653
007500	77	USE-TC-WKA	PC 9(20) CMP VA 0.	* MICR *	[0016] 000,0000654
007600	77	TC-WK-A	PC 9(20) CMP VA 0.	* MICR *	[0017] 000,0000674
007700	01	PCK-SEL-BUF.		* MICR *	000,0000694
007800	05	PCK-DESC.		* MICR *	000,0000694
007900	10	PCK-OPCOMP	PC 9 CMP.	* MICR *	000,0000694
008000	10	PCK-EX-COND	PC 9 CMP.	* MICR *	000,0000695
008100	10	PCK-NRDY	PC 9 CMP.	* MICR *	000,0000696
008200	10	FILLER	PC 9(6) CMP.	* MICR *	000,0000697
008300	10	PCK-DD	PC 9 CMP.	* MICR *	000,0000703
008400	10	PCK-TLT-RD	PC 9 CMP.	* MICR *	000,0000704
008500	10	PCK-JAM	PC 9 CMP.	* MICR *	000,0000705
008600	10	PCK-SORT	PC 9 CMP.	* MICR *	000,0000706
008700	10	FILLER	PC 9(11) CMP.	* MICR *	000,0000707
008800	05	PCK-DOCUMENT	PC X(100).	* MICR *	000,0000718
008900	01	USE-ITEM.		* MICR *	000,0000918
009000	05	FILLER	PC X(60).	* MICR *	000,0000918
009100	05	USE-FIELD8	PC X(20).	* MICR *	000,0001038
009200	05	FILLER	PC X(20).	* MICR *	000,0001078
009300	01	ITEM-PRGC.		* MICR *	000,0001118
009400	05	FILLER	PC X(10).	* MICR *	000,0001118
009500	05	ALT-ITMWR.		* MICR *	000,0001138
009600	07	DATA-MSG	PC X(20).	* MICR *	[0018] 000,0001138
009700	07	FILLER	PC X(30).	* MICR *	000,0001178
009800	07	FIELD8	PC X(20).	* MICR *	000,0001238
009900	07	FILLER	PC X(20).	* MICR *	

012510 /		* MICR *	000,0002460
012600	PROCEDURE DIVISION.	* MICR *	
012700	DECLARATIVES.	* MICR *	000,0000000
012800	POCKET-SELECT-DECLARATIVE SECTION.	* MICR *	000,0000000
012900	USE FOR READER-SORTER POCKET MICRIN.	* MICR *	000,0000000
013000	DESC-SEARCH.	* MICR *	000,0000000
013100	MOVE SW2 TO PCK-NUM. MOVE TALLY TO STORE-TALLY.	* MICR *	000,0000000
013200	IF PCK-EX-COND = 0 GO TO PROC-ITM.	* MICR *	000,0000076
013300	IF 1 = PCK-ARDY OR PCK-DD OR PCK-TLT-RD OR PCK-JAM	* MICR *	000,0000132
013400	OR PCK-SORT, MOVE SW2 TO PCK-NUM, GO TO PCK-SEL.	* MICR *	000,0000312
013500	PROC-ITM.	* MICR *	000,0000472
013600	ADD 1 TO PCK-COUNT.	* MICR *	000,0000472
013700	FORMAT PCK-DOCUMENT INTO USE-ITEM ON SIZE ERROR,	* MICR *	000,0000486
013800	GO TO PCK-SEL.	* MICR *	000,0000588
013900	MICR-EDIT USE-FIELD8 INTO USE-TC-MKA.	* MICR *	000,0000606
014000	IF (TALLY = 3 AND USE-TC-MKA = 96) MOVE 1 TO FLOW-STP-FLAG.	* MICR *	000,0000678
014100	MOVE STORE-TALLY TO TALLY.	* MICR *	000,0000796
014200	IF POCKET-NCT-FULL GO TO PCK-SEL.	* MICR *	000,0000834
014300	STP-FLOW.	* MICR *	000,0000928
014400	CONTROL MICRIN STOP-FLOW POCKET PCK-NUM.	* MICR *	000,0000928
014500 *	NOTE THAT EXECUTION OF THE "CONTROL" STATEMENT WILL CAUSE	* MICR *	000,0001134
014600 *	THE DECLARATIVE SECTION TO BE EXITED.	* MICR *	000,0001134
014700	PCK-SEL.	* MICR *	000,0001134
014800	IF FLOW-STP-FLAG = 1 GO TO STP-FLOW.	* MICR *	000,0001134
014900	CONTROL MICRIN POCKET PCK-NUM.	* MICR *	000,0001170
015000	END DECLARATIVES.	* MICR *	000,0001376

015005 /		* MICR *	000,0001385
015010	NON-DECLARATIVE SECTION.	* MICR *	000,0001385
015100	STARTER.	* MICR *	000,0001385
015200	OPEN INPUT MICRIN OUTPUT PRT-MICR. MOVE SPACES TO PRTMICR.	* MICR *	000,0001385
015300	MOVE TODAYS-DATE TO HDG-DATE. MOVE TODAYS-NAME TO DAY-NAME.	* MICR *	000,0001567
015400	MOVE TIME TO RUN-TIME.	* MICR *	000,0001925
015500	HCG-RTN.	* MICR *	000,0002113
015600	MOVE 0 TO LINE-CNT. ADD 1 TO PG-NO.	* MICR *	000,0002113
015700	WRITE PRTMICR BEFORE ADVANCING TO CHANNEL 1.	* MICR *	000,0002165
015800	WRITE PRTMICR FROM HDG-ONE BEFORE ADVANCING 1 LINES.	* MICR *	000,0002321
015900	WRITE PRTMICR FROM HDG-TWO.	* MICR *	000,0002515
016000	WRITE PRTMICR FROM HDG-THREE.	* MICR *	000,0002709
016100	WRITE PRTMICR FROM HDG-FOUR BEFORE ADVANCING 2 LINES.	* MICR *	000,0002903
016200	MOVE SPACES TO PRTMICR.	* MICR *	000,0003097
016400	READ-ITEM.	* MICR *	000,0003111
016500	IF EOJFLAG = 1 GO TO TST-FLOW-STOPPED.	* MICR *	000,0003111
016600	READ MICRIN.	* MICR *	000,0003147
016700	IF EXCEPTION-CONDITION ADD 1 TO TOT-EXC ELSE GO TO PRT-RTN.	* MICR *	000,0003303
016800	IF TOO-LATE-TO-POCKET-SELECT MOVE "TLP" TO PRT-INV ELSE	* MICR *	000,0003413
016900	IF DOUBLE-DOCUMENT MOVE "DOUBLE-DOC" TO PRT-INV	* MICR *	000,0003577
017000	MOVE "*** INVALID DATA ***" TO DATA-MSG MSG-DATA ELSE	* MICR *	000,0003637
017100	IF JAM MOVE "JAM" TO PRT-INV	* MICR *	000,0004121
017200	MOVE "*** INVALID DATA ***" TO DATA-MSG MSG-DATA ELSE	* MICR *	000,0004181
017300	IF MISSORT MOVE "MISSORT" TO PRT-INV	* MICR *	000,0004665
017400	MOVE "*** INVALID DATA ***" TO DATA-MSG MSG-DATA ELSE	* MICR *	000,0004725
017500	IF TOO-LATE-TO-READ MOVE "ILTR" TO PRT-INV	* MICR *	000,0005209
017600	MOVE "*** INVALID DATA ***" TO DATA-MSG MSG-DATA ELSE	* MICR *	000,0005269
017700	IF NOT-READY MOVE "NOT READY" TO PRT-INV	* MICR *	000,0005753
017800	MOVE "*** INVALID DATA ***" TO DATA-MSG MSG-DATA ELSE	* MICR *	000,0005813
017900	IF UNENCODED-DOCUMENT MOVE "UNENCODED" TO PRT-INV ELSE	* MICR *	000,0006297
018000	IF BATCH-TICKET MOVE "BLACK BAND" TO PRT-INV ELSE	* MICR *	000,0006477
018100	IF CANT-READ-CHAR MOVE "CANT READ" TO PRT-INV ELSE	* MICR *	000,0006657
018200	IF HALT-FLOW-STOPPED MOVE "HALT VAR" TO PRT-INV	* MICR *	000,0006837
018300	PRT-RTN.	* MICR *	000,0006999
018400	FORMAT DOCUMENT INTO ITEM-PRCC ON SIZE ERROR	* MICR *	000,0006999
018500	MOVE "SIZE ERROR" TO DATA-MSG. GO TO SET-UP-PRT.	* MICR *	000,0007101
018600	MICR-EDIT FIELDS INTO TC-WK-A.	* MICR *	000,0007221
018700	IF (TALLY = 3 AND TC-WK-A = 96) MOVE 1 TO EOJFLAG.	* MICR *	000,0007293
018800	GO TO TST-PCK.	* MICR *	000,0007411
018900	TST-FLOW-STOPPED.	* MICR *	000,0007429
019000	IF BATCH-TICKET OR HALT-FLOW-STOPPED GO TO EOJ-RTN.	* MICR *	000,0007429
019100	GO TO READ-ITEM.	* MICR *	000,0007567
019200	TST-PCK.	* MICR *	000,0007585
019300	IF POCKET-NOT-FULL GO TO SET-UP-PRT.	* MICR *	000,0007585
019400	IF HALT-FLOW-STOPPED MOVE "POCKET" TO PRT-INV.	* MICR *	000,0007679
019500	MOVE 0 TO PCK-COUNT.	* MICR *	000,0007739
019600	CONTROL MICRIN POCKET-LIGHT PKT-NUMBER.	* MICR *	000,0007825

```

019610 /
019700 SET-UP-PRT.
019800 IF (SW1 = 1 OR 2) MOVE PRNT-ITM TO PRT-ITEM
019900 PERFORM ACTUAL-PRINT.
020000 IF (SW1 = 0 OR 1) MOVE ALT-ITMKN TO PRT-ITEM
020100 PERFORM ACTUAL-PRINT.
020200 ADD 1 TO TOT.
020300 IF LINE-CNT > 59 GO TO HDG-RTN.
020400 GO TO READ-ITEM.
020500 ACTUAL-PRINT.
020600 MOVE RESULT-DES TO PRT-RDESC.
020700 IF EXCEPTION-CONDITION
020800 EXAMINE PRT-XRDESC REPLACING ALL "0" BY "--"
020900 ELSE
021000 EXAMINE PRT-XRDESC REPLACING ALL "0" BY " ".
021100 MOVE PKT-NUMBER TO PRT-POCKNUM.
021200 WRITE PRTMICR BEFORE ADVANCING 1.
021300 MOVE SPACES TO PRTMICR.
021400 ADD 1 TO LINE-CNT.
021500 EQJ-RTN.
021600 MOVE TOT TO DOC-TOT. MOVE TOT-EXC TO EXC-TOT.
021700 DIVIDE TOT INTO TOT-EXC GIVING E-PERCENT ROUNDED.
021800 MOVE E-PERCENT TO PCEAT-EXC.
021900 WRITE PRTMICR FROM TOT-LINE AFTER ADVANCING 2 LINES.
022000 CLOSE MICRIA RELEASE PRT-MICR.
022100 STOP RUN.
022200 END-OF-JOB.

```

```

* MICR * 000.0007949
* MICR * 000.0007949
* MICR * 000.0007949
* MICR * 000.0008069
* MICR * 000.0008133
* MICR * 000.0008249
* MICR * 000.0008313
* MICR * 000.0008327
* MICR * 000.0008367
* MICR * 000.0008385
* MICR * 000.0008385
* MICR * 000.0008447
* MICR * 000.0008447
* MICR * 000.0008507
* MICR * 000.0008572
* MICR * 000.0008619
* MICR * 000.0008657
* MICR * 000.0008813
* MICR * 000.0008827
* MICR * 000.0008841
* MICR * 000.0008858
* MICR * 000.0008970
* MICR * 000.0009253
* MICR * 000.0009309
* MICR * 000.0009611
* MICR * 000.0009779
* MICR * 000.0009819

```


C O D E D I C T I O N A R Y
 BYTE LENGTH CODEFILE RELATIVE DISK ADR

 000 C0C01228 C0C013
 C0C01228 TOTAL CODE

C A T A D I C T I O N A R Y
 BYTE LENGTH CODEFILE RELATIVE DISK ADR

 000 C0C01516 C0C0C3

S - P A C H I N E P A R A M E T E R S (S C R A T C H P A D)

LEA8=8, SEGB=0, DISPB=12, COPXB=5, COPB=24, D.E.F=354, BDISPB=14
 BASE RELATIVE ADDRESSES:
 DATA-SEGMENT=0=640, COF-TAEL=40, STACK=11125 (BIT LENGTH=1000)

P R O G R A M P A R A M E T E R B L O C K

FIRST-EXECUTABLE-INSTRUCTION=0,1385
 INTERPRETER-NAME=COBOL /PICR
 STATIC-CORE=12128 BITS
 CYAAPIC-CORE=0 BITS
 DATA DICTIONARY STARTS AT CODEFILE SEGMENT 2, 1 ENTRY
 CODE DICTIONARY STARTS AT CODEFILE SEGMENT 12, 1 ENTRY
 FILE PARAMETER BLOCKS START AT CODEFILE SEGMENT 20, 2 ENTRIES

***** COMPILATION COMPLETE

ELAPSED TIME : 02 MINUTES, 57 SECONDS
 PROGRAM REQUIRES 22 DISK SEGMENTS OF 180 BYTES EACH,
 MEMORY REQUIREMENTS

00C1228 BYTES = LARGEST CODE SEGMENT
 C0C1516 BYTES = BASE-TC-LIMIT AREA
 C0C0315 BYTES = DICTIONARIES AND RUN STRUCTURE
 C001823 BYTES = FILE BUFFERS & FILE INFO AREAS - INCLUDES 129 BYTES (+72 TO 540 IF DISK) FOR EACH FILE
 C0C4882 BYTES = ESTIMATED MEMORY REQUIRED TO RUN IF ALL FILES OPEN
 225 SYMBOLIC RECORDS COMPILED AT 76.260 RECORDS PER MINUTE.

DATA COMMUNICATIONS

GENERAL

This section deals with the COBOL constructs of the PROCEDURE DIVISION required to activate the data communications equipment as defined by the ASSIGN to hardware-name clause.

SPECIFIC VERB FORMATS

The following differences exist in OPEN, CLOSE, READ, WRITE, and USE when I/O is to be performed on remote files.

When the OPEN statement is executed, the Network Controller begins to poll all stations attached to the file that has just been opened if:

- a. The file is not currently open.
- b. No Message Control System is present.

When the CLOSE statement is executed, the Network Controller stops polling all the stations on the file if:

- a. The file has been opened only by the program that is attempting to close it.
- b. No MCS is present.

The READ statements will wait for a message and suspend program execution if no messages are queued for that program. If an ACTUAL KEY is specified, the READ statement sets the STATION-RSN, TEXT-LENGTH, and MSG-TYPE fields of ACTUAL KEY.

AT END will be executed when the datacom network controller receives a QC message, or when the MCS (if one exists) does an MCS.COMMUNICATE with MESSAGE.TYPE set to 1 and MESSAGE.VARIANT set to 3.

WRITE requires MESSAGE-TYPE to be set to 0 if an actual key is used, as well as TEXT-LENGTH set to the actual message length, and STATION-RSN to be set to the correct relative station number of the terminal to which the message is to be sent.

USE AFTER STANDARD ERROR PROCEDURE ON file-name must be specified to avoid a DS or DP condition when library request sets in NDL are used. (This is not a problem but is a standard procedure to follow.) The execution of TERMINATE ERROR by the Network Controller will always invoke the USE procedure in the associated application program.

INTER-PROGRAM COMMUNICATION

GENERAL

This section describes the COBOL inter-program communication, which is achieved by means of "message queues" in memory. (A queue is treated by the MCP and compilers almost exactly as if it were a normal file.) A discussion on the implementation of queues follows.

MESSAGE QUEUES

The term "queue" refers to the actual data structure maintained by the MCP as a means of inter-program communication. Queues have various attributes including a 20-character name, user count, and message count. Most important, a queue may contain a list of messages (possibly empty). A queue user may add to the back or remove from the front of this list. The queue may be shared -- one or more programs may put messages in the list and one or more programs may remove messages. Only the MCP may access a queue directly. User programs must use other interfaces built upon queues, such as queue files or remote files.

The following general information applies to queues:

- a. The maximum number of messages allowed in a queue at any one time is specified by the program first opening the file.
- b. A means of reporting to the user program that the queue was empty during a READ, or that the queue was full and the WRITE did not complete, has been specified for queue handling.
- c. Queues which contain a large number of messages may be optionally maintained on disk. Disk I/O to place or retrieve messages to or from disk is overlapped with user processing whenever possible.
- d. The end of file reporting on READ statements has been defined for queue files in such a way as to permit label equation to other devices without causing incorrect reporting of end of file.
- e. Queue files may not be blocked. Records written to a queue file are immediately put into the queue.

- f. The use of an actual key of zero on a READ of a queue file family is defined as an UNSPECIFIC READ. The members of the queue family will be examined beginning with queue number one, and the first queue member found not empty will be read.

Messages stored in a queue may reside on disk or in memory. At queue creation time, an area of system disk is obtained for the queue large enough to hold the maximum number of messages of the maximum message length allowed in the queue. Users with small amounts of available system disk may have to limit the maximum number of messages allowed to avoid disk space problems. Queue disk is not locked in the directory; if the system should fail while queues are active, the disk is returned to the available list during CLEAR/START.

In addition to parameters common to all files, the user may specify three parameters whose interpretation has special meaning for queue files.

- a. Q-MAX-MESSAGES - the maximum number of messages a queue can store, at which point it is full (maximum 255).
- b. Buffers - the highest number of messages (maximum 255) the user wishes to allow in memory (alternate areas).
- c. Family Size - the number of subqueues in a multiple queue file (FILE CONTAINS NN QUEUES).

A COBOL example of the specifications of these parameters follows:

```
SELECT MY-Q ASSIGN TO QUEUE,  
RESERVE 3 ALTERNATE AREAS.  
  
SELECT MY-QFF ASSIGN TO QUEUE,  
ACTUAL KEY IS WHICH-QUEUE,  
RESERVE 2 ALTERNATE AREAS.
```

```
FD MY-Q VALUE OF Q-MAX-MESSAGES IS 20.
```

```
01 MY-Q-BUF PIC X(80).
```

```
FD MY-QFF FILE CONTAINS 3 QUEUES  
VALUE OF Q-MAX-MESSAGES IS 10.
```

```
01 MY-QFF-BUF PIC X(80).
```

```
77 WHICH-QUEUE, PC 9(8), CMP.
```

When a queue file family is opened, the same parameters apply to every member individually. In MY-QFF, above, for example, all 3 members may hold 10 messages each with 2 each in memory. Queue file names for single queue files are created the same as disk files on system disk; they may contain up to two 10-character names. For a queue file family however, two 10-character names

are required. The first name is the main directory or family name. The second name is created from the member number for each queue in the family. For example, the name of the first member of MY-QFF would be "MY-QFF"/"###00000001".

When a queue file is opened, the name of the queue is compared with the names of all queues currently in existence. If a queue of that name is found, the opening program is linked to the existing queue and that queue's user count is incremented to reflect the additional user. If the requested queue does not exist, a new queue is created with the parameters provided in the file parameter block (FPB).

The parameters regarding queues are set when the queue is first created; that is, the parameters defined by the first program to open the queue apply to all other programs which open that queue.

Some recommendations for users of queue files follow:

- a. Declare short queues (Q-MAX-MESSAGES = 1) unless the extra length is needed for "BURST MODE" operation. Long queues tie up resources and incur longer delays for each message, while providing no additional functions. Disk queues (those where Q-MAX-MESSAGES is greater than buffers) are discouraged for the same reasons. Note, however, that if the queue is kept empty, the cost of a disk queue is limited to the systems disk required to hold the queue. Therefore, if you must have the potential for occasional long queues, try to ensure that the READ program has a high enough priority to keep the queue emptied. It will then run as fast as, and in the same real memory as, a memory queue.
- b. Use the USE procedures for Q-FULL/Q-EMPTY only if there are serious time difficulties with another data stream (for example, driving data communications I/O or a reader-sorter). A program may just as well be suspended on queue empty or full and release the processor to perform other tasks. Care should be used to avoid a "deadlocked" situation in these cases.
- c. Use the end of file branch on READ -- it notifies the reading program of the termination of the writing program. It could serve as a termination condition, possibly for an abnormal end, for the reading program.

COBOL COMPILER CONTROL

GENERAL

The COBOL compiler, in conjunction with the Master Control Program, allows for various types of actions during compilation and is explained in the text that follows.

COMPILATION CARD DECK

Control of the COBOL source-language input is derived from presenting the compilation card deck, illustrated in figure 11-1, to the MCP.

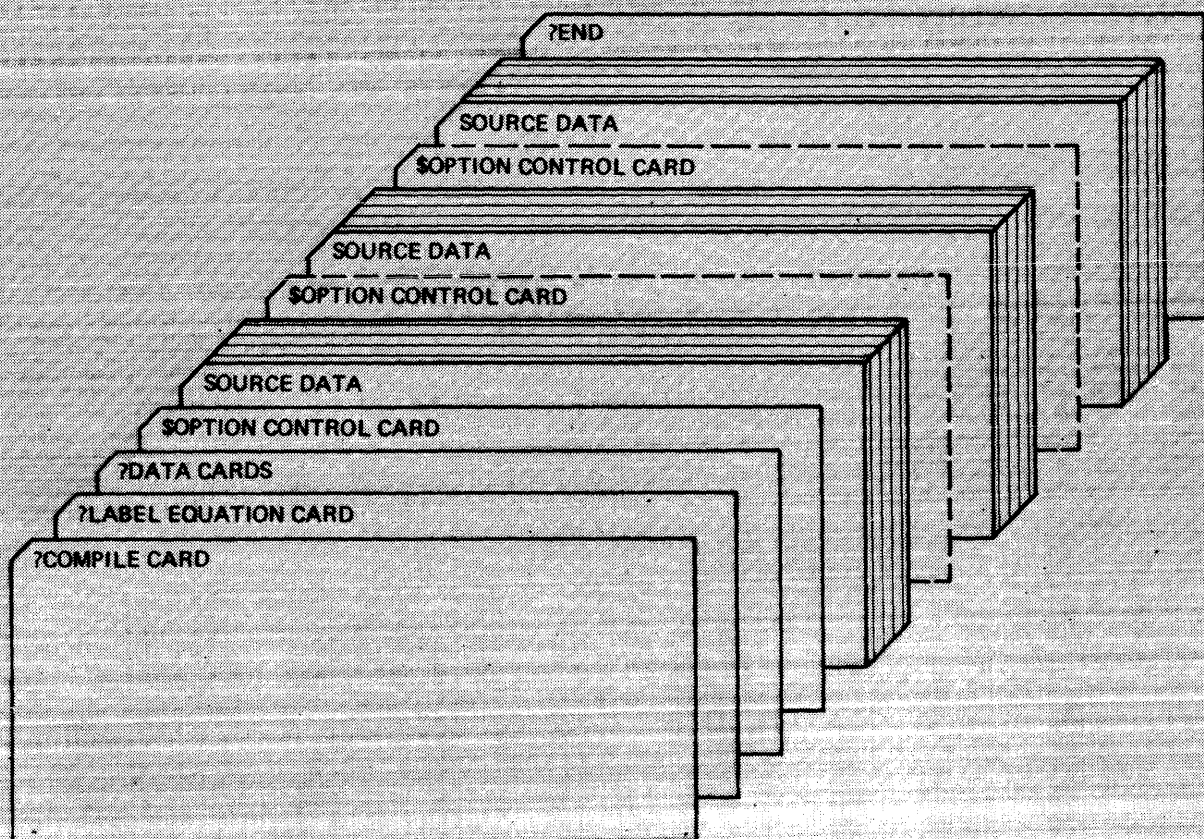


Figure 11-1. Compilation Card Deck

The compilation card deck is comprised of several cards; these cards, along with a detailed discussion of their function, are presented in the paragraphs that follow.

?COMPILE CARD

The first input control card instructs the MCP to call the COBOL compiler and to compile the indicated program-name (P-N) by means of one of the following options:

- a. To compile and run the resultant object program, the card is coded:
 ?COMPILE P-N WITH COBOL
- b. To compile for a syntax check only, the card is coded:
 ?COMPILE P-N WITH COBOL SYNTAX
- c. To compile and place the resultant object code into the Systems Library, the card is coded:
 ?COMPILE P-N WITH COBOL LIBRARY
- d. To compile and place the resultant object code into the Systems Library, and then run the object program, the card is coded:
 ?COMPILE P-N WITH COBOL SAVE

NOTE

The word WITH is for readability only and may be excluded from the above statements.

MCP LABEL CARD

The second control card, excluding Label Equation cards, is the MCP LABEL Card and is formatted in the following form:

 ?DATA CARDS (indicates EBCDIC or BCD source language input).

SOPTION CONTROL CARD

The COBOL compiler option control card (\$ sign in column 7) is used to notify the compiler as to which options are required during the compilation. If this card is omitted, \$CARD LIST CHECK SINGLE CONTROL will be assumed. There must be at least one space between each item on the control card. The options may be in any order. Columns 1 through 6 of the \$ card are used for sequence numbers. Any number of \$ cards may be used and may appear anywhere in the source deck. The options specified will become either active or inactive from that point on. The options available for the COBOL compiler option control cards are as follows:

ANSI	Causes the compiler to inhibit certain non-ANSI extensions. For example, the ELSE clause associated with AT END, or ON SIZE ERROR.
CARD	This option is for documentation only. The input is from the source language cards or from paper tape.
CHECK	Causes the compiler to check for sequence errors and print a warning message for each sequence error. The CHECK option is set, by default, at the beginning of each compile, but may be terminated with the NO option.
CODE	Lists the object code following each line of source code from the point of insertion.
CONTROL	Prints the \$ Option Control Cards on the output listing. The LIST option must be set.
DOUBLE	Causes the output listing to be printed in a double-spaced format.
LIST	Creates a single-spaced output listing of the source language input, with error and warning message (or both), where required.
LISTP	This option causes the compiler to print a listing of source images and to print errors as they occur.
MERGE	Primary input is from a source other than a card reader and may be merged with a patch deck in the

card reader. It is assumed to be from a disk file, with a file-ID of COBOLW/SOURCE, by default. If it is desirable to change the input file-ID or change the input device from disk to tape, a label equation card must be used. The NEW option may be used with the MERGE option to create a new output source file plus changes.

NEW

Creates a NEW output source file with changes, if any, entered through the use of the MERGE option, but does not include the compiler option cards, if any, which must be merged in from the card reader when the compilation is from disk or tape. The output file will be created on disk, by default, with the file-ID of COBOLW/SOURCE. If it is desirable to change the output file-ID device from disk to tape, a label equation card must be used.

NO

When the NO option precedes one of the options (with the exception of MERGE which cannot be terminated), it will terminate the function of that option.

NO DEBUG

When this option is specified, the compiler will not generate monitor object code even though the statements are left in the source program. This permits the user to approximate a conditional compile for the debugging facilities.

NO SEQ

Terminates the SEQ option and resumes using the sequence number in the source statement as it is read in.

NOCOP

This option causes the compiler to generate current operand table entries in-line in the code. This option requires more memory in order to run, but it will increase execution speed by approximately 2 per cent.

**Non-numeric
literal**

Inserted in columns 73-80 of all following card image for creation of a new source file and/or listing. This option can be reset or

	set by a subsequent control card, with the area between the quote marks containing blank characters.
REF	Provides monitoring of data-names specified in the MONITOR declaration and referenced in the program, even if the data-name values are unchanged. This option must be used in conjunction with the MONITOR statements.
SEQ	Starts resequencing the output listing and the new source file, if applicable, from the last sequence number read in and it increments the sequence number by 10 or by the last increment presented in a previous \$ option card. When resequencing starts at the beginning of the program source statements, the sequence will start with 000010.
SEQ nnnnnn	Starts resequencing the output listing and new source file, if applicable, from the sequence number specified by nnnnnn, incrementing the sequence numbers by 10.
SEQ + nnnnnn	Starts resequencing the output listing and new source file, if applicable, from the last sequence number read in, incrementing by the number specified by +nnnnnn. When resequencing starts at the beginning of the program source statements, the sequence will start with 000010.
SEQ nnnnnn+nnnnnn	Starts resequencing the output listing and new source file, if applicable, from the sequence number specified by nnnnnn, incrementing by the value of +nnnnnn.
SINGLE	Causes the output listing to be printed in a single-spaced format.
SPEC	This option negates the CONTROL and LIST options and causes only the syntax errors and associated source code to be printed if syntax errors occur. Otherwise, the CONTROL and LIST options remain in effect.

STACK nnnnn

Allows the Perform Stack size to be changed. The Perform Stack is used to save the 39-bit return address associated with a PERFORM. The return address is removed from the stack when the terminal paragraph of that PERFORM is reached. The default value for the Perform Stack is 1000 bits.

SUPPRESS

Suppresses all warning messages except sequence error messages. The sequence error message can be suppressed with the NO CHECK option.

The MERGE option without the NEW option allows a disk or tape input file to be referenced and to have external source images included from the card reader on the output listing and in the object program. A new output source file will not be created.

Columns 1-6 of the compiler option control card may be left blank when compilation is from cards. A sequence number is required when compilation is from tape or disk, if the insertion of the \$ option is requested within the source input.

SOURCE DATA CARD

Source data cards follow the \$ option control cards. The following source cards are used to create an updated version of the source input file or to cause temporary changes to the tape or disk source language input:

- a. VOID nnnnn Patch Card. The punch sequence number in card columns 1-6 is followed by a \$ in column 7, and then the word VOID. This will delete the source records from the sequence number in the first six positions of the VOID card through the sequence number specified by nnnnn. If "n" is left blank only the source record identified by the sequence number in the VOID card will be deleted from the compilation and the output listing, tape or disk files.
- b. Change or Addition Patch Card. Punch sequence number in card columns 1-6 and changed or added source language data in application card columns. These cards must be in the proper sequence for the source input file in order to be properly merged into that file.

The COBOL compiler has the capability of merging inputs from two sources (punched cards or paper tape, either of which may be merged with magnetic tape or disk) on the basis of the sequence numbers.

When merging inputs, the output compilation listing will indicate all inserts and replacements (or both).

All of the \$ options may be inserted at any point within the source language input data. Once an option has been set it will remain set until reset with the NO option in another \$ option card. In the case of the non-numeric literal it must be reset by coding a non-numeric literal of blanks.

LABEL EQUATION CARD

This card may be used to change a compiler file-name in order to avoid duplication of file-names when operating in a multiprogramming environment.

The label equation card must be used in conjunction with the MERGE or NEW options when the primary input or output is magnetic tape, the input disk file does not have a file-ID of COBOLW/SOURCE, or when a file-ID other than COBOLW/SOURCE is desired for the new disk output file.

The internal file-names and external file-ids of the COBOL compiler are used for label equation as follows:

<u>INTERNAL FILE-NAME</u>	<u>EXTERNAL FILE-ID</u>	<u>DESCRIPTION</u>
CARDS	CARDS	Input file from the card reader. If \$ MERGE is used this file will be merged with the input file on disk or tape. The default input is from the card reader.
SOURCE	COBOLW/SOURCE	Input file from disk or tape when the MERGE option is used. The default input is from disk.
NEWSOURCE	COBOLW/SOURCE	Output file to disk or tape for a NEW source file when the NEW option is used. The default output is to disk.
LINE	LINE	Source output listing to the line printer.

The following are examples of the label equation uses.

Example 1:

To compile a COBOL program from the card reader and create a copy of the source program blocked five on a disk file with the file-ID of COBOL/TEST1, the following Label Equation (FILE) cards could be used:

```
? COMPILE P-N WITH COBOL SYNTAX
? FILE NEWSOURCE NAME COBOL/TEST1 RECORDS.BLOCK 5;
? DATA CARDS
  $ CARD LIST DOUBLE NEW
... SOURCE PROGRAM DECK ...
? END
```

To create the same program file on magnetic tape, use the following FILE card:

```
? FILE NEWSOURCE NAME COBOL/TEST1 TAPE RECORDS.BLOCK 5;
```

Example 2:

To compile a COBOL program from a disk file which had been created by the default option of the \$ NEW option and to create a new source file on disk with the file-ID of TEST2, the following FILE card could be used:

```
? COMPILE P-N WITH COBOL SYNTAX
? FILE NEWSOURCE NAME = TEST2;
? DATA CARDS
  $ MERGE NEW
... PATCH CARDS IF ANY ...
? END
```

If the input file had a file-ID of COBOL/TEST1, in place of the default file-ID of SOURCE the following FILE card should have also been used in the above examples.

```
? FILE SOURCE NAME COBOL/TEST1;
```

DATA MANAGEMENT

COBOL SOURCE PROGRAM SYNTAX

The interface between COBOL and the B 1800/B 1700 Data Management System II (DMSII) is provided by extensions to the standard COBOL compiler. The extensions only affect non-data-base users of COBOL in that new words have been added to the reserved word list. No special compiler or extra control cards are required for the data base user.

IDENTIFICATION DIVISION AND ENVIRONMENT DIVISION

No additions or modifications have been made to either the IDENTIFICATION DIVISION or the ENVIRONMENT DIVISION.

DATA DIVISION**General**

The COBOL program refers to the data base in terms defined by a Data and Structure Definition Language (DASDL) program. The names of data sets, sets, and items which may be used are those described to DASDL.

Data-Base Section

The names and other relevant properties are made known to the COBOL compiler by the addition of a new section, the DATA-BASE SECTION, which has been inserted into the DATA DIVISION. The presence of this section in a COBOL source program causes the compiler to bring in (from the DASDL-created library files) the DATA DIVISION record(s) describing the specified data sets. One work area, sufficient to hold one data set record is established for each data set INVOKEd in the DATA-BASE SECTION. The syntax for data set invocation is as follows:

```
DB data-base-name.  
01 internal-data-set-name  
    INVOKE external-data-set-name
```


INVOKE is not an executable statement, but rather a generative declaration of data in the data-base to be used by the program.

The internal-data-set-name is an identifier which is used locally within the COBOL program to refer to the external data-base data-set-name.

The same data-set-name may be invoked more than once. This will give the user more work areas and more than one current occurrence, so that two or more records of the same data set are available and two or more records can be pointed at simultaneously.

Standard COBOL rules for unique definition of every variable also apply to data-base names. This requires that no name can appear twice in the same hierarchy branch.

Level numbers in the generation represent the hierarchy level in the same manner that COBOL level numbers do. Embedded data sets are listed at the embedded item level. This is a reflection of the manner in which the data sets must be accessed. The embedded level listing of the data-set-name is analogous to a reference to the embedded set through which the data set items can be located.

PROCEDURE DIVISION

Within an application program, all data in the data-base must be referenced through the data-base statements. Once the data has been retrieved from the data-base it can be manipulated by any COBOL statements, provided there are no item priority conflicts. Extensions to the COBOL language have been added for the processing of a data-base. Any number of application programs may access the data base or data bases without knowledge of one another.

Data-base items are of type NUMBER and ALPHA and have the same properties as computational items and alphanumeric items, respectively.

Special data management statements within the PROCEDURE DIVISION may refer to data sets which have been implicitly or explicitly INVOKEd. The compiler verifies syntax of all data base operations. If the syntax is acceptable, the compiler generates communicates to the DMSII access routines of the MCP.

Each data base statement may contain an exception statement which will be executed if an abnormal condition is encountered. The exceptions are noted for each statement. The user is able to interrogate the system in order to identify the exception condition so that appropriate action may be taken at that time.

INTRODUCTION

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 a key within that record.

The B 1800/B 1700 COBOL compiler utilizes the ANSI-74 syntax for the Indexed I-O module, with the exception of the DELETE verb and the ALTERNATE KEYS option which are not implemented.

LANGUAGE CONCEPTS

Organization

A file whose organization is indexed is a mass storage file in which data records may be accessed by the value of a key. A record description includes a key data item, which is associated with an index. This 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 record key for that file. For purposes of inserting, and updating records in a file, each record is identified solely by the value of its record key. This value must be unique and must not be changed when updating the record.

Access Modes

In the SEQUENTIAL access mode, the sequence in which records are accessed is the ascending order of the record key values.

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 a record key data item.

In the DYNAMIC access mode, the programmer can change the program 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 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, 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 one of five values to indicate one of five conditions upon completion of the input-output operation. The possible values and the conditions they indicate follow:

<u>Value</u>	<u>Condition</u>
0	Successful Completion
1	At End
2	Invalid Key
3	Permanent Error
9	Implementor Defined

The meanings of the above conditions 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:
 - 1. Sequence error.
 - 2. Duplicate key.
 - 3. No record found.

- 3 - Permanent Error - The input-output statement was unsuccessful as the result of an input-output error, such as data check, parity error, or a transmission error.
- 9 - Implementor Defined - The input-output statement was unsuccessfully executed as a result of a condition that is specified by the implementor. This value is used only to indicate a condition not indicated by other defined values of status key 1, or by specified combinations 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:

- a. If no further information is available concerning the input-output operation, then status key 2 contains the value of 0.
- b. 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 as follows:
 - 1. A value of 1 in status key 2 indicates a sequence error for a sequentially accessed indexed file. The ascending sequence requirements of successive record key values have been violated (refer to the WRITE statement), or the 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. A value of 2 in status key 2 indicates a duplicate key value. An attempt was made to WRITE or REWRITE a record that would create a duplicate key in an indexed file.
 - 3. A value of 3 in status key 2 indicates no record was found. An attempt was made to access a record, identified by a key, and that record does not exist in the file.
- c. When status key 1 contains a value of 9 indicating an implementor-defined condition, the value of status key 2 is defined as follows:
 - 1. A value of 3 in status key 2 indicates that the record key value contained in the data file record is not equal to the record key value contained in the associated tag file record.

Valid Combinations of Status Keys

The valid permissible combinations of the value of status key 1 and status key 2 are shown in table 13-1. An 'X' at an intersection indicates a valid permissible combination.

Table 13-1. Status Key Combinations

Status Key 1 Condition	Status Key 2			
	No Further Information (0)	Sequence Error (1)	Duplicate Key (2)	No Record Found (3)
Successful Completion (0)	X			
At End (1)	X			
Invalid Key (2)		X	X	X
Permanent Error (3)	X			
Implementor Defined (9)				X

The Invalid Key Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, or REWRITE statement. For details on the causes of the condition, refer to the subsections entitled The START Statement, The READ Statement, The WRITE Statement, and the REWRITE Statement in this section.

When the INVALID KEY condition is recognized, actions are taken in the following order:

- a. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an INVALID KEY condition. Refer to I-O Status for additional information.
- b. If the INVALID KEY phrase is specified in the statement causing the condition, control is transferred to the INVALID KEY statement. Any USE procedure specified for this file is not executed.
- c. 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 on the causes of the condition, refer to the subsection entitled The READ Statement.

GENERAL IMPLEMENTATION INFORMATION

Physical Files

Indexed files, as implemented, consist of two physical files:

- a. The data file contains records written to the file by the program, as is maintained in the order in which records are first written.
- b. The tag file contains the record key value and the relative record number of its associated record in the data file. This file is ordered by record key value, and is read, written, and maintained transparent to the COBOL user by compiler-generated subroutines. These generated subroutines, or Indexed Control System (ICS), also provided for the opening, closing, and resequencing of the tag file.

Tag Files

A tag file record contains a record key and a pointer to the relative location in the data file of the data record whose record key is the same value as that in the tag file record. The record key is an exact copy of the record key field in the data file record. Duplicate keys are not allowed. The tag file is blocked so that as many records as possible are placed into a 180-character block. If the record key is greater than 176 characters in length, the tag file is unblocked.

The internal and external names of the tag file are generated by the ICS by prefixing TAG to the internal and external names of the data file as shown in table 13-2.

Table 13-2. Tag File and Data File Names

Internal File Names		External File Names	
Data File	Tag File	Data File	Tag File
FILE1	TAGFILE1	A/B	TAGA/B
FILE2	TAGFILE2	FILE2	TAGFILE2
FILE3	TAGFILE3	CCC/A/B	CCC/TAGA/B
FILE4	TAGFILE4	A/ABCDEFGHI	TAGA/ABCDEFGHI
FILE5	TAGFILE5	A/B/	A/TAGB/

RPG and COBOL Compatibility

In order to accommodate those users wishing to process indexed files by both RPG and COBOL programs, a new dollar option named RPGTAGS has been added to the COBOL compiler. This option, when set, causes the compiler to generate code to process indexed files using tag files with the same format and external names as if those tag files had been created by an RPG program. The files cannot be used if the program is executed under File Security (unless run with a privileged usercode/password).

The difference in tag file format is due to the use, by COBOL, of an eight-digit pointer in the tag file records where RPG uses a six-digit pointer. This larger pointer enables COBOL users to process files larger than RPG limitations allow.

The difference in the external names of the tag files is due to the prefixing by COBOL of the word "TAG" to the main-directory name of the data file to produce the tag file name, where RPG prefixes the word "TAG" to the sub-directory name. Prefixing to the main-directory name rather than the sub-directory name enables COBOL to generate, in some cases, a single file-name for the tag file. This allows indexed files to be processed under the File Security System.

If RPGTAGS had been specified in the COBOL program which created the tag files in the preceding example, the external tag file names would have been created as follows:

- a. A/TAGB
- b. FILE2/TAG
- c. CCC/A/TAGB
- d. A/TAGABCDEFGH
- e. A/B/TAG

Another point to consider if the user wishes to access the same indexed files with both COBOL and RPG programs, is that only COBOL permits alphanumeric keys.

Rough Tables

In order to minimize disk accesses when searching the tag file for record keys, the ICS, as a function of opening the tag file, creates a rough table in memory which logically breaks the tag file into partitions.

The rough table consists of as many entries as fit into the space provided or by default, ten (10) entries. Refer to the subsection entitled The VALUE OF Clause. Each entry consists of that record key which occupies the last record in that partition. The number of records contained in a partition is determined by the number of records in the file divided by the number of table entries allowed.

A rough table is built for all opens of files with ORGANIZATION INDEXED except for those OPENed OUTPUT with SEQUENTIAL ACCESS, and those which are OPENed INPUT with SEQUENTIAL ACCESS for which there is no START statement contained in the program. As the rough table is built, the entries are sequence checked to ensure that the binary search algorithm functions correctly. If the entries are not in sequence, the program is terminated.

Searching The Tag File

A binary search method is used to locate the desired key between successive entries in the rough table. The partition boundaries are then calculated by multiplying these entry numbers by the number of tag records contained in the partitions. A binary search of the tag file between those boundaries then occurs to find the required record. When the tag record is found, the relative record number of its associated record in the data file is used to make the data record available to the COBOL program. If the key is not found in the above manner and records have been added to the file since the last reordering of the tag file, the records which have been added to the tag file are serially searched for the required key.

In an effort to minimize the number of disk reads required to locate a particular record, the following optimizations are used:

- a. The required key is checked for being equal to the key found in the rough table. If they are equal, the tag file record is read without requiring the binary search of that partition.

- b. If the partition found in the rough table contains the End-Of-File record, the required key is compared to the key of the End-Of-File record. If the requested key is greater, the added records (if any) are searched, and the binary search of that partition is avoided.
- c. When records are added, the highest and lowest keys added are saved. The required key is range-checked with those values. If the required key is outside that range, the serial search of the added records is unnecessary.

Adding Records to the Files

When the tag and data files are opened, their End-Of-File pointers (which must be equal) are obtained. This End-Of-File pointer (EOF-Pointer) is placed in New-EOF-Pointer. New records are appended to the end of each file by adding a one to the value of New-EOF-Pointer, then using the new value as the actual key of both the tag and data files.

When the file is CLOSED, or when a START, or READ ... NEXT statement is executed and records have been added to the file, the tag file is sorted. This does not apply to ACCESS SEQUENTIAL.

ENVIRONMENT DIVISION FOR INDEXED I-O

The File-Control Paragraph

Function

The FILE-CONTROL entry names a file and may specify other file-related information. This description of the FILE-CONTROL entry only describes those clauses which are limited to the Indexed I-O module.

General Format

SELECT file-name ASSIGN TO [DISK
DISK PACK]

[ORGANIZATION IS INDEXED

[ACCESS MODE IS { SEQUENTIAL
RANDOM
DYNAMIC }]

[RECORD KEY IS data-name-1]

[FILE STATUS IS data-name-2]

Syntax Rules

- a. The SELECT clause must be specified first in the FILE-CONTROL entry. The clauses which follow the SELECT clause may appear in any order.
- b. 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.
- c. If the ACCESS clause is not specified, ACCESS MODE IS SEQUENTIAL is implied.
- d. 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.
- e. Data-name-1, and data-name-2 may be qualified.
- f. 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.
- g. Data-name-1 cannot describe an item whose size is variable. Refer to the subsection entitled The OCCURS Clause for additional information.

General Rules

- a. The ASSIGN clause specifies the association of the file referenced by file-name to a mass storage medium.
- b. 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.
- c. 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.
- d. If the access mode is RANDOM, the value of the record key data item indicates the record to be accessed.
- e. When the access mode is DYNAMIC, records in the file may be accessed sequentially and/or randomly. Refer to general rules c and d.
- f. When the FILE STATUS clause is specified, a value is moved by the ICS 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 that statement.

- g. The RECORD KEY clause specifies the record key for the file. The values of the record key must be unique among records of the file. This record key provides the access path to records in an indexed file.
- h. The data description of data-name-1 as well as its relative location within a record must be the same as that used when the file was created.

DATA DIVISION FOR INDEXED I-O

The File Description

Function

The file description furnishes information concerning the physical structure, identification, and record names pertaining to a given file. This discussion of the file description entry (FD) provides details only of those clauses which have special meaning in the Indexed I-O module.

General Format

FD file-name

$$\left[\left\{ \begin{array}{l} \text{VALUE} \\ \text{VA} \end{array} \right\} \text{ OF } \left\{ \begin{array}{l} \text{IDENTIFICATION} \\ \text{ID} \end{array} \right\} \text{ IS } \left\{ \begin{array}{l} \text{data-name-1} \\ \text{literal-1} \end{array} \right\} \right]$$

[VALUE OF CORE-INDEX IS integer-1 CHARACTERS]

Syntax Rules

- a. The level indicator FD identifies the beginning of a file description and must precede the file-name.
- b. The clauses which follow the name of the file are optional in many cases, and their order of appearance is immaterial.
- c. One or more record description entries must follow the file description entry.

The VALUE OF Clause

Function

The VALUE OF clause provides for the identification of a data file and its related tag file, as well as allowing specification of the amount of memory to be available to contain the rough table.

General Format

$$\left\{ \begin{array}{c} \underline{\text{VALUE}} \\ \underline{\text{VA}} \end{array} \right\} \text{ OF } \left\{ \begin{array}{c} \underline{\text{IDENTIFICATION}} \\ \underline{\text{ID}} \end{array} \right\} \text{ IS } \left\{ \begin{array}{c} \text{data-name-1} \\ \text{literal-1} \end{array} \right\}$$

[VALUE OF CORE-INDEX IS integer-1 CHARACTERS]

Syntax Rule

- a. Data-name-1 should be qualified when necessary, but cannot be subscripted or indexed, nor can it be an item described with the USAGE IS INDEX clause.

General Rules

- a. For the IDENTIFICATION option:
 1. For an input file, the MCP checks to see if a data file whose name is equal to the value of literal-1, or of data-name-1, whichever has been specified, as well as its associated tag file are present in the disk directory.
 2. For each output file, a data file whose name is made equal to the value of literal-1 or of data-name-1, as well as its associated tag file are inserted into the disk directory. Refer to the subsection entitled Tag Files.
- b. For the CORE-INDEX option, the number of characters specified by integer-1 is allocated for use in the storage of a rough table. If this option is omitted, the number of characters required to contain ten (10) entries is allocated. Refer to the subsection entitled Rough Tables for additional information.

PROCEDURE DIVISION FOR INDEXED I-O

The verbs in the PROCEDURE DIVISION are CLOSE, OPEN, READ, REWRITE, START, USE and WRITE.

The CLOSE Statement

Function

The CLOSE statement terminates the processing of a file and specifies the disposition of the file.

General Format

CLOSE file-name-1 [WITH { LOCK
RELEASE
REMOVE
PURGE }]

[file-name-2 [WITH { LOCK
RELEASE
REMOVE
PURGE }]] ...

Syntax Rule

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

General Rules

- a. A CLOSE statement can only be executed for a file in an open mode.
- b. Indexed files are labeled mass storage files. The results of executing each type of CLOSE are summarized in table 13-3.

Table 13-3. File Close Type and Disposition

CLOSE Statement Format	File Disposition
CLOSE	1, 3
CLOSE WITH LOCK	1, 2, 4
CLOSE WITH RELEASE	1, 2, 4
CLOSE WITH REMOVE	1, 4, 5
CLOSE WITH PURGE	1, 6

The definitions of the numbers in table 13-3 follows:

- 1 Close file
The CLOSE operation marks the logical file as closed.
- 2 File Lock
The physical file is made a permanent file and is entered into the MCP Disk Directory.

3 File Retention

The association between the logical file and the physical file is retained. Subsequent reopening of the file may not require the MCP to search the Disk Directory for the physical file. A new file is not entered in the Disk Directory, and is lost at End-of-Job or with a subsequent OPEN output, if not CLOSED with LOCK, RELEASE, or REMOVE.

4 File Release

The association between the logical file and the physical file is severed. The areas of memory allocated for buffers may be released to the system.

5 File Remove

This option causes the MCP to remove a file from the Disk Directory which has the same external name as the file being CLOSED. This action takes place prior to entering the external name in the Disk Directory.

6 File Purge

If the file is a permanent file, the file name is removed from the Disk Directory, and the storage area occupied by the file is released as available to the MCP.

- c. If a file is in the open mode when a STOP RUN statement is executed, or when an abnormal termination occurs, the action taken is to close the file as if a simple CLOSE statement was executed.
- d. If a CLOSE statement was 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.

The execution of a CLOSE statement has no effect upon the contents or the availability of the file's record area.

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

OPEN {
 INPUT file-name-1 [WITH LOCK [ACCESS]]
 [file-name-2 [WITH LOCK [ACCESS]]] ...
 OUTPUT file-name-3 [file-name-4] ...
 I-O file-name-5 [WITH LOCK [ACCESS]]
 [file-name-6 [WITH LOCK [ACCESS]]] ...
}

Syntax Rule

- a. The fix referenced in the OPEN statement need not all have the same organization or access.

General Rules

- a. The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode.
- b. The execution of an OPEN statement does not affect either the contents or availability of the file's record area.
- c. When a given file is not in an open mode, no statement (except for a SORT or MERGE statement with the USING or GIVING phrases) can be executed that references that file, either explicitly or implicitly.
- d. 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 OPEN statement execution for that same file must be preceded by the execution of a CLOSE statement for that file.
- e. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. In table 13-4, Permissible Statements, an X at an intersection indicates that the specified statement, used in the File Access Mode given for that row, may be used with the Open Mode given at the top of the column.

Table 13-4. Permissible Statements

File Access Mode	Statement	Open Mode		
		Input	Output	Input-Output
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
RANDOM	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
DYNAMIC	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X

- f. Execution of the OPEN statement does not obtain or release the first data record.
- g. 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.
- h. 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 record key is used to determine the first record to be accessed. If no records exist in the files, the current record pointer is set such that the next executed Format 1 READ statement for the file results in an AT END condition.
- i. The I-O phrase permits the opening of a file for both input and output operations. Since this phrase implies the existence of the file, it cannot be used if the file is being initially created.
- j. 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.
- k. When an OPEN WITH LOCK option (without ACCESS) is executed, the following occurs:
 - 1. If the specified file is already in an open mode by another program or by a different file-name within this program, the program is suspended awaiting exclusive availability of the file.
 - 2. If the file is not currently in an open mode by any program, the file is OPENed.

1. When an OPEN WITH LOCK ACCESS is executed the program is suspended if any of the following conditions exist:
 1. The file is currently OPEN WITH LOCK.
 2. The file is currently OPEN I-O.
 3. This OPEN is I-O and the file is currently OPEN INPUT LOCK ACCESS.

The program remains suspended until none of these conditions hold, or until the program is terminated. In any other case, the OPEN WITH LOCK ACCESS causes the file to be OPENed.

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

The two formats of the READ statement are as follows:

Format 1:

```

READ file-name [NEXT] RECORD [INTO identifier]
  [;AT END statement-1 [;ELSE statement-2]]

```

Format 2:

```

READ file-name RECORD [INTO identifier]
  [KEY IS data-name]
  [;INVALID KEY statement-1 [;ELSE statement-2]]

```

Syntax Rules

- a. 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.
- b. Data-name must be the name of the data item specified as the record key associated with file-name.
- c. Data-name may be qualified.
- d. Format 1 must be used for all files in SEQUENTIAL access mode.
- e. The NEXT phrase must be specified for files in DYNAMIC access mode, when records are to be retrieved sequentially.

- f. Format 2 is used for files in RANDOM access mode or for files in DYNAMIC access mode when records are to be retrieved randomly.
- g. The INVALID KEY phrase or the AT END phrase must be specified if no applicable USE procedure is specified for file-name.

General Rules

- a. The associated file must be open in the INPUT or I-O mode at the time this statement is executed. Refer to the subsection entitled The OPEN Statement for additional information.
- b. The record to be made available by a Format 1 READ statement is determined as follows:
 - 1. The record, pointed to by the current record pointer, is made available provided that the current record pointer was positioned by a START or OPEN statement.
 - 2. 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.
- c. The execution of the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. Refer to the subsection entitled I-O status for additional information.
- d. 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.
- e. When the logical records of a file are described with more than one record description, these records automatically share storage area; this is equivalent to an implicit redefinition of the area.
- f. 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 with the sending area considered to be a group item whose size is equal to the maximum record size for this file.

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.

- g. 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.
- h. 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.
- i. If, at the time of the 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. Refer to the subsection entitled I-O Status for additional information.
- j. When the AT END condition is recognized, the following actions are taken in the specified order:
 - 1. A value is placed into the FILE STATUS data item, if specified for this file, to indicate an AT END condition. Refer to the subsection entitled I-O Status for additional information.
 - 2. If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END statement. Any USE procedure specified for this file is not executed.
 - 3. If the AT END phrase is not specified, then a USE procedure must be specified, 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.

- k. 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.
- l. 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:
 - 1. A successful CLOSE statement followed by the execution of a successful OPEN statement for that file.
 - 2. A successful START statement for that file.
 - 3. A successful Format 2 READ statement for that file.
- m. 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 b.

- n. For an indexed file if the KEY phrase is specified in a Format 2 READ statement, the phrase is used for documentation only.
- o. If the KEY phrase is not specified in a Format 2 READ statement, the record key is used for this retrieval. If the DYNAMIC access mode is specified, this key is also used for retrievals by any subsequent executions of Format 1 READ statements for the file.
- p. Execution of a Format 2 READ statement causes the value of the record key to be compared with the value contained in the corresponding data item of the stored records in the file, until the record having an equal value is found. The current 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. Refer to the subsection entitled The INVALID KEY Condition for additional information.

The REWRITE Statement

Function

The REWRITE statement logically replaces a record existing in a mass storage file.

General Format

```

REWRITE record-name [FROM identifier]
    [;INVALID KEY statement-1 [;ELSE statement-2]]
  
```

Syntax Rules

- a. Record-name and identifier must not refer to the same storage area.
- b. Record-name is the name of the logical record in the File Section of the Data Division and may be qualified.
- c. The INVALID KEY phrase must be specified in the REWRITE statement for files for which an appropriate USE procedure is not specified.

General Rules

- a. The file associated with record-name must be open in I-O mode at the time of execution of this statement. Refer to the subsection entitled the OPEN Statement for additional information.
- b. 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 record that was accessed by the READ statement is logically replaced.

- c. 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.
- d. The logical record released by a successful execution of the REWRITE statement is no longer available in 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.
- e. 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.

- f. The current record pointer is not affected by the execution of a REWRITE statement.
- g. The execution of the REWRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. Refer to the subsection entitled I-O Status for additional information.
- h. For a file in the SEQUENTIAL mode, the record to be replaced is specified by the value contained in the record key. When the REWRITE statement is executed, the value contained in the record key data item of the record to be replaced must be equal to the value of the record key of the last record read from this file.
- i. For a file in the RANDOM or DYNAMIC access mode, the record to be replaced is specified by the record key data item.
- j. The INVALID KEY condition exists when:
 - 1. The access mode is SEQUENTIAL and the value contained in the record key data item of the record to be replaced is not equal to the value of the record key of the last record read from this file, or
 - 2. The value contained in the record key data item does not equal that of any record stored in the file.

The updating operation does not take place and the data in the record area is unaffected. Refer to the subsection entitled The INVALID KEY Condition for additional information.

The START Statement

Function

The START statement provides a basis for logical positioning within an indexed file, for subsequent sequential retrieval of records.

General Format

$$\left[\begin{array}{l} \text{START file-name} \quad \text{KEY IS} \left\{ \begin{array}{l} \underline{\text{EQUAL TO}} \\ = \\ \underline{\text{GREATER THAN}} \\ > \\ \underline{\text{NOT LESS THAN}} \\ \underline{\text{NOT}} < \end{array} \right\} \text{data-name} \end{array} \right]$$

[; INVALID KEY statement-1 |; ELSE statement-2|]

NOTE

The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols such as "greater than or equal to".

Syntax Rules

- a. File-name must be the name of an indexed file.
- b. File-name must be the name of a file with SEQUENTIAL or DYNAMIC access.
- c. Data-name may be qualified.
- d. The INVALID KEY phrase must be specified if no applicable USE procedure is specified for file-name.
- e. If the KEY phrase is specified, data-name must reference the data item specified as the record key associated with file-name.

General Rules

- a. File-name must be open in the INPUT or I-O mode at the time that the START statement is executed. Refer to the subsection entitled The OPEN Statement for additional information.
- b. If the KEY phrase is not specified the relational operator "IS EQUAL TO" is implied, and the comparison described below uses the data item referenced in the RECORD KEY clause associated with file-name.

- c. The type of comparison specified by the relational operator in the KEY phrase occurs between the key associated with a record in the file referenced by the file-name and the data item specified as the record key. The results of that comparison are as follows:
 - 1. The current record pointer is positioned to the first logical record currently existing in the file whose record key satisfies the comparison.
 - 2. 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. Refer to the subsection entitled The INVALID KEY Condition for additional details.
- d. The execution of the START statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated. Refer to the subsection entitled I-O Status for additional details.
- e. Upon completion of the successful execution of the START statement, the current record pointer is established and used in the subsequent Format 1 READ statement. Refer to the subsection entitled The READ Statement for additional details.
- f. If the execution of the START statement is not successful, the current record pointer is undefined.

The USE Statement

Function

The USE statement specifies procedures for input-output error or exception handling that are in addition to the standard procedures provided by the MCP.

General Format

```

USE AFTER STANDARD { EXCEPTION } PROCEDURE ON
                        { ERROR }

{ file-name-1 [;file-name-2] ... }
{ INPUT }
{ OUTPUT }
{ I-O }

```


Syntax Rules

- a. 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 one or more procedural paragraphs that define the procedures to be used.
- b. The USE statement itself is never executed; it merely defines the conditions calling for the execution of the USE procedures.
- c. 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.
- d. The words ERROR and EXCEPTION are synonymous and can be used interchangeably.
- e. The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

General Rules

- a. The designated procedures are executed by the ICS after completing the standard input-output error routine, or upon recognition of the INVALID KEY or AT END conditions, when the INVALID KEY phrase or AT END phrase, respectively, has not been specified in the input-output statement.
- b. After execution of a USE procedure, control is returned to the invoking routine.
- c. Within a USE procedure, there must not be any reference to any non-DECLARATIVE procedures. Conversely, in the non-DECLARATIVE portion there must be no reference to procedure-names that appear in the DECLARATIVE portion, except that PERFORM statements can refer to a USE procedure or to the paragraphs contained within the USE procedure.
- d. 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 statement-1 [;ELSE statement-2]]
```

Syntax Rules

- a. Record-name and identifier must not reference the same storage area.
- b. The record-name is the name of a logical record in the File Section of the Data Division and may be qualified.
- c. The INVALID KEY phrase must be specified if an applicable USE procedure is not specified for the associated file.

General Rules

- a. The associated file must be open in the OUTPUT or I-O mode at the time of the execution of this statement. Refer to the subsection entitled The OPEN Statement for additional information.
- b. 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 the associated output file, as well as to the file associated with record-name.
- c. The execution of the WRITE statement with the FROM phrase is equivalent to the execution of:

MOVE identifier TO record-name

followed by the execution of 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 the WRITE statement.

After execution of the WRITE statement is complete, the information in the area referenced by the identifier is available, even though the information in the area referenced by record-name may not be. Refer to general rule b, also.

- d. The current record pointer is unaffected by the execution of a WRITE statement.
- e. The execution of the WRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. Refer to the subsection entitled I-O Status for additional information.

- f. The maximum record size for a file is established at the time the file is created and must not subsequently be changed.
- g. The number of character positions of 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.
- h. The execution of the WRITE statement releases a logical record to the MCP.
- i. Execution of the WRITE statement causes the contents of the record area to be released. The ICS utilizes the content of the record key in such a way that subsequent access of that record may be made based upon the value of that specified record key.
- j. The value of the record key must be unique within the records in the file.
- k. The data item specified as the record key must be set by the program to the desired value prior to the execution of the WRITE statement. Refer to general rule c, also.
- l. If SEQUENTIAL access mode is specified for the file, records must be released to the ICS in ascending order of record key values.
- m. If RANDOM or DYNAMIC access mode is specified, records may be released to the ICS in any program-specified order.
- n. The INVALID KEY condition exists under the following circumstances:
 - 1. When SEQUENTIAL access mode is specified for a file OPENed in the output mode, and the value of the record key is not greater than the value of the record key of the previous record, or
 - 2. When the file is OPENed in the output or I-O mode, and the value of the record key is equal to the value of a record key of a record already existing in the file.
- o. 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 to a value indicating the cause of the condition. Execution of the program proceeds according to the rules stated in the subsection entitled The INVALID KEY Condition. Also, refer to the subsection entitled I-O Status for additional information.

APPENDIX A

RESERVED WORDS

All reserved words known to the B 1800/B 1700 COBOL compiler are listed in this Appendix, together with notations (X) of the DIVISION(s) in which the words are used or other reasons for reservation:

- a. for DATA MANAGEMENT (DMSII),
- b. for future use or standards requirements.

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
ABOUT	-	-	-	-	-	X
ACCEPT	-	-	-	X	-	-
ACCESS	-	X	-	-	-	-
ACTUAL	-	X	-	-	-	-
ADD	-	-	-	X	-	-
ADDRESS	-	-	-	-	-	X
ADVANCING	-	-	-	X	-	-
AFTER	-	-	-	X	-	-
ALL	X	-	X	X	-	-
ALL-AT-OPEN	-	X	-	-	-	-
ALPHABETIC	-	-	-	X	-	-
ALTER	-	-	-	X	-	-
ALTERNATE	-	X	-	-	-	-
ALTERNATING	-	-	-	-	-	X
AND	-	-	-	X	-	-
APPLY	-	X	-	-	-	-
ARE	-	X	X	-	-	-
AREA	-	X	-	-	-	-
AREAS	-	X	-	-	-	-
ASCENDING	-	-	X	-	-	-
ASCII	-	-	X	-	-	-
ASSIGN	-	X	-	-	-	-
AT	-	-	-	X	-	-
AUDIT	-	-	-	-	X	-
AUTHOR	X	-	-	-	-	-
AUTO	-	-	-	-	X	-
AUXILIARY	-	-	-	X	-	-
BACKUP	-	X	-	-	-	-
BATCH-COUNT	-	-	-	X	-	-
BEFORE	-	-	-	X	-	-
BEGINNING	-	-	-	X	-	-
BEGIN-TRANSACTION	-	-	-	-	X	-

Appendix A (Cont)

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMS II	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
BINARY	-	-	-	-	-	X
BLANK	-	-	X	-	-	-
BLOCK	-	-	X	-	-	-
BY	-	X	X	X	-	-
BZ	-	-	X	-	-	-
CARD96	-	X	-	-	-	-
CASSETTE	-	X	-	-	-	-
CF	-	-	-	-	-	X
CH	-	-	-	-	-	X
CHANNEL	-	-	-	X	-	-
CHARACTERS	-	X	X	-	-	-
CLOCK-UNITS	-	-	-	-	-	X
CLOSE	-	-	-	X	-	-
CMP	-	-	X	-	-	-
CMP-1	-	-	X	-	-	-
CMP-3	-	-	X	-	-	-
COBOL	-	-	-	-	-	X
CODE	-	-	-	-	-	X
CODEFILE	-	-	-	-	-	X
COLUMN	-	-	-	-	-	X
COMMA	-	X	-	-	-	-
COMP	-	-	X	-	-	-
COMP-1	-	-	X	-	-	-
COMP-3	-	-	X	-	-	-
COMPUTATIONAL	-	-	X	-	-	-
COMPUTATIONAL-1	-	-	X	-	-	-
COMPUTATIONAL-3	-	-	X	-	-	-
COMPUTE	-	-	-	X	-	-
CONFIGURATION	-	X	-	-	-	-
CONTAINS	-	X	X	-	-	-
CONTROL	-	-	-	X	-	X
CONTROLS	-	-	-	-	-	X
CONVERSION	-	-	-	X	-	-
COPY	X	X	X	X	-	-
CORE-INDEX	-	-	-	-	-	X
CORR	-	-	-	X	-	-
CORRESPONDING	-	-	-	X	-	-
CREATE	-	-	-	-	X	-
CRUNCH	-	-	-	X	-	-
CURRENCY	-	X	-	-	-	-
CURRENT	-	-	-	-	X	-
CYLINDER	-	-	-	-	-	X
DATA	-	X	X	-	-	-
DATA-BASE	-	-	-	-	X	-
DATASET	-	-	-	-	X	-
DATE	-	-	-	X	-	-
DATE-COMPILED	X	-	X	-	-	-
DATE-WRITTEN	X	-	-	-	-	-
DB	-	-	-	-	X	-
DDL-NUMBER	-	-	-	-	X	-

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
DE	-	-	-	-	-	X
DECIMAL-POINT	-	X	-	-	-	-
DECLARATIVES	-	-	-	X	-	-
DELETE	-	-	-	-	X	-
DEMAND	-	-	-	-	-	X
DEPENDING	X	-	X	X	-	-
DESCENDING	-	-	X	X	-	-
DETAIL	-	-	-	-	-	X
DISC	-	X	-	-	-	-
DISK	-	X	-	-	-	-
DISK-DFC1	-	X	-	-	-	-
DISK-DFC2	-	X	-	-	-	-
DISK-DPC1	-	X	-	-	-	-
DISK-DPC2	-	X	-	-	-	-
DISK-HPT	-	X	-	-	-	-
DISKPACK	-	X	-	-	-	-
DISK-PPC2	-	X	-	-	-	-
DISPLAY	-	-	X	X	-	-
DIVIDE	-	-	-	X	-	-
DIVISION	X	X	X	X	-	-
DMSTATUS	-	-	-	-	X	-
DOWN	-	-	-	X	-	-
DUMP	-	-	-	X	-	-
DYNAMIC	-	-	-	-	-	X
ELSE	-	-	-	X	-	-
ENABLE	-	-	-	-	-	X
END	-	X	-	X	-	-
END-OF-JOB	-	-	-	X	-	-
END-OF-PAGE	-	-	-	X	-	-
END-TRANSACTION	-	-	-	-	X	-
END-TRANSIT	-	-	-	-	-	X
ENDING	-	-	-	X	-	-
ENTER	-	-	-	-	-	X
ENVIRONMENT	-	X	-	-	-	-
EOP	-	-	-	X	-	-
EQUAL	-	-	-	X	-	-
EQUALS	-	-	-	X	-	-
ERROR	-	-	-	X	-	-
EVERY	-	-	-	-	-	X
EXAMINE	-	-	-	X	-	-
EXCEPTION	-	-	-	-	X	-
EXIT	-	-	-	X	-	-
FD	-	-	X	-	-	-
FILE	-	X	X	X	-	-
FILE-CONTROL	-	X	-	-	-	-
FILE-LIMIT	-	X	-	-	-	-
FILE-LIMITS	-	X	-	-	-	-
FILL	-	-	-	-	-	-
FILLER	-	-	X	-	-	-
FINAL	-	-	-	-	-	X

Appendix A (Cont)

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
FIND	-	-	-	-	X	-
FIRST	-	-	-	X	-	-
FLOW	-	-	-	-	-	X
FOOTING	-	-	-	-	-	X
FOR	-	X	-	X	-	-
FORM	-	X	-	-	-	-
FORMAT	-	-	-	X	-	-
FREE	-	-	-	-	X	-
FROM	-	-	-	X	-	-
GENERATE	-	-	-	-	-	X
GIVING	-	-	-	X	-	-
GO	-	-	-	X	-	-
GREATER	-	-	-	X	-	-
GROUP	-	-	-	-	-	X
HARDWARE-MONITOR	-	-	-	-	-	X
HEADERS	-	-	X	-	-	-
HEADING	-	-	-	-	-	X
HERE	-	-	-	-	-	X
HIGH-VALUE	-	-	X	X	-	-
HIGH-VALUES	-	-	X	X	-	-
I-O	-	-	-	X	-	-
I-O-CONTROL	-	X	-	-	-	-
ID	-	-	X	-	-	-
IDENTIFICATION	X	-	X	-	-	-
IF	-	-	-	X	-	-
IN	-	-	-	X	-	-
INC-EU	-	X	-	-	-	-
INDEX	-	-	X	-	-	-
INDEXED	-	-	X	-	-	-
INDICATE	-	-	-	-	-	X
INITIATE	-	-	-	-	-	X
INPLACE	-	-	-	X	-	-
INPUT	-	-	-	X	-	-
INPUT-OUTPUT	-	-	-	X	-	-
INQUIRY	-	-	-	-	X	-
INSERT	-	-	-	-	X	-
INSTALLATION	X	-	-	-	-	-
INTERPRET	-	-	-	X	-	-
INTO	-	-	-	X	-	-
INVALID	-	-	-	X	-	-
INVALID-REQUEST	-	-	-	-	-	X
INVOKE	-	-	-	-	X	-
IS	-	X	X	X	-	-
JS	-	-	X	-	-	-
JUST	-	-	X	-	-	-
JUSTIFIED	-	-	X	-	-	-
KEY	-	X	X	X	-	-
KEYS	-	-	-	-	X	-

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
LABEL	-	-	X	X	-	-
LAST	-	-	-	-	X	-
LEADING	-	-	-	X	-	-
LEFT	-	-	X	-	-	-
LESS	-	-	-	X	-	-
LIBRARY	-	-	-	-	-	X
LIMIT	-	-	-	-	-	X
LIMITS	-	-	-	-	-	X
LINE	-	-	-	-	-	X
LINE-COUNTER	-	-	-	-	-	X
LINES	-	-	-	X	-	-
LOCK	-	-	-	X	-	-
LOW-VALUE	-	-	X	X	-	-
LOW-VALUES	-	-	X	X	-	-
MANUAL	-	-	-	-	X	-
MEMORY	-	X	-	-	-	-
MERGE	-	-	-	X	-	-
MFCU	-	X	-	-	-	-
MICR	-	X	-	-	-	-
MICR-EDIT	-	-	-	X	-	-
MICR-OCR	-	-	-	-	-	X
MOD	-	-	-	X	-	-
MODE	-	X	-	-	-	-
MODIFY	-	-	-	-	X	-
MODULES	-	-	-	-	-	X
MONITOR	X	-	-	-	-	-
MOVE	-	-	-	X	-	-
MULTIPLE	-	X	-	-	-	-
MULTIPLY	-	-	-	X	-	-
NEGATIVE	-	-	-	X	-	-
NEXT	-	-	-	X	X	-
NO	-	X	-	X	-	-
NO-AUDIT	-	-	-	-	X	-
NO-ERRORS	-	-	-	-	-	X
NO-FORMAT	-	-	-	-	-	X
NON-STANDARD	-	-	X	-	-	-
NOT	-	-	-	X	-	-
NOT-READY	-	-	-	-	-	X
NOTE	-	-	-	X	-	-
NULL	-	-	-	-	X	-
NUMBER	-	-	-	-	-	X
NUMERIC	-	-	-	X	-	-
O-I	-	-	-	X	-	-
OBJECT-COMPUTER	-	X	-	-	-	-
OC	-	-	X	-	-	-
OCCURS	-	-	X	-	-	-
OCR	-	X	-	-	-	-
OF	-	-	-	X	-	-
OFF	-	-	-	-	-	X

Appendix A (Cont)

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
OMITTED	-	-	X	-	-	-
ON	-	-	-	X	-	-
OPEN	-	-	-	X	-	-
OPTIONAL	-	X	-	-	-	-
OR	-	X	-	X	-	-
ORDERED	-	-	-	-	X	-
ORDERING	-	-	-	-	X	-
ORGANIZATION	-	-	-	-	-	X
OTHERWISE	-	-	-	X	-	-
OUTPUT	-	-	-	X	-	-
OUTPUT-INPUT	-	-	-	X	-	-
PACK	-	-	-	-	-	X
PAGE	-	-	-	-	-	X
PAGE-COUNTER	-	-	-	-	-	X
PC	-	-	X	-	-	-
PERFORM	-	-	-	X	-	-
PF	-	-	-	-	-	X
PH	-	-	-	-	-	X
PIC	-	-	X	-	-	-
PICTURE	-	-	X	-	-	-
PLUS	-	-	-	-	-	X
POCKET	-	-	-	X	-	-
POCKET-LIGHT	-	-	-	X	-	-
POSITION	-	X	-	-	-	-
PRINTER	-	X	-	-	-	-
PRINT128	-	-	-	X	-	-
PRIOR	-	-	-	-	X	-
PROCEDURE	-	-	-	X	-	-
PROCEED	-	-	-	X	-	-
PROCESSING	-	X	-	-	-	-
PROCESSOR	-	-	-	-	-	X
PROGRAM-ID	X	-	-	-	-	-
PT-PUNCH	-	X	-	-	-	-
PT-READER	-	X	-	-	-	-
PUNCH	-	X	-	X	-	-
PURGE	-	-	-	X	-	-
Q-EMPTY	-	-	-	X	-	-
Q-FULL	-	-	-	X	-	-
Q-MAX-MESSAGES	-	-	X	-	-	-
QUEUE	-	X	X	-	-	-
QUEUES	-	X	X	-	-	-
QUOTE	-	-	X	X	-	-
QUOTES	-	-	X	X	-	-
RANDOM	-	X	-	-	-	-
RD	-	-	-	-	-	X
READ	-	-	-	X	-	-
READER	-	X	-	-	-	-
READER-SORTER	-	X	-	X	-	-
READ-OK	-	-	-	X	-	-
RECEIVE	-	-	-	-	-	X

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
RECORD	-	X	X	X	-	-
RECORDING	-	-	X	-	-	-
RECORDS	-	-	X	-	-	-
RECREATE	-	-	-	-	X	-
REDEFINES	-	-	X	-	-	-
REEL	-	X	-	X	-	-
RELEASE	-	-	-	X	-	-
REMAINDER	-	-	-	X	-	-
REMARKS	X	-	-	-	-	-
REMOTE	-	X	-	-	-	-
REMOVE	-	-	-	X	-	-
RENAMES	-	-	X	-	-	-
REPLACING	-	-	-	X	-	-
REPORT	-	-	-	-	-	X
REPORTING	-	-	-	-	-	X
REPORTS	-	-	-	-	-	X
RERUN	-	-	-	-	-	X
RESERVE	-	X	-	-	-	-
RESET	-	-	-	-	-	X
RESTART	-	-	-	-	X	-
RETRIEVAL	-	-	-	-	X	-
RETURN	-	-	-	X	-	-
REVERSED	-	-	-	X	-	-
REWIND	-	-	-	X	-	-
REWRITE	-	-	-	-	-	X
RF	-	-	-	-	-	X
RH	-	-	-	-	-	X
RIGHT	-	-	X	-	-	-
ROLLOUT	-	-	-	-	-	X
ROUNDED	-	-	-	X	-	-
RUN	-	-	-	X	-	-
SAME	-	X	-	-	-	-
SAVE	-	X	-	-	-	-
SAVE-FACTOR	-	-	X	-	-	-
SD	-	-	X	-	-	-
SEARCH	-	-	-	X	-	-
SECTION	-	X	X	X	-	-
SECURITY	X	-	-	-	-	-
SEEK	-	-	-	X	-	-
SEGMENT-LIMIT	-	X	-	-	-	-
SELECT	-	X	-	-	-	-
SEND	-	-	-	-	-	X
SENTENCE	-	-	-	X	-	-
SEQUENTIAL	-	X	-	-	-	-
SET	-	-	-	X	-	-
SIGN	-	-	-	-	-	X
SINGLE	-	X	-	-	-	-
SIZE	-	-	-	X	-	-
SORT	-	X	-	X	-	-
SORTER	-	-	-	-	-	X
SOURCE	-	-	-	-	-	X

Appendix A (Cont)

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
SOURCE-COMPUTER	-	X	-	-	-	-
SPACE	-	-	X	X	-	-
SPACES	-	-	X	X	-	-
SPECIAL-NAMES	-	X	-	-	-	-
SPO	-	X	X	-	-	-
SPO-INPUT-PRESENT	-	-	-	X	-	-
STACKER	-	-	-	X	-	-
STACKERS	-	-	-	X	-	-
STALEMATE	-	-	-	-	-	X
STANDARD	-	-	X	-	-	-
START	-	-	-	X	-	-
START-FLOW	-	-	-	-	-	X
STATION	-	-	X	-	-	-
STATIONS	-	-	X	-	-	-
STATUS	-	-	-	-	-	X
STOP	-	-	-	X	-	-
STOP-FLOW	-	-	-	-	-	X
STORE	-	-	-	-	X	-
STREAM	-	-	-	-	-	X
SUB-QUEUE	-	-	-	-	-	X
SUB-QUEUES	-	-	-	-	-	X
SUBSET	-	-	-	-	X	-
SUBTRACT	-	-	-	X	-	-
SUM	-	-	-	-	-	X
SW1	-	-	-	X	-	-
SW2	-	-	-	X	-	-
SW3	-	-	-	X	-	-
SW4	-	-	-	X	-	-
SW5	-	-	-	X	-	-
SW6	-	-	-	X	-	-
SW7	-	-	-	X	-	-
SW8	-	-	-	X	-	-
SY	-	-	X	-	-	-
SYMBOLIC	-	-	-	-	-	X
SYNC	-	-	X	-	-	-
SYNCHRONIZED	-	-	X	-	-	-
TAG-SEARCH	-	-	-	X	-	-
TAG-KEY	-	-	-	X	-	-
TALLY	-	-	-	X	-	-
TALLYING	-	-	-	X	-	-
TAPE	-	X	-	-	-	-
TAPE-MTC1	-	X	-	-	-	-
TAPE-MTC2	-	X	-	-	-	-
TAPE-MTC3	-	X	-	-	-	-
TAPE-MTC4	-	X	-	-	-	-
TAPE-MTC5	-	X	-	-	-	-
TAPE-7	-	X	-	-	-	-
TAPE-9	-	X	-	-	-	-
TAPES	-	X	-	-	-	-
TERMINATE	-	-	-	-	-	X
THAN	-	-	-	X	-	-

RESERVED WORD	REASON FOR RESERVATIONS					
	D I V I S I O N				DMSII	FUTURE/ STANDARDS
	IDEN	ENVI	DATA	PROC		
THEN	-	-	-	X	-	-
THROUGH	-	X	X	X	-	-
THRU	-	X	X	X	-	-
TIME	-	-	-	X	-	-
TIMES	-	-	X	X	-	-
TO	-	X	X	X	-	-
TODAYS-DATE	-	-	-	X	-	-
TODAYS-NAME	-	-	-	X	-	-
TOP	-	-	-	-	-	X
TRACE	-	-	-	X	-	-
TRANSLATION	-	-	-	-	-	X
TYPE	-	-	-	-	-	X
UNIT	-	-	-	-	-	X
UNLOCK	-	-	-	-	-	X
UNTIL	-	-	-	X	-	-
UP	-	-	-	X	-	-
UPDATE	-	-	-	-	X	-
UPON	-	-	-	X	-	-
USAGE	-	-	X	-	-	-
USE	-	-	-	X	-	-
USING	-	-	-	X	-	-
VA	-	-	X	-	-	-
VALUE	-	-	X	-	-	-
VALUES	-	-	X	-	-	-
VARYING	-	-	-	X	-	-
VIA	-	-	-	-	X	-
WAIT	-	-	-	X	-	-
WHEN	-	-	-	X	-	-
WITH	-	-	X	X	-	-
WORDS	-	-	-	-	-	X
WORK	-	X	-	-	-	-
WORKING-STORAGE	-	-	X	-	-	-
WRITE	-	-	-	X	-	-
WRITE-OK	-	-	-	X	-	-
ZERO	-	-	X	X	-	-
ZEROES	-	-	X	X	-	-
ZEROS	-	-	X	X	-	-
ZIP	-	-	-	X	-	-

COBOL SYNTAX SUMMARY

IDENTIFICATION DIVISION

[MONITOR ...]IDENTIFICATION DIVISION.[PROGRAM-ID. Any COBOL word.][AUTHOR. Any entry.][INSTALLATION. Any entry.][DATE-WRITTEN. Any entry.][DATE-COMPILED. Any entry - appended with
current date and time as
maintained by the MCP.][SECURITY. Any entry.][REMARKS. Any entry. Continuation lines must
be coded in Area B of the coding form.]MONITOR**[MONITOR** [DEPENDING] file-name ([data-name] ...]
{[ALL
{procedure-name...}]}].

ENVIRONMENT DIVISION

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

CONFIGURATION SECTION.

Option 1:

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

Option 2:

SOURCE-COMPUTER. {any entry} .

Option 1:

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

Option 2:

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

Option 1:

SPECIAL-NAMES. COPY library-name
[REPLACING word-1 BY word-2
[, word-3 BY word-4] ...].

Option 2:

SPECIAL-NAMES. [CURRENCY SIGN IS literal]
[[, implementor-name IS mnemonic-name] ...]
[, DECIMAL-POINT IS COMMA].

INPUT-OUTPUT SECTION

Option 1:

FILE-CONTROL. COPY library-name
$$\left[\text{REPLACING } \left\{ \begin{array}{l} \text{word-1} \\ \text{data-name-1} \end{array} \right\} \text{ BY } \left\{ \begin{array}{l} \text{word-2} \\ \text{data-name-2} \\ \text{literal-1} \end{array} \right\} \right.$$

$$\left. \left[\left\{ \begin{array}{l} \text{word-3} \\ \text{data-name-3} \end{array} \right\} \text{ BY } \left\{ \begin{array}{l} \text{word-4} \\ \text{data-name-4} \\ \text{literal-2} \end{array} \right\} \right] \dots \right].$$

Option 2:

FILE-CONTROL.SELECT [OPTIONAL] file-name ASSIGN TO hardware-name-1
$$\left[\text{BY } \left[\text{integer-1 } \left[\text{INC-EU} \right] \right] \left[\text{WORK} \right] \left[\text{ALL-AT-OPEN} \right] \left[\text{SINGLE} \right] \left[\text{SAVE} \right] \right.$$

$$\left[\left\{ \begin{array}{l} \text{NO} \\ \text{OR} \end{array} \right\} \text{ BACKUP } \left\{ \begin{array}{l} \text{TAPE} \\ \text{DISK} \end{array} \right\} \right] \left[\text{FORM} \right] \left[\text{FOR MULTIPLE REEL} \right]$$

$$\left[\text{RESERVE } \left\{ \begin{array}{l} \text{NO} \\ \text{integer-2} \end{array} \right\} \left[\text{ALTERNATE } \left\{ \begin{array}{l} \text{AREA} \\ \text{AREAS} \end{array} \right\} \right] \right]$$

$$\left[\left\{ \begin{array}{l} \text{FILE-LIMIT IS} \\ \text{FILE-LIMITS ARE} \end{array} \right\} \left\{ \begin{array}{l} \text{literal-1} \\ \text{data-name-1} \end{array} \right\} \left\{ \begin{array}{l} \text{THRU} \\ \text{THROUGH} \end{array} \right\} \left\{ \begin{array}{l} \text{END} \\ \text{literal-2} \\ \text{data-name-2} \end{array} \right\} \right]$$

$$\left[\left\{ \begin{array}{l} \text{literal-m} \\ \text{data-name-m} \end{array} \right\} \left\{ \begin{array}{l} \text{THRU} \\ \text{THROUGH} \end{array} \right\} \left\{ \begin{array}{l} \text{literal-n} \\ \text{data-name-n} \end{array} \right\} \dots \right]$$

$$\left[\text{ACCESS MODE IS } \left\{ \begin{array}{l} \text{RANDOM} \\ \text{SEQUENTIAL} \end{array} \right\} \right]$$

$$\left[\text{ACTUAL KEY IS data-name-3} \right]$$

$$\left[\text{PROCESSING MODE IS SEQUENTIAL} \right] . \left[\text{SELECT} \right] \dots$$

Option 3:

FILE-CONTROL

$$\text{SELECT sort-file-name ASSIGN TO } \left[\begin{array}{l} \text{integer-3} \\ \text{data-name-4} \end{array} \right] \text{ SORT } \left\{ \begin{array}{l} \text{TAPES} \\ \text{TAPE} \\ \text{DISK} \end{array} \right\}$$

I-O-CONTROL.

Option 1:

I-O-CONTROL. COPY library-name

[REPLACING word-1 BY word-2
[, word-3 BY word-4] ...].

Option 2:

I-O-CONTROL.

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

DATA DIVISION

DATA DIVISION.

[FILE SECTION.

[{file-description-entry } [record-description-entry] ...] ...]
[{sort-merge-description-entry }

[WORKING-STORAGE SECTION.

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

FILE SECTION.

Option 1:

FD file-name COPY library-name

[REPLACING { word-1
data-name-1 } BY { word-2
data-name-2
literal-1 }
[, { word-3
data-name-3 } BY { word-4
data-name-4
literal-2 }] ...] .

Option 2

FD file-name-1

; RECORDING MODE IS

{ ASCII
STANDARD
NON-STANDARD }

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

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

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

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

[SAVE-FACTOR IS integer-7]

[; { VA
VALUE } OF Q MAX MESSAGES IS integer-8]

[; LABEL { RECORD IS
RECORDS ARE } { OMITTED
STANDARD } [data-name-2 [, data-name-3 . . .]]]

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

Option 3:

SD sort-file-name COPY library-name

[REPLACING {word-1
data-name-1} BY {word-2
data-name-2
literal-1}
[, {word-3
data-name-3} BY {word-4
data-name-4
literal-2}] ...] .

Option 4:

SD sort-file-name

[: FILE CONTAINS integer-1 [BY integer-2] RECORDS
[; BLOCK CONTAINS [integer-3 TO] integer-4 [RECORDS
CHARACTERS]
[; RECORD CONTAINS [integer-5 TO] integer-6 CHARACTERS]
[; DATA { RECORD IS
RECORDS ARE } data-name-1 [data-name-2] ...]

Option 1:

O1 data-name-1; COPY library-name

[REPLACING {word-1
data-name-2} BY {word-2
data-name-3
literal-1}
[, {word-3
data-name-4} BY {word-4
data-name-5
literal-2}] ...] .

Option 2:

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

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

[; { BZ
BLANK WHEN ZERO }]

[; { JS
JUST
JUSTIFIED } RIGHT]

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

[; { OC
OCCURS } [integer-1 TO] integer-2 TIMES

[DEPENDING ON data-name-3]

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

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

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

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

Appendix B (Cont)

Option 3:

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

Option 4:

88 condition-name { VALUE } [IS
ARE] literal-1 [{ THRU
THROUGH } literal-2] .
[, literal-3 [{ THRU
THROUGH } literal-4]]

PROCEDURE DIVISION

PROCEDURE DIVISION.

[DECLARATIVES.

section-name SECTION. declarative-statement.

paragraph-name. [statement.] ...

[paragraph-name. [statement.] ...] ...

[section-name SECTION. declarative-statement.

paragraph-name. [statement.] ...

[paragraph-name. [statement.] ...] ...] ...

END DECLARATIVES.]

[[section-name SECTION [priority-number]] .

paragraph-name. [statement.] ...

[[paragraph-name.] ... [statement.] ...] ...] ...

END-OF-JOB.]

Verb Formats:

ACCEPT identifier [FROM { SPO
mnemonic-name }]

Option 1:

```

ADD {literal-1 } [{literal-2 } ...]
      TO identifier-m [ROUNDED] [identifier-n [ROUNDED] ...]
      [; ON SIZE ERROR statement-1 [; ELSE statement-2] ]

```

Option 2:

```

ADD {literal-1 } {literal-2 } [{literal-3 } ...]
      {identifier-1 } {identifier-2 } [{identifier-3 } ...]
      GIVING identifier-m [ROUNDED] [, identifier-n [ROUNDED] ...]
      [; ON SIZE ERROR statement-1 [; ELSE statement-2] ]

```

Option 3:

```

ADD {CORR } {CORRESPONDING } identifier-1 TO identifier-2
      [ROUNDED] [; ON SIZE ERROR statement-1 [; ELSE statement-2] ]

```

```

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
      [, procedure-name-3 TO [PROCEED TO] procedure-name-4 ...]

```

```

CLOSE file-name-1 [REEL] [ WITH { LOCK
                                     { PURGE
                                     { RELEASE
                                     { NO REWIND
                                     { REMOVE
                                     } } } } [WITH CRUNCH] [,file-name-2...] ]

```

```

COMPUTE identifier-1 [ROUNDED] [, identifier-n [ROUNDED] ...]
      = { identifier-2
        { literal-1
        { arithmetic-expression }
      }
      [; ON SIZE ERROR statement-1 [; ELSE statement-2] ]

```

Option 1:

CONTROL file-name [STOP-FLOW] POCKET {literal-1
identifier-1}

Option 2:

CONTROL file-name { BATCH-COUNT
POCKET-LIGHT {literal-2
identifier-2} }

Option 1:

COPY library-name.

Option 2:

COPY library-name

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

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

DISPLAY {literal-1
identifier-1} [{literal-2
identifier-2} ...]

[UPON {SPO
mnemonic-name}]

Option 1:

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

[; ON SIZE ERROR statement-1 [; ELSE statement-2]]

Option 2:

DIVIDE [MOD] {literal-1
identifier-1} {BY} {literal-2
INTO} {identifier-2}

GIVING identifier-3 [ROUNDED]

[REMAINDER identifier-5 [ROUNDED]]

[; ON SIZE ERROR statement-1 [; ELSE statement-2]]

DUMP [list]

Option 1:

EXAMINE identifier -1 TALLYING { ALL
LEADING
 [UNTIL] FIRST } { literal-1
 identifier-2 }
 [REPLACING BY { literal-2
 identifier-3 }]

Option 2:

EXAMINE identifier-1 REPLACING { ALL
LEADING
 [UNTIL] FIRST } { literal-1
 identifier-2 }
BY { literal-2
 identifier-3 }

EXIT.

FORMAT identifier-1 INTO identifier-2 [ON SIZE ERROR statement-1
 [ELSE statement-2]]

Option 1:

GO TO [procedure-name]

Option 2:

GO TO procedure-name-1 [, procedure-name-2] ...
 , procedure-name-n DEPENDING ON identifier

IF condition-1; { sentence-1
NEXT SENTENCE } [; ELSE { sentence-2
NEXT SENTENCE }]

Option 1:

MERGE file-name-1 [{ PURGE
END } ON ERROR]
ON { ASCENDING
DESCENDING } KEY data-name-1 [data-name-2] ...
 [ON { ASCENDING
DESCENDING } KEY data-name-3 [data-name-4] ...] ...
 [WITH file-name-6]
USING file-name-2 file-name-3 [file-name-4] ...
 { OUTPUT PROCEDURE IS procedure-name-1 [{ THROUGH
THRU } procedure-name-2] }
 { GIVING file-name-5 [LOCK
RELEASE] }

Appendix B (Cont)

~~MOVE~~ ~~EDIT~~ identifier-1 ~~INTO~~ identifier-2

Option 1:

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

Option 2:

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

MULTIPLY { literal-1
 { identifier-1 } } BY { literal-2
 { identifier-2 }

[GIVING identifier-3] [ROUNDED]

[; ON SIZE ERROR statement-1 [; ELSE statement-2]

Option 1 Paragraph NOTE:

Paragraph-name. NOTE any comment.

Option 2 Paragraph NOTE:

NOTE. Any comment.

Option 3 Sentence NOTE:

NOTE. Any comment.

Option 1:

OPEN

{ [INPUT file-name-1 [{ WITH LOCK [ACCESS]
 { REVERSED
 { WITH NO REWIND } }] [file-name-2...]] ... }
[OUTPUT file-name-3 [WITH NO REWIND] [file-name-4...]] ... } ...
[{ INPUT-OUTPUT } file-name-5 [LOCK [ACCESS]] [file-name-6 ...]
 { I-O }] ...
[O-I file-name-7 [file-name-8...]] ...
[EXTEND file-name-9 [WITH LOCK [ACCESS]] [file-name-10 ...]] ... }

Option 2:

OPEN { OUTPUT
 { I-O
 { INPUT-OUTPUT } } file-name

[WITH PUNCH] [WITH { PRINT128
 { INTERPRET } }] [WITH STACKERS]

Option 1:

PERFORM procedure-name-1 [{ THRU
 { THROUGH } } procedure-name-2]

Appendix B (Cont)

Option 2:

PERFORM procedure-name-1 [{ THRU
THROUGH } procedure-name-2]

{ integer-1
{ identifier-10 } TIMES

Option 3:

PERFORM procedure-name-1 [{ THRU
THROUGH } procedure-name-2]

VARYING { index-name-1 }
{ identifier-1 } FROM { index-name-2 }
{ identifier-2 } BY
{ literal-2 }

{ identifier-3 }
{ literal-3 } UNTIL condition-1 [AFTER { index-name-4 }
{ identifier-4 }

FROM { index-name-5 }
{ identifier-5 } BY { index-name-6 }
{ literal-6 }

UNTIL condition-2] [AFTER { index-name-7 }
{ identifier-7 } FROM

{ index-name-8 }
{ identifier-8 } BY { identifier-9 }
{ literal-8 } UNTIL condition-3]

Option 1:

READ file-name [INTO identifier]

Option 2:

READ file-name RECORD [INTO identifier]

[; { AT END
{ INVALID KEY } statement-1 [; ELSE statement-2]]

RELEASE record-name [FROM identifier]

RETURN file-name RECORD [INTO identifier]

[; AT END statement-1 [; ELSE statement-2]]

Option 1:

SEARCH identifier-1 [VARYING {index-name-1}
{identifier-2}]

[; AT END imperative-statement-1]

; WHEN condition-1 {statement-1}
{NEXT SENTENCE}

[; WHEN condition-2 {statement-2}
{NEXT SENTENCE}]

Option 2:

SEARCH ALL identifier-3 [; AT END imperative-statement-2]

; WHEN condition-3 {statement-3}
{NEXT SENTENCE}

SEEK file-name RECORD [WITH KEY CONVERSION]

Option 1:

SET {index-name-1}
{identifier-1} [, {index-name-2}
{identifier-2} ...] TO {index-name-3}
{identifier-3}
{literal-1}

Option 2:

SET index-name-4 [, index-name-5 ...] {UP BY} {identifier-4}
{DOWN BY} {literal-2}

SORT TAG-SEARCH TAG-KEY INPLACE file-name-1

[WITH file-name-4] [RESTART WITH data-name-5]

[PURGE END] ON ERROR]

ON { DESCENDING ASCENDING } KEY data-name-1 [, data-name-2] ...

[ON { DESCENDING ASCENDING } KEY data-name-3 [, data-name-4] ...] ...

[INPUT PROCEDURE IS section-name-1 [{ THRU THROUGH } section-name-2]]

[USING file-name-2 [LOCK PURGE RELEASE]]

[OUTPUT PROCEDURE IS section-name-3 [{ THRU THROUGH } section-name-4]]

[GIVING file-name-3 [LOCK RELEASE]]

STOP { RUN literal }

Option 1:

SUBTRACT { literal-1 identifier-1 } [, { literal-2 identifier-2 } ...] FROM
identifier-m [ROUNDED] [, identifier-n [ROUNDED] ...]
[; ON SIZE ERROR statement-1 [; ELSE statement-2]]

Option 2:

SUBTRACT { literal-1 identifier-1 } [, { literal-2 identifier-2 } ...] FROM { literal-m identifier-m }
GIVING identifier-n [ROUNDED] [, identifier-o [ROUNDED]] ...
[; ON SIZE ERROR statement-1 [; ELSE statement-2]]

Option 3:

SUBTRACT { CORR
 { CORRESPONDING } identifier-1 FROM identifier-2

[ROUNDED] [; ON SIZE ERROR statement-1 [; ELSE statement-2]

TRACE 20

Option 1:

USE AFTER STANDARD ERROR PROCEDURE ON { file-name-1 [, file-name-2] ...
 { INPUT
 { INPUT-OUTPUT
 { I-O } }

Option 2:

USE FOR KEY CONVERSION ON file-name-1 [, file-name-2] ...

Option 3:

USE FOR Q-EMPTY ON file-name-1 [file-name-2]... .

Option 4:

USE FOR Q-FULL ON file-name-1 [file-name-2]... .

Option 5:

USE FOR READER-SORTER POCKET file-name.

Option 1:

WAIT UNTIL

{ identifier-1
literal-1 }

[SPO-INPUT-PRESENT]

[WRITE-OK ON file-name-1 { { identifier-2 }
{ literal-2 } } ...]

[READ-OK ON file-name-2 { { identifier-3 }
{ literal-3 } } ...]

[USING identifier-4]

[GIVING identifier-5]

Option 1:

WRITE record-name [FROM identifier-1]

[{ AFTER
BEFORE } ADVANCING { { integer-1
identifier-2 } LINES
TO CHANNEL { integer-2
identifier-3 } }]

[; AT { END-OF-PAGE
EOP } imperative-statement [ELSE STATEMENT-1]]

[TO { ERROR
AUXILIARY
STACKER { integer-3
identifier-4 } }]

Option 2:

WRITE record-name [FROM identifier-1]

[; INVALID KEY statement-1 [; ELSE statement-2]]

ZIP date-name

COMPILER ERROR MESSAGES

<u>ERROR NO.</u>	<u>MESSAGE</u>
000	FILE-NAME EXPECTED
001	INTEGER LITERAL REQUIRED
002	INVALID LITERAL
003	RESERVED WORD REQUIRED
004	PARAGRAPH HEADER EXPECTED IN AREA A (COL. 8-11)
005	MISSING DIVISION
006	DOLLAR CARD ERROR
007	"DIVISION" REQUIRED
008	COMPILER ERROR
009	MISSING PERIOD
010	RESERVED WORD OR DATA NAME REQUIRED
011	COPY REPLACING OR MNEMONIC LIST OVERFLOW
012	DUPLICATE MNEMONIC NAME
013	ILLEGAL WORD
014	IMPROPER LABEL RECORD(S) DECLARATION
015	ILLEGAL NESTED COPY
016	ILLEGAL COPY OPERAND
017	STANDARD OR NON-STANDARD OR ASCII REQUIRED
018	DUPLICATE REPLACING
019	ILLEGAL SUBSCRIPTING
020	ILLEGAL LIBRARY NAME
021	ILLEGAL TYPE
022	ILLEGAL QUALIFICATION
023	ILLEGAL PROGRAM ID
024	"SECTION" REQUIRED
025	MISSING FILE NAME
026	A PARENTHESIS WAS EXPECTED HERE
027	MISSING LABEL QUALIFICATION...MONITOR
028	NO FD OR SD
029	INVALID FD
030	ILLEGAL LEVEL
031	ILLEGAL DATA NAME
032	RELATIONAL OPERATOR REQUIRED
033	PICTURE SIZE ERROR
034	"PROCEDURE" EXPECTED
035	ILLEGAL FD OR SD IN WORKING-STORAGE
036	PRIORITY NUMBER ERROR
037	MISSING IMPLIED LABEL OR LABEL QUALIFICATION
038	MISSING SECTION
039	NO "USE"
040	PARAGRAPH-NAME OR SECTION-NAME REQUIRED
041	VERB OR PARAGRAPH-NAME OR SECTION-NAME REQUIRED
042	MISSING FILE NAME
043	ILLEGAL LABEL RECORD REFERENCE OUTSIDE DECLARATIVES
044	ILLEGAL ARITHMETIC OPERAND
045	MISSING "=" OR "FROM"
046	NO VALID CORRESPONDING OPERANDS

Appendix C (Cont)

<u>ERROR NO.</u>	<u>MESSAGE</u>
047	COMPOSITE ARITHMETIC SIZE > 125...MAY USE LARGE AMOUNT OF CORE
048	MISSING "END DECLARATIVES"
049	MISSING "DECLARATIVES"
050	ILLEGAL MOVE OPERAND
051	"TO" REQUIRED
052	AN ALPHABETIC ITEM CANNOT BE MOVED TO A NUMERIC ITEM
053	ILLEGAL GROUP TO ELEMENTARY MOVE
054	ILLEGAL "ALL" LITERAL
055	ILLEGAL SUBSCRIPTING OF A SUBSCRIPT
056	SUBSCRIPT NOT S-SIGN OR UNSIGNED
057	SUBSCRIPT NOT NUMERIC INTEGER
058	SUBSCRIPT NOT ELEMENTARY ITEM
059	ILLEGAL MIXING OF INDEX AND SUBSCRIPT
060	EXPLICIT DATA NAME TABLE OVERFLOW
061	THIS DATA NAME IS NOT DESCRIBED IN THE DATA DIVISION
062	QUALIFIER ARRAY TABLE OVERFLOW
063	ILLEGAL QUALIFIER
064	INSUFFICIENT QUALIFICATION
065	OVERLAPPING CORRESPONDING OPERANDS
066	NO MATCHING CORRESPONDING OPERANDS
067	CORRESPONDING NAMES ARE THE SAME
068	FD NAME ILLEGAL FOR CORRESPONDING
069	CORRESPONDING DATA NAME NOT GROUP ITEM
070	DUPLICATE PARAGRAPH OR SECTION NAME
071	LABEL NOT UNIQUE
072	LABEL QUALIFICATION NOT A SECTION
073	ALTER TABLE OVERFLOW
074	QUALIFIER LABEL TABLE OVERFLOW
075	REFERENCED PARAGRAPH OR SECTION DOES NOT EXIST
076	LABEL QUALIFIER IS NOT UNIQUE
077	LABEL RECORD IS NOT AN 01 LEVEL
078	ILLEGAL CONDITIONAL STATEMENT
079	ILLEGAL DOUBLE NEGATIVE
080	INVALID IMPLIED SUBJECT OR MISSING RELATIONAL OPERATOR
081	PICTURE TABLE FULL : RECOMPILE
082	PICTURE SPECIFIED ON A GROUP ITEM
083	RENAMES OPERAND OUT OF RANGE
084	RENAMES OPERAND LEVEL CANNOT BE 01 OR 66 OR 77 OR 88
085	RENAMES OPERAND IS SUBSCRIPTED
086	DUPLICATE NAME
087	"RENAMES" REQUIRED
088	GROUP RENAMES ITEM ADDRESS OR LENGTH NOT 0 MOD 2
089	BLANK WHEN ZERO SPECIFIED FOR NON NUMERIC CLASS
090	JUSTIFIED SPECIFIED FOR NUMERIC OR EDITED NUMERIC CLASS
091	UNSIGNED INTEGER EXPECTED
092	"OCCURS" SPECIFIED FOR LEVEL 01 OR 77
093	VARIABLE LENGTH DISK FILE MUST HAVE SEQUENTIAL ACCESS AND NO FILE LIMITS
094	NON-ZERO VALUE EXPECTED
095	DUPLICATE "VALUE" CLAUSE
096	ILLEGAL "VALUE" LITERAL
097	DATA CLAUSE EXPECTED
098	ILLEGAL DATA CLAUSE FOR GROUP ITEM
099	ILLEGAL 4 BIT SPECIFICATION FOR HARDWARE DEVICE
100	"ZERO" EXPECTED

<u>ERROR NO.</u>	<u>MESSAGE</u>
101	ASCII MAY BE SPECIFIED ON ONLY WORKING-STORAGE LEVEL 01 OR 77
102	MISSING "OCCURS" FOR INDEX-NAME
103	INDEX NAME EXPECTED
104	LEVEL NUMBER EXPECTED
105	LEVEL NOT 01 THRU 49, 66, 77, OR 88
106	LEVEL 77 MUST FOLLOW ONLY WORKING-STORAGE SECTION
107	PICTURE REQUIRED FOR ELEMENTARY DATA NAME
108	NO DATA CLAUSE FOR INDEX DATA ITEM
109	COMPUTATIONAL ITEM NOT NUMERIC
110	EXPECTED A COLON IN MONITOR STATEMENT
111	IMPROPER REDEFINED NAME
112	LEVEL NUMBER NEQ REDEFINED LEVEL NUMBER
113	REDEFINED OPERAND IS SUBSCRIPTED
114	REDEFINED GROUP ADDRESS IS ODD
115	VALUE CANNOT BE SPECIFIED FOR SUBSCRIPTED ITEM
116	VALUE CANNOT BE SPECIFIED FOR REDEFINED AREA
117	VALUE CONFLICTS WITH GROUP VALUE
118	FILLER ADDED TO PREVIOUS ITEM
119	REDEFINED AREA NEQ REDEFINING AREA
120	USAGE CONFLICTS WITH GROUP USAGE
121	SUBSCRIPT MAXIMUM IS 3
122	INCONSISTENT LEVEL NUMBER
123	01 LEVEL NUMBER EXPECTED
124	"SELECT" EXPECTED
125	FILE PREVIOUSLY SELECTED
126	FILE NOT SELECTED
127	FILE INFO TABLE FULL
128	HARDWARE NAME EXPECTED
129	SD FILE NOT ASSIGNED TO DISK OR TAPE
130	ILLEGAL CHARACTER
131	WORD EXCEEDS 30 CHARACTERS
132	INVALID NUMERIC OR UNDIGIT LITERAL
133	ZERO SIZE LITERAL
134	MISSING RIGHT QUOTE
135	WORD ENDS IN A HYPHEN
136	NO ALPHA CHARACTER IN NAME
137	MISSING LEFT QUOTE
138	TABLE OVERFLOW IN MERGE: RECOMPILE
139	MISSING "BY" OR "INTO"
140	MISSING "GIVING"
141	"MOD" AND "REMAINDER" ARE MUTUALLY EXCLUSIVE
142	MISSING "ERROR"
143	ILLEGAL PERFORM OPERAND
144	ILLEGAL PERFORM "TIMES" OPERAND OR MISSING "TIMES"
145	MISSING "UNTIL"
146	ILLEGAL PERFORM "VARYING" OPERAND
147	ILLEGAL PERFORM "FROM" OPERAND OR MISSING "FROM"
148	ILLEGAL PERFORM "BY" OPERAND OR MISSING "BY"
149	ILLEGAL SET OPERAND
150	FILE-LIMITS SPECIFICATION GIVEN FOR SD FILE
151	INVALID DESCRIPTION FOR ACTUAL KEY
152	PICTURE REPEAT ERROR
153	PICTURE FLOAT ERROR
154	PICTURE SIGN ERROR
155	PICTURE "P" SPECIFICATION ERROR

Appendix C (Cont)

<u>ERROR NO.</u>	<u>MESSAGE</u>
156	PICTURE SIZE SPECIFICATION ERROR
157	PICTURE DECIMAL POINT ERROR
158	PICTURE ERROR...IMPROPER CHARACTER PRECEDING FLOAT,ZERO SUPPRESS,OR CHECK PROTECT
159	PICTURE CLASS ERROR
160	PICTURE MASK SIZE (100) EXCEEDED
161	VALUE OF ID CATEGORY IS NUMERIC
162	FILE-LIMIT MUST BE PC 9(8) COMP
163	"THRU" EXPECTED
164	DATA NAME OR INTEGER EXPECTED
165	"RANDOM" OR "SEQUENTIAL" EXPECTED
166	SD FILE MUST BE SEQUENTIAL
167	ACTUAL KEY REQUIRED
168	"BACKUP" EXPECTED
169	FILE-CONTROL CLAUSE EXPECTED
170	BACKUP FOR LINE PRINTER OR PUNCH ONLY
171	ILLEGAL USE OF FILE-NAME OR CONDITION-NAME
172	"MULTI-FILE-ID" EXPECTED
173	FILE NOT ASSIGNED TO TAPE OR DISK
174	I-O CONTROL CLAUSE OR "." EXPECTED
175	APPLY CLAUSE NOT IMPLEMENTED
176	FILE NOT ASSIGNED TO HARDWARE DEVICE
177	DECLARATIVES NOT 1ST ITEM IN PRO.DIV. OR USE NOT BETWEEN SECTION & PARAGRAPH
178	CONDITION-NAME LITERAL REQUIRED
179	"VALUE" REQUIRED
180	VALUE THRU...1ST LITERAL GEQ 2ND LITERAL
181	I/O OPERAND MUST BE 01 RECORD OF A FILE
182	I/O OPERAND CANNOT BE LABEL RECORD
183	I/O OPERAND MUST BE SORT FILE
184	I/O OPERAND MUST BE FILE
185	I/O OPERAND CANNOT BE WORKING-STORAGE 01 RECORD
186	I/O OPERAND CANNOT BE SORT FILE
187	CAN MONITOR ONLY ON FILE
188	SORT KEY NOT WITHIN SCOPE OF SORT FILE
189	SORT STATEMENT NOT PERMITTED IN DECLARATIVES SECTION
190	MISSING "ASCENDING" OR "DESCENDING"
191	SORT KEY CANNOT BE SUBSCRIPTED
192	MISSING SORT KEY
193	NUMBER OF SORT KEYS GREATER THAN 40
194	SIZE OF KEY TOO LARGE
195	MISSING "INPUT"/"OUTPUT"/"USING"/"GIVING"
196	USING/GIVING FILE REC SIZE NEQ SORT REC SIZE
197	MISSING SORT INPUT/OUTPUT PROCEDURE NAME
198	SORT THRU PROCEDURE APPEARS BEFORE BEGINNING POINT
199	GO TO DEPENDING OPERAND MUST BE ELEMENTARY DATA-NAME INTEGER
200	MORE THAN 1 ACCEPT OPERAND
201	MISSING WORD OR LITERAL
202	NOT READABLE/WRITEABLE HARDWARE
203	GO TO DEPENDING LABEL LIMIT (1022) EXCEEDED
204	FILE-LIMITS REQUIRE "AT END" OR "INVALID KEY"
205	ILLEGAL READER-SORTER OPERAND
206	MISSING "DEPENDING" IN GO TO
207	READ/WRITE/SEEK ON SD CR RELEASE/RETURN ON FD
208	SEEK FILE NOT RANDOM DISK
209	ILLEGAL ADVANCING/STACKER OPERAND
210	INVALID I/O OPERAND

<u>ERROR NO.</u>	<u>MESSAGE</u>
211	MINUS SIGN NOT ALLOWED
212	MISSING GO TO LABEL
213	GO TO MUST BE TERMINATED BY "." OR "ELSE"
214	ILLEGAL OPTION FOR I/O DEVICE
215	INVALID OR MISSING OPEN TYPE
216	ATTEMPTED ALTER OF NON-GOTO PROCEDURE-NAME
217	CHANNEL NUMBER GTR 11
218	RESERVED WORD (VERB) REQUIRED
219	CURRENT SECTION MUST HAVE SAME PRIORITY AS REFERENCED GOTO PROC
220	TAPE FILE CANNOT HAVE 2 NAMES
221	EXPECTED A FILE DECLARATION CLAUSE
222	MISSING FILE CONTAINS CLAUSE - 1 AREA OF 500 BLOCKS BY DEFAULT
223	RECORDS PER AREA MADE MULTIPLE OF RECORDS PER BLOCK
224	DATA DICTIONARY FULL : RECOMPILE
225	REDEFINES NOT ALLOWED ON 01 RECORD OF FILE
226	MISSING FILE RECORD DESCRIPTION
227	BLOCK SIZE NOT MULTIPLE OF MAXIMUM RECORD SIZE
228	ARITHMETIC OPERAND MUST BE ELEMENTARY ITEM
229	ARITHMETIC OPERAND CANNOT BE INDEX ITEM
230	ARITHMETIC OPERAND MUST BE NUMERIC
231	ARITHMETIC LITERAL OPERAND MUST BE NUMERIC
232	FILE LABEL RECORDS OMITTED
233	DUPLICATE USE PROCEDURE
234	MONITOR ALLOWED ONLY ON FILE
235	CANNOT MONITOR ON SORT FILE
236	MONITOR ALLOWED ON LINE PRINTER ONLY
237	OBJECT OF SEARCH MUST BE INDEX
238	MISSING ALTER LABEL
239	DATA-NAME REQUIRED
240	DATA-NAME OR INDEX-NAME REQUIRED
241	WHEN CLAUSE REQUIRED
242	ILLEGAL USE OF RESERVED WORD
243	ITEM NOT DISPLAY
244	DATA LENGTH EXCEEDS 1 CHARACTER
245	ILLEGAL USE OF FIRST IN EXAMINE
246	MISSING FIRST IN EXAMINE
247	SIGN CONDITION OPERAND MUST BE ELEMENTARY NUMERIC OR ARITHMETIC EXPRESSION
248	CLASS TEST OPERAND CANNOT BE ARITHMETIC EXPRESSION
249	CLASS TEST OPERAND MUST BE DISPLAY
250	NUMERIC VERSUS ALPHA COMPARE IS ILLEGAL
251	VIOLATION OF ANSI RULES FOR PERFORMING OVERLAYABLE SEGMENTS
252	RECEIVING FIELD TRUNCATION
253	FIELD TREATED AS 8 BIT DISPLAY
254	SEQUENCE ERROR
255	LITERAL EXCEEDS 160 CHARACTERS
256	CLASS TEST OPERAND CANNOT BE ASCII
257	MISSING SUBSCRIPT
258	MISUSE OF CONDITION-NAME
259	COMPARISON OF INDEX DATA ITEM MUST BE AGAINST INDEX DATA ITEM OR INDEX NAME
260	COMPARISON OPERANDS MUST HAVE SAME USAGE
261	ILLEGAL USE OF "NEXT SENTENCE"
262	CANNOT COMPARE LITERALS
263	CANNOT COMPARE INDEX-NAME VS ZERO OR - LITERAL
264	SORTER FILE RECORD NOT MOD 112 IN LENGTH

Appendix C (Cont)

<u>ERROR NO.</u>	<u>MESSAGE</u>
265	CANNOT BLOCK SORTER FILE
266	GROUP NAME CANNOT BE "FILLER"
267	IF STATEMENT MUST BE TERMINATED BY "." OR "WHEN"
268	LITERAL SUBSCRIPT CAUSES OUT OF BOUNDS ERROR
269	HARDWARE MUST BE READER-SORTER
270	READER-SORTER ACTUAL KEY NOT IN DATA SEGMENT ZERO
271	RECORDING MODE BINARY NOT ALLOWED
272	MISSING MONITOR DECLARATION (TO DECLARE DUMP FILE)
273	MULTI RECEIVING FIELDS ILLEGAL WITH CORRESPONDING OPTION
274	TAG-KEY GIVING FILE MUST HAVE 01 RECORD OF PC 9(8) CMP
275	TAG-KEY SORT REQUIRES GIVING FILE, RECORD OF PC 9(8) CMP
276	COMPILER ERROR IN CODEGEN GET [(<-&\$*);-/,%=]#@:>+&\$**<*>\$
277	COMPILER ERROR IN CODGEN GET.TRASH [(<-&\$*);-/,%=]#@:>+&\$**& \$**<*>\$
278	COMPILER ERROR IN CODEGEN GET.POOL [(<-&\$*);-/,%=]#@:>+&\$** &\$**<*>\$
279	COMPILER ERROR IN CODEGEN CONTROL [(<-&\$*);-/,%=]#@:>+&\$** &\$**<*>\$
280	COMPILER ERROR IN CODEGEN ARITH.EXP [(<-&\$*);-/,%=]#@:>+& **&\$**<*>\$
281	COMPILER ERROR IN CODEGEN OPND.OVER [(<-&\$*);-/,%=]#@:>+& **&\$**<*>\$
282	COMPILER ERROR IN PROSYN GET [(<-&\$*);-/,%=]#@:>+&\$**&\$**<*>\$
283	USAGE DECLARED FOR AN ASCII FILE
284	BLOCK SIZE MADE EQUAL TO MAXIMUM RECORD SIZE
285	CURRENT COMPILER DATA SEGMENT LIMIT OF 000 EXCEEDED.. TEMPORARY SOLUTION=RESEGMENT
286	COMPILER ERROR..COLUMN TOO LARGE..DOES NOT AFFECT COMPILATION [(<-&\$*);-/,%=]#@:>+&\$**&\$**<*>\$
287	A ROUTINE IN CODEGEN HAS ENCOUNTERED AN UNEXPECTED TOKEN.. COMPILER ERROR [(<-&\$*);-/,%=]#@:>+&\$**&\$**<*>\$
288	CANNOT COMPARE ALPHA VS REAL
289	MINIMUM OF 2 OPERANDS MUST PRECEDE THE WORD GIVING IN AN ADD
290	SUBSCRIPT NOT INTEGER (SPACE REQUIRED BEFORE LIT IF DECIMAL- POINT IS COMMA
291	INVOKED DATASET NAME IS NOT A WORD
292	EXPECTED "DB" AS A LEVEL INDICATOR
293	INVALID DB NAME
294	EXPECTED A DDL-NUMBER
295	DATA-BASE DECLARATION NOT EXPECTED IN THIS SECTION
296	"DATA SET" EXPECTED
297	WORKING-STORAGE SECTION OR PROCEDURE DIVISION EXPECTED
298	DATA MANAGEMENT LEVEL 01 DATASET NAME REQUIRED
299	DATA BASE NAME REQUIRED
300	DATA MANAGEMENT DATASET NAME REQUIRED
301	SPECIFIED DATASET IS NOT THE TARGET SET OF THIS SUBSET
302	DATA MANAGEMENT SUBSET NAME REQUIRED
303	DATA MANAGEMENT PATH NAME REQUIRED
304	TARGET DATASET NAME MUST BE SPECIFIED WHEN PATHNAME IS A SUBSET
305	SPECIFIED DATASET IS NOT THE TARGET SET OF THIS PATH NAME
306	COMPONENT NAME LIST MAY NOT BE USED WHEN PATHNAME IS A DATASET
307	FIRST, LAST, AND PRIOR MAY NOT BE USED WITH RANDOM RETRIEVAL
308	IMPROPER COMPONENT NAME FOR PATH NAME SPECIFIED
309	INCORRECT NUMBER OF COMPONENT NAMES FOR PATH NAME SPECIFIED
310	EQUAL SIGN MUST FOLLOW A COMPONENT NAME
311	ONLY A DATA NAME OR LITERAL IS ALLOWED
312	FROM OR INTO REQUIRED HERE

<u>ERROR NO.</u>	<u>MESSAGE</u>
313	"INVOKE" EXPECTED
314	PARAGRAPH-NAME REQUIRED FOLLOWING SECTION-NAME
315	COMPILER ERROR IN CODEGEN...CODE.DISP TOO LARGE
316	AN ALPHABETIC ITEM CANNOT BE MOVED TO A NUMERIC EDITED ITEM
317	AN ALPHANUMERIC EDITED ITEM CANNOT BE MOVED TO A NUMERIC ITEM
318	AN ALPHANUMERIC EDITED ITEM CANNOT BE MOVED TO A NUMERIC EDITED ITEM
319	A NUMERIC ITEM CANNOT BE MOVED TO AN ALPHABETIC ITEM
320	A NON-INTEGER NUMERIC ITEM CANNOT BE MOVED TO AN ALPHANUMERIC ITEM
321	A NON-INTEGER NUMERIC ITEM CANNOT BE MOVED TO AN ALPHANUMERIC EDITED ITEM
322	A NUMERIC EDITED ITEM CANNOT BE MOVED TO AN ALPHABETIC ITEM
323	A NUMERIC EDITED ITEM CANNOT BE MOVED TO A NUMERIC ITEM
324	A NUMERIC EDITED ITEM CANNOT BE MOVED TO A NUMERIC EDITED ITEM
325	A NON-NUMERIC LITERAL CANNOT BE MOVED TO A COMPUTATIONAL ITEM
326	LIMIT ON NUMBER OF DATA BASES HAS BEEN EXCEEDED
327	DMSTATUS MUST BE FOLLOWED BY PARENTHESESIZED STATUS NAME
328	THIS STATUS NAME IS INVALID
329	COMPONENT AND SEARCH KEY CLASSES ARE INCOMPATIBLE
330	DATASET IS NOT ALLOWED, CURRENT WILL BE ASSUMED
331	NON-GRAPHIC CHARACTER IN PICTURE CLAUSE
332	TOO MANY DATA MANAGEMENT PATHNAMES
333	FOLLOWING STATEMENT NEEDS PARAGRAPH-NAME IN ORDER TO BE EXECUTED
334	ILLEGAL CHARACTER IN COLUMN 7
335	ORDERING NOT PERMITTED WITH RETRIEVAL KEY
336	DATA ENTRY LENGTH EXCEEDS LIMIT OF 16383 DIGITS
337	VA OF ID DATA-NAME SIZE MUST BE 30 CHARACTERS
338	PACK-ID ILLEGAL FOR NON-DISK FILE
339	MISSING OR ILLEGAL RESTART DATASET
340	MISSING COMPONENT LIST
341	TARGET DATASET INVALID WITH DATASET PATHNAME IN THIS FORMAT
342	ORDERED PATHNAME INVALID IN ORDERED ACCESS FORMAT
343	INPLACE INVALID FOR TAPE SORT
344	ILLEGAL SORT TAPES DATANAME
345	OPTIONAL WORD USED AS A PROCEDURE-NAME
346	FILLER USED AS A CONDITIONAL VARIABLE NAME
347	Q-MAX-MESSAGES EXCEEDS 255
348	MAXIMUM OF 9 EVENTS EXCEEDED
349	RECORD CONTAINS SIZE NOT = MAXIMUM FILE RECORD SIZE
350	2 TO 8 MERGE FILES REQUIRED
351	USING REQUIRED IN MERGE STATEMENT
352	VARIABLE LENGTH RECORDS ILLEGAL IN TAG-SEARCH OR INPLACE SORT
353	VARIABLE LENGTH RECORDS UNUSABLE WITH TAG-KEY SORT
354	FILE STATUS MUST BE TWO CHARACTERS LONG
355	RECORD KEY NOT A DATA-NAME WITHIN THE SELECTED FILE
356	RECORD KEY MUST BE SPECIFIED FOR AN INDEXED FILE
357	ORGANIZATION MUST BE INDEXED FOR ACCESS TO BE DYNAMIC
358	EXPECTED SEQUENTIAL OR INDEXED ORGANIZATION
359	EXPECTED FILE STATUS DECLARATION
360	MISMATCH OF FILE ORGANIZATION/ACCESS CONSTRUCTS
361	INDEXED FILE REQUIRES USE PROCEDURE OR AT END/INVALID KEY CLAUSE
362	KEY NOT AS DECLARED IN RECORD KEY CLAUSE
363	ACCESS RANDOM ILLEGAL FOR START STATEMENT

Appendix C (Cont)

<u>ERROR NO.</u>	<u>MESSAGE</u>
364	FILE-LIMIT(S) NOT ALLOWED ON INDEXED FILES
365	RECORD KEY CATEGORY MUST BE ALPHANUMERIC
366	MERGE COMPILER ERROR, BAD PC. OCCUR, SUBMIT SOURCE AND TROUBLE REPORT
380	INVALID CASE VARIABLE PASSED TO ERROR PRINT ROUTINE

S—LANGUAGE PROGRAMS

TABLE OF CONTENTS

PART		Page
1	INTRODUCTION.....	D-1
	Background Information.....	D-1
	Program Parameters.....	D-3
	Container Size.....	D-3
	S-Instruction.....	D-3
	S-Operators (OP1, OP2).....	D-3
	Operands (OPND).....	D-3
	Literal String.....	D-4
	Current Operand Index (COPX).....	D-4
	Current Operand Table (COP).....	D-5
	Segment Number.....	D-6
	Displacement.....	D-6
	Data Length.....	D-6
	Subscript or Index Flag.....	D-6
	Data Type.....	D-6
	ASCII Flag.....	D-7
	Number of Subscripts or Indexes.....	D-7
	Subscript Flag.....	D-7
	Subscript Factors.....	D-7
	Indexing.....	D-8
	Table Bound.....	D-8
	In-Line COP Index (ICOP).....	D-8
2	INSTRUCTION SET.....	D-10
	Arithmetic.....	D-10
	Data Movement.....	D-10
	Branching.....	D-11
	Conditional Branching.....	D-11
	Miscellaneous.....	D-11

TABLE OF CONTENTS (Cont)

PART		Page
3	ARITHMETIC OPERANDS AND INSTRUCTIONS.....	D-12
	Introduction.....	D-12
	Add Three Address (ADD).....	D-14
	Subtract Three Address (SUB).....	D-15
	Add Two Address (INC).....	D-16
	Subtract Two Address (DEC).....	D-17
	Multiply (MULT).....	D-18
	Divide (DIV).....	D-19
	Divide Special (DIVS).....	D-20
	Increment by One (INCL).....	D-21
	Decrement by One (DECL).....	D-22
4	DATA MOVEMENT OPERANDS AND INSTRUCTIONS.....	D-23
	Introduction.....	D-23
	Move Alphanumeric (MVA).....	D-24
	Move Spaces (MVS).....	D-25
	Move Numeric (MVN).....	D-26
	Move Zeros (MVZ).....	D-27
	Concatenate (CAT).....	D-28
	Scaled Move Numeric (SMVN).....	D-29
	Move Translate (MVT).....	D-30
	Examine (EXAM).....	D-31
	Edit Instructions and Micro-Operators.....	D-33
	Introduction.....	D-33
	Edit (EDIT).....	D-34
	Edit With Explicit Mask (EDTE).....	D-35
	Edit Micro-Operators.....	D-36
	Move Digits.....	D-37
	Move Characters.....	D-38
	Move Suppress.....	D-38
	Fill Suppress.....	D-38
	Skip Reverse Destination.....	D-38
	Insert Unconditionally.....	D-38
	Insert on Minus.....	D-39
	Insert Suppress.....	D-39
	Insert Float.....	D-39
	End Float Mode.....	D-40

TABLE OF CONTENTS (Cont)

PART		Page
	End Non-Zero.....	D-40
	End of Mask.....	D-40
	Start Zero Suppress.....	D-40
	Complement Check Protect.....	D-40
	MICR Format (MICF).....	D-41
	MICR Edit (MICE).....	D-44
5	BRANCHING OPERANDS AND INSTRUCTIONS.....	D-45
	Introduction.....	D-45
	Branch Unconditionally (BUN).....	D-46
	Branch On Overflow (BOFL).....	D-47
	Set Overflow Toggle (SOFL).....	D-48
	Perform Enter (PERF).....	D-49
	Perform Exit (PXIT).....	D-50
	Enter (NTR).....	D-51
	Exit (XIT).....	D-52
	Go To Depending (GOTD).....	D-53
	Altered Go To Paragraph (GPAR).....	D-54
	Alter (ALTR).....	D-55
6	CONDITIONAL BRANCH OPERANDS AND INSTRUCTIONS.....	D-56
	Introduction.....	D-56
	Compare Alphanumeric (CMPA).....	D-57
	Compare Numeric (CMPN).....	D-58
	Compare for Zeros (CMPZ).....	D-59
	Compare for Spaces (CMPS).....	D-60
	Compare for Class (CMPC).....	D-61
	Compare Repeat (CMPR).....	D-62
7	MISCELLANEOUS INSTRUCTIONS.....	D-63
	Communicate (COMM).....	D-63
	Load Communicate Reply (LDCR).....	D-64
	Convert (CONV).....	D-65
	Make Present (MAKP).....	D-66
	Hardware Monitor (HMON).....	D-67
8	ALPHABETIC INDEX.....	D-68

INTRODUCTION

BACKGROUND INFORMATION

B 1800/B 1700 COBOL S-Language provides the virtual machine interface between the code generated by the COBOL Compiler and the COBOL Interpreter. This appendix describes the format of COBOL S-instructions and then explains each operator as a member of one of the following classes:

- ARITHMETIC
- DATA MOVEMENT
- BRANCHING
- CONDITIONAL BRANCHING
- MISCELLANEOUS

All COBOL S-language programs have associated with them a base register and a limit register. The area between the base and the limit is to be used as data space only. All program code, organized in segment form, is stored at any available location in memory, according to the memory management algorithms used by the B 1800/B 1700 operating system.

The data space contains the COP table and various other parameters such as Edit Masks and Record Areas.

Various parameters, necessary for the running of the S-Language object code and maintained by the MCP, are stored beyond the Limit Register in the Run Structure Nucleus (RSN).

A typical COBOL program layout in memory is shown in table D-1. Special registers, which are part of the program layout in memory, are shown in table D-2.

Table D-1. COBOL Program Layout

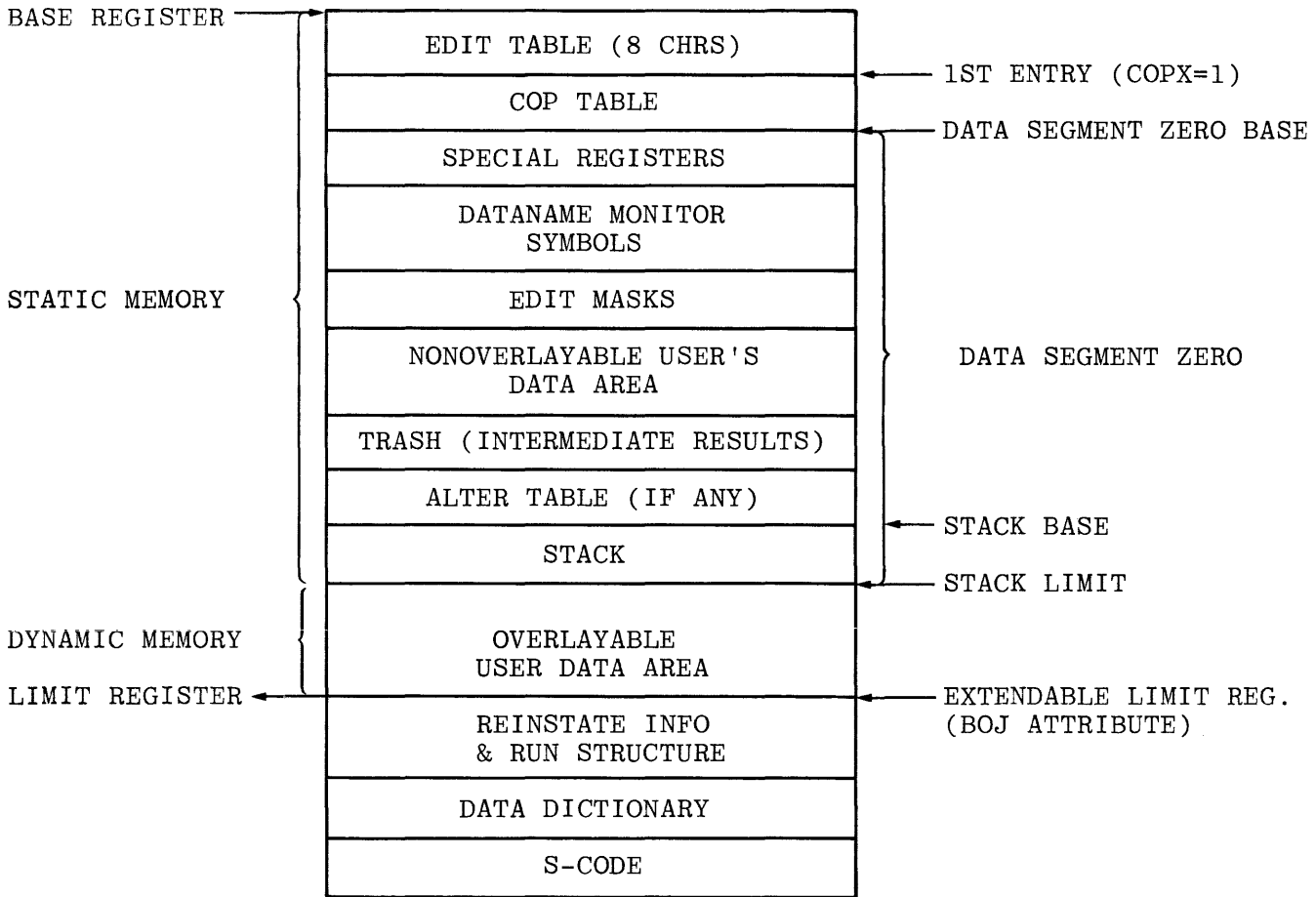


Table D-2. Special Registers

ADR	NAME	PIC
0	SW1	9 CMP
1	SW2	9 CMP
2	SW3	9 CMP
3	SW4	9 CMP
4	SW5	9 CMP
5	SW6	9 CMP
6	SW7	9 CMP
7	SW8	9 CMP
8	TALLY	9(5) CMP
13	DATE (JULIAN) (YYDDD)	9(5) CMP
18	TIME (HHMSST)	9(7) CMP
25	TODAYS-DATE (MMDDYY)	9(6) CMP
31	TODAYS-NAME	X(9)

PROGRAM PARAMETERS

The parameters pertaining to a particular program are listed below. The numbers in parentheses signify the number of bits used to contain the parameter.

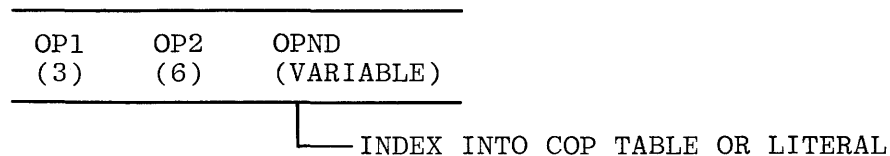
BDISPBI (5)	BRANCH DISPLACEMENT CONTAINER SIZE + 1
DSEGZ (24)	BASE OF DATA SEGMENT ZERO
STACK-POINTER (24)	BASE ADDRESS OF STACK
STACK-SIZE (5)	SIZE OF THE STACK
COP-BASE (24)	BASE ADDRESS OF COP TABLE
COPB (12)	COP ENTRY CONTAINER SIZE
SEGB (5)	DATA SEGMENT NUMBER CONTAINER SIZE
DISPB (5)	DATA DISPLACEMENT CONTAINER SIZE
LENB (5)	DATA LENGTH CONTAINER SIZE
COPXB (5)	COP INDEX CONTAINER SIZE

CONTAINER SIZE

Container size is a field size (in number of bits) necessary to contain the maximum value required for that field. For example: A container size of five bits allows a field value to house 32 bit addresses (0-31).

S-INSTRUCTION FORMAT

Each COBOL S-Instruction consists of an S-operator followed by arguments consisting of a variable number of bits. The format and interpretation of these arguments is specified by the S-operator and is described in detail by the specification of the individual operators. An example of one such instruction format is illustrated below.

**S-Operators (OP1, OP2)**

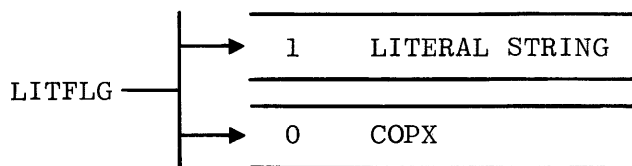
The most frequently used S-operators are encoded in a three bit S-operator denoted as OP1. If OP1 is equal to seven, the operator is encoded in the next six bits denoted as OP2. If OP1 is less than seven, OP2 is omitted.

Operands (OPND)

An operand is normally referenced indirectly through a table containing the attributes of the operand. An argument which references an operand in this manner is denoted as COPX. An operand is either contained in the instruction

Appendix D (Cont)

as a literal or is referenced indirectly through the table. An argument of this type is denoted as OPND. The first bit of OPND is denoted as LITFLG and is used to indicate a literal string or COPX as follows:



Literal String. When LITFLG specifies a literal, the literal string, which includes the literal type (LTYPE), the literal length (LLGTH), and the literal (LSYMB) itself in that order, is included in the code stream immediately following the LITFLG. The format is as follows:

LTYPE	LLTGH1	LLTGH2	LSYMB
(2)	(3)	(8)	(variable)

NOTE

LTYPE

00=Unsigned 4-bit

01=Unsigned 8-bit

10=Signed 4-bit (sign is MSD)

11=Reserved

LLGTH2 present if LLGTH1 equal zero

The length of the literal expressed in binary is encoded in LLGTH1 and LLGTH2. If the length of the literal is less than eight digits or characters, its length is encoded in LLGTH1; and LLGTH2 is omitted. If the length of the literal is greater than or equal to eight digits or characters, its length is encoded in LLGTH2 and LLGTH1 is set to zero. The maximum literal length is 255 digits or characters excluding the sign.

Current Operand Index (COPX). The argument COPX is an index value used to index into the current operand table (COP table). The number of bits (COPXB) used to index into the COP table is a function of the maximum number of COP table entries required for the source program. For example, a COP table consisting of between 512 and 1023 entries would require ten bits.

The address of an entry is calculated by multiplying the value "COPX" by the value "COPB" and then adding the result to the base address of the COP table.

Segment Number

Segment number is expressed in binary and specifies the data segment number of the operand. The container size (SEGB) is a function of the maximum number of data segments specified in the source program. The range of the segment number container size (SEGB) is 0 through 18. If it is zero, then there is no segment number container for that particular program; that program has no segmented (overlayable) data and all data references are to data segment zero, the non-overlayable data segment.

Displacement

Displacement is expressed in binary and specifies the digit displacement of the data from the base of the data segment. All data is stored beginning at an address which modulo 4-BIT must equal zero. The container size (DISPB) is a function of the maximum data displacement specified in the source program. The range of the displacement container size (DISPB) is 1 through 21.

Data Length

Data length is expressed in binary and specifies the number of digits or characters in the data item, excluding the sign. The container size (LENB) is a function of the maximum length specified in the source program. The range of the data length container size (LENB) is 1 through 14; however, the largest data item allowed is 8,191 8-BIT units or 16,383 4-BIT units.

Subscript-or-Index-Flag

The subscript-or-index-flag bit is true to indicate subscripting or indexing and false otherwise. When true, the necessary subscripting or indexing information is contained in the next entry or entries.

Data Type

Data type specifies the type of data as follows:

- 00 = Unsigned 4-BIT
- 01 = Unsigned 8-BIT
- 10 = Signed 4-BIT (sign is MSD)
- 11 = Signed 8-BIT (sign over MSD)

ASCII Flag

The ASCII flag bit of the destination field influences the execution of certain code sensitive S-language instructions. These instructions are:

ADD	MVA	CAT	COMPA
SUB	MVS	SMVN	CMPS
INC	MVN	MVT	
DEC	MVZ		
INCL			
DECL			
DIV			
DIVS			

The ASCII flag bit does not influence the execution of the following code sensitive instructions in which EBCDIC is assumed:

EDIT	MICF
EDTE	MICE
CMPC	

Number of Subscripts or Indexes

When indexing or subscripting is indicated by the subscript-or-index-flag, the number of subscripts or indexes required for the variable is specified as follows:

00	= One
01	= Two
10	= Three
11	= Reserved

Subscript Flag

The bit immediately following the number of subscripts or indexes field indicates the appropriate operation:

0	= Indexing
1	= Subscripting

Subscript Factors

Subscripting requires one to three fields, LENB bits in length, containing the binary factor by which each subscript value is to be multiplied to obtain the proper digit address. The factor is the digit displacement between elements of the table. The value one is subtracted from the subscript value prior to multiplying by the factor. The subscript value may be signed.

Appendix D (Cont)

If the subscript value is zero or negative, or if the final sum of the multiplied subscript values exceeds the table bound, an error communicate will be issued.

If the binary equivalent of the multiplied subscript value or the sum of the multiplied subscript values exceeds 24 bits, overflow is ignored.

A COPX for each subscript value immediately follows the primary COPX in the S-Instruction. A subscript variable must not itself be subscripted or indexed.

NOTE

Literal subscript values are optimized by the compiler by building a new descriptor in-line in the S-Instruction.

Indexing

When indexing is indicated, a COPX for each index value (up to three) immediately follows the primary COPX in the S-Instruction. An index variable must not itself be indexed or subscripted.

An index value is contained in a 28 bit field. The value consists of a 4-bit sign followed by six 4-bit decimal digits. The value is converted to binary and combined with the binary data address at execution time.

If any index value is less than zero or if the sum of the index values exceeds the table bound, an error communicate will be issued.

Table Bound

Table bound is a binary value used to specify the maximum permissible digit displacement from a table base for subscripting and indexing. Its container size is DISPB.

IN-LINE COP INFORMATION (ICOP)

The format for in-line COP information differs from its COP table format (See "CURRENT OPERAND TABLE") when subscripting or indexing is required.

The format for in-line COP information is as follows:

DATA SEG.# (SEGB)	ADDRESS DISPL. (DISPB)	DATA LENGTH (LENB)	SUBSCRIPT-OR- INDEX-FLAG (1)	DATA TYPE (2)	ASCII FLAG (1)
-------------------------	------------------------------	--------------------------	------------------------------------	---------------------	----------------------

NO. OF SUBSCRIPTS OR INDEXES (2)	SUBSCRIPT-FLAG (1)	COPX1 (COPXB)	SUBSCRIPT FACTOR 1 (LENB)	COPX2 (COPXB)	SUBSCRIPT FACTOR 2 (LENB)
---	-----------------------	------------------	---------------------------------	------------------	---------------------------------

COPX3 (COPXB)	SUBSCRIPT FACTOR 3 (LENB)	TABLE BOUND (DISPB)
------------------	---------------------------------	---------------------------

- a. None of the subscripting/indexing information (all entries following the ASCII flag) is present unless the subscript-or-index-flag equals one.
- b. A COPX for each index value, or a COPX/subscript factor pair for each subscript value, must be present as indicated by the value of number of subscripts or indexes.
- c. COPX1, COPX2, and COPX3 may be in-line entries but must not be subscripted or indexed.

INSTRUCTION SET**ARITHMETIC**

<u>NAME</u>	<u>MNEMONIC</u>	<u>OP</u>	<u>ARGUMENTS</u>
INCREMENT	INC	02	OPND1, COPX1
ADD	ADD	08	OPND1, COPX1, COPX2
DECREMENT	DEC	09	OPND1, COPX1
SUBTRACT	SUB	10	OPND1, OPND2, COPX1
MULTIPLY	MULT	11	OPND1, COPX1, COPX2
DIVIDE	DIV	12	OPND1, COPX1, COPX2
DIVIDE SPECIAL	DIVS	16	OPND1, COPX1, COPX2
INCREMENT BY ONE	INC1	13	COPX1
DECREMENT BY ONE	DEC1	14	COPX1

DATA MOVEMENT

<u>NAME</u>	<u>MNEMONIC</u>	<u>OP</u>	<u>ARGUMENTS</u>
MOVE ALPHANUMERIC	MVA	00	OPND1, COPX1
MOVE SPACES	MVS	15	COPX1
MOVE NUMERIC	MVN	01	OPND1, COPX1
MOVE ZEROS	MVZ	22	COPX1
CONCATENATE	CAT	32	N, COPX1, OPND0, ..., OPNDN
SCALED MOVE NUMERIC	SMVN	28	OPND1, COPX1, V, SCL
EXAMINE	EXAM	44	M, T, COPX1, OPND1, COPX2, OPND2
MOVE TRANSLATE	MVT	47	OPND1, COPX1, COPX2
EDIT	EDIT	17	OPND1, COPX1, DADDR
EDIT WITH EXPLICIT MASK	EDTE	21	OPND1, COPX1, MASK
MICR FORMAT	MICF	48	COPX1, COPX2
MICR EDIT	MICE	49	COPX1, COPX2, COPX3

BRANCHING

<u>NAME</u>	<u>MNEMONIC</u>	<u>OP</u>	<u>ARGUMENTS</u>
BRANCH ON OVERFLOW	BOFL	23	V, BADDR
SET OVERFLOW	SOFL	07	V
BRANCH UNCONDITIONALLY	BUN	03	BADDR
PERFORM ENTER	PERF	06	K, BADDR
PERFORM EXIT	PXIT	34	K
ENTER	NTR	18	BADDR
EXIT	XIT	19	
GO TO DEPENDING	GOTD	39	COPX1, L, DBADDR0, . . . , DBADDRL
ALTERED GO TO PARAGRAPH	GPAR	35	DADDR
ALTER	ALTR	36	DADDR, ACON

CONDITIONAL BRANCHING

<u>NAME</u>	<u>MNEMONIC</u>	<u>OP</u>	<u>ARGUMENTS</u>
COMPARE ALPHANUMERIC	CMPA	04	OPND1, COPX1, R, BADDR
COMPARE NUMERIC	CMPN	05	OPND1, COPX1, R, BADDR
COMPARE FOR ZEROS	CMPX	27	COPX1, R, BADDR
COMPARE FOR SPACES	CMPS	37	COPX1, R, BADDR
COMPARE FOR CLASS	CMPC	38	COPX1, C, BADDR
COMPARE REPEAT	CMPR	45	OPND1, COPX1, R, BADDR

MISCELLANEOUS

<u>NAME</u>	<u>MNEMONIC</u>	<u>OP</u>	<u>ARGUMENTS</u>
COMMUNICATE	COMM	33	COPX1
LOAD COMMUNICATE REPLY	LDCR	41	DADDR
CONVERT	CONV	40	COPX1, DADDR
MAKE PRESENT	MAKP	42	COPX1, DADDR
HARDWARE MONITOR	HMON	13	OPND1

ARITHMETIC OPERANDS AND INSTRUCTIONS

INTRODUCTION

In general, arithmetic operands can have any of the following formats:

- a. Unsigned 4-bit
- b. Unsigned 8-bit
- c. Signed 4-bit (sign is MSD)
- d. Signed 8-bit (sign over MSD)

Any restrictions concerning the types of operands permitted in an operation are specified under the description of the particular operation.

All fields are addressed by pointing to the most significant bit of the most significant unit, which in the case of a signed field is the sign.

All fields are considered to be comprised of decimal integers.

The absolute value is stored if the receiving field is unsigned.

Unsigned fields are considered positive.

When signed format is specified for the receiving field for any arithmetic operation, the sign position is set to 1100 for a positive result and to 1101 for a negative result.

Four-bit operands are interpreted in units of four bits. When a signed operand is specified, the sign is interpreted as a separate and leading (left-most) 4-bit unit which is not included in the statement of length.

Eight-bit operands are interpreted in units of eight bits. When a signed operand is specified, the sign is interpreted as being contained in the left-most four bits of the leftmost 8-bit unit.

The length of the operand field specifies the number of 4-bit or 8-bit units.

When 8-bit units are specified for the receiving field of an arithmetic operation, the leftmost four bits of each 8-bit unit, except the unit carrying a sign, is set to 1111 if EBCDIC or to 0011 if ASCII.

The value of an 8-bit unit is carried in the rightmost four bits of the unit. Its value is as defined below for the 4-bit unit. The leftmost four bits, except for a sign, are ignored. The value and sign interpretation of a 4-bit unit is as follows:

<u>UNIT</u>	<u>VALUE</u>	<u>SIGN</u>
0000	0	+
0001	1	+
0010	2	+
0011	3	+
0100	4	+
0101	5	+
0110	6	+
0111	7	+
1000	8	+
1001	9	+
1010	UNDEFINED	+
1011	UNDEFINED	+
1100	UNDEFINED	+
1101	UNDEFINED	-
1110	UNDEFINED	+
1111	UNDEFINED	+

In addition and subtraction, results generated when the size of the result field is not sufficient to contain the result are not specified. When the result field is longer than the length of the result, leading zero units are stored.

In ADD THREE ADDRESS, SUBTRACT THREE ADDRESS, and MULTIPLY, total or partial overlap of the first two operands is permitted. Results generated when the result field totally or partially overlaps either of the operand fields are not specified.

In ADD TWO ADDRESS and SUBTRACT TWO ADDRESS, total overlap is permitted. Results generated when the result field partially overlaps the first operand field are not specified. Note that total overlap implies that the two fields are identical.

No overlap of operands or result fields is permitted in DIVIDE. Results generated under any condition of overlap are not specified.

ADD THREE ADDRESS



OP: 08

Format:

```
*****  
* ADD OPND1, COPX1, COPX2 *  
*****
```

Function:

Algebraically add an addend denoted by OPND1 to an augend denoted by COPX1 and store the sum in the field denoted by COPX2.

SUBTRACT THREE ADDRESS

SUB

OP: 10

Format:

```
*****  
* SUB OPND1, OPND2, COPX1 *  
*****
```

Function:

Algebraically subtract a subtrahend denoted by OPND1 from a minuend denoted by OPND2 and store the difference in the field denoted by COPX1.

Appendix D (Cont)

ADD TWO ADDRESS

INC

OP: 02

Format:

```
*****  
* INC OPND1, COPX1 *  
*****
```

Function:

Algebraically add an addend denoted by OPND1 to an augend denoted by COPX1 and store the sum in the field denoted by COPX1.

SUBTRACT TWO ADDRESS

DEC

OP: 09

Format:

```
*****  
* DEC OPND1, COPX1 *  
*****
```

Function:

Algebraically subtract a subtrahend denoted by OPND1 from a minuend denoted by COPX1 and store the difference in the field denoted by COPX1.

MULTIPLY

MULT

OP: 11

Format:

```
*****  
* MULT OPND1, COPX1, COPX2 *  
*****
```

Function:

Algebraically multiply a multiplicand denoted by COPX1 by a multiplier denoted by OPND1 and store the product in the field denoted by COPX2.

The result field length is the sum of the lengths of the two operands and must be denoted by COPX2.

The result field is always either signed 4-bit format or unsigned 4-bit format.

DIVIDE

DIV

OP: 12

Format:

```
*****
* DIV OPND1, COPX1, COPX2 *
*****
```

Function:

Algebraically divide a dividend denoted by COPX1 by a divisor denoted by OPND1 and store the quotient in the field denoted by COPX2. Store the remainder in the field denoted by COPX1.

The result field length is the difference of the lengths of the two operands and must be denoted by COPX2.

Results are not specified if the length of the dividend is not greater than the length of the divisor.

If the absolute value of the divisor is not greater than the absolute value of an equivalent number of leading digits of the dividend, the result is undefined.

Division by zero results in a fatal error communicate to the MCP.

The sign of the remainder is that of the original dividend.

The dividend field is always either signed 4-bit format or unsigned 4-bit format.

DIVIDE SPECIAL

DIVS

OP: 16

Format:

```
*****  
* DIVS OPND1, COPX1, COPX2 *  
*****
```

Function:

This operation is performed in exactly the same manner as the standard divide (DIV) operator; except that when a divisor equal to zero is encountered, an overflow toggle is set and processing is allowed to continue. The overflow toggle can be manipulated by the "SOFL" and "BOFL" S-operators.

INCREMENT BY ONE

INCI

OP: 13

Format:

```
*****  
* INCl COPX1 *  
*****
```

Function:

Algebraically add the positive integer one to an augend denoted by COPX1 and store the sum in the field specified by COPX1.

Appendix D (Cont)

DECREMENT BY ONE



OP: 14

Format:

```
*****  
* DECI COPX1 *  
*****
```

Function:

Algebraically subtract the positive integer one from a minuend denoted by COPX1 and store the difference in the field specified by COPX1.

DATA MOVEMENT OPERANDS AND INSTRUCTIONS**INTRODUCTION**

In general, fields involved in data movement operations can have any of the following formats:

- a. Unsigned 4-bit
- b. Unsigned 8-bit
- c. Signed 4-bit (sign is MSD)
- d. Signed 8-bit (sign over MSD)

Any restrictions as to the type of fields permitted in an operation are specified under the description of the particular operation.

See arithmetic operands and instructions for a description of the four types of fields.

Totally or partially overlapped fields are not permitted, unless specifically specified by the description of the individual instruction.

MOVE ALPHANUMERIC



OP: 00

Format:

```
*****  
* MVA OPND1, COPX1 *  
*****
```

Function:

Move 8-bit or 4-bit units from the source field denoted by OPND1 to the 8-bit or 4-bit destination field denoted by COPX1.

If the destination field is signed, it receives either the sign of the source if the source is signed, or 1100 if the source is unsigned.

If the data type of the source field is 4-bit and the data type of the destination field is 8-bit, each 4-bit unit is moved to the destination with 1111 if EBCDIC or 0011 if ASCII moved to the leftmost four bits of each 8-bit unit.

If the data type of the source field is 8-bit and the data type of the destination is 4-bit, the rightmost four bits are moved.

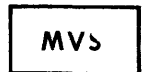
If the data type of the source field is the same as the data type of the destination field, each unit is moved unchanged to the destination.

If the destination length is greater in size than the source length, the destination field is filled in on the right with trailing spaces (0100 0000 if EBCDIC or 0010 0000 if ASCII) if the destination type is 8-bit; otherwise it is filled in on the right with zeros (0000).

If the destination length is lesser in size than the source length, the source data is truncated on the right.

Overlapping operand fields are permitted if the data type of both fields is the same. It can be assumed that the source is moved 24 bits (six digits or three characters) at a time into the destination field and that the move is from left to right.

MOVE SPACES



OP: 15

Format:

```
*****  
* MVS COPX1 *  
*****
```

Function:

Fill the destination field denoted by COPX1 with spaces (0100 0000 if EBCDIC or 0010 0000 if ASCII).

The data type of the destination field is ignored and is assumed to be unsigned 8-bit.

MOVE NUMERIC



OP: 01

Format:

```
*****  
* MVN OPND1, COPX1 *  
*****
```

Function:

Move 8-bit or 4-bit units from the source field denoted by OPND1 to the 8-bit or 4-bit destination field denoted by COPX1.

If the destination field is signed, it receives either the sign of the source if the source is signed, or 1100 if the source is unsigned.

If the destination field is unsigned, the sign of the source is ignored.

If the data type of the destination field is 8-BIT, the leftmost four bits of each 8-bit unit, except for the sign position, if signed, are set to 1111 if EBCDIC or to 0011 if ASCII, regardless of the data type of the source field.

If the data type of the destination field is 4-BIT, the leftmost four bits of each source 8-bit unit are ignored and only the rightmost four bits are moved; if the source field is a 4-bit field, each 4-bit unit is moved unchanged.

If the destination length is greater in size than the source length, the destination field is filled in on the left with leading zeros of appropriate type (1111 0000 if EBCDIC, 0011 0000 if ASCII or 0000 if 4-bit).

If the destination length is lesser in size than the source length, the source data is truncated on the left.

Note that a sign is placed in the leftmost four bits of a field, whether 4-bit or 8-bit.

Overlapping operand fields are permitted if the data type of both fields is the same. It can be assumed that the source is moved 24 bits (six digits or three characters) at a time into the destination field and that the move is from left to right.

MOVE ZEROS

MVZ

OP: 22

Format:

```
*****  
* MVZ COPX1 *  
*****
```

Function:

Fill the destination field denoted by COPX1 with zeros of the appropriate type (1111 0000 if EBCDIC, 0011 0000 if ASCII or 0000 if 4-bit).

If the destination field is signed, 1100 is placed into the sign position.

CONCATENATE



OP: 32

Format:

```
*****  
* CAT N, COPX1, OPND0, ..., OPNDN *  
*****
```

Function:

Move each of the N+1 fields denoted by OPND0 through OPNDN, in the order specified, into an output string starting at the field denoted by COPX1.

The number of source fields is specified by the 4-bit binary value N. The value N ranging from 0000 to 1111 is used to indicate 1 to 16 source fields.

Each field is moved according to the rules specified for MOVE ALPHANUMERIC.

If the destination length is greater in size than the combined source length, the destination field is filled in on the right with trailing spaces (0100 0000 if EBCDIC or 0010 0000 if ASCII).

If the destination length is lesser in size than the combined source lengths, the source data is truncated on the right.

SCALED MOVE NUMERIC**SMVN**

OP: 28

Format:

```

*****
* SMVN OPND1, COPX1, V, SCL *
*****

```

Function:

If V equals 0, perform a MOVE NUMERIC operation after first adding the scale factor to the field length of the source field and assuming that the added portion of the field is zeros on the right. The scale factor must not be greater than the destination field length.

If V equals one, perform a MOVE NUMERIC operation after first subtracting the scale factor from the field length of the source field. The scale factor must not be greater than the source field length.

All rules specified for MOVE NUMERIC are applicable after adjustment by the scale factor.

The container size for the scale factor is the same as the container size for the length of an operand (LENB). The length of V is one bit.

MOVE TRANSLATE



OP: 47

Format:

```
*****  
* MVT OPND1, COPX1, COPX2 *  
*****
```

Function:

Move 8-bit units from the source field denoted by OPND1 to the destination field denoted by COPX2, translating enroute.

Translation is accomplished by using each 8-BIT source character, multiplied by eight, as an index into the translation table, denoted by COPX1, to obtain the translated character.

The data type of the source and table fields are ignored and are assumed to be unsigned 8-BIT. The destination field is also assumed to be unsigned, but may be 4-bit or 8-bit.

If the destination length is greater in size than the source length, the destination field is filled in on the right with trailing spaces (0100 0000 if EBCDIC, 0010 0000 if ASCII or 0000 if 4-bit).

If the destination length is lesser in size than the source length, the source data is truncated on the right.

Total overlap of operand fields is permitted, to allow in-place translation.

EXAMINE**EXAM**

OP: 44

Format:

```
*****
* EXAM M, T, COPX1, OPND1, COPX2, OPND2 *
*****
```

Function:

Examine the operand defined by COPX1, tallying and/or replacing a variable number of 8-bit characters. The particular 8-bit character to be tallied and/or replaced is specified by OPND1. The character to be used as the replacement character is specified by OPND2. The field into which the tally is stored is specified by COPX2.

The type of operation is specified by the 4-bit parameter, T. If these four bits are identified, left to right, as T1, T2, T3 and T4, then T is interpreted as follows:

```
T1T2 = 00 undefined
        01 tally T3T4 occurrences of the character specified by OPND1
        10 replace T3T4 occurrences of the character specified by OPND1
        11 tally and replace T3T4 occurrences of the character specified
           by OPND1

T3T4 = 00 all
        01 (all) leading
        10 until first
        11 first
```

NOTE

T1T2T3T4 = 0111 and 1111 not specified
and results are undefined.

The OPND2 argument is not present when T1T2 = 01.

The COPX2 argument, is not present when T1T2 = 10.

The data type of the examined operand (COPX1) is assumed to be signed or unsigned 8-bit. If it is signed, then the original sign is preserved by this operation.

Appendix D (Cont)

The data type of the examining operand, defined by OPND1, must be unsigned. Its length is assumed to be one. When 4-bit format is specified, the operand is assumed to have the four bits 1111 if EBCDIC or 0011 if ASCII appended to the left.

The data type of the replacing operand, defined by OPND2, must be unsigned. Its length is assumed to be one. When 4-bit format is specified, the leftmost four bits of the position replaced are set to 1111 if EBCDIC or 0011 if ASCII, and the rightmost four bits receive the four bits from the replacing source. When 8-bit format is specified, the position replaced receives all eight bits from the replacing source.

The data type of the tally field defined by COPX2 is assumed to be unsigned 4-bit. Its length is assumed to be five.

If the one bit parameter M equals zero, it denotes numeric items, and only the rightmost four bits of a character are used in the comparison; the leftmost four bits are ignored. If M equals one, alphanumeric items are denoted, and all eight bits of a character are used in comparing.

EDIT INSTRUCTIONS AND EDIT MICRO-OPERATORS

Introduction

No restrictions are placed on the data type of the source field of an edit operation.

The data type of the destination field of an edit operation must be unsigned 8-bit.

If the destination length is greater in size than the source length, the source data is assumed to have leading zero fill on the left.

If the destination length is lesser in size than the source length, the source data is truncated on the left.

The operation is terminated by an edit micro-operator and not by exhaustion of either the source or destination fields.

EDIT



OP: 17

Format:

```
*****  
* EDIT OPND1, COPX1, DADDR *  
*****
```

Function:

Move data from the source field, denoted by OPND1, to the destination field, denoted by COPX1, under the control of the micro-operator string contained at the location denoted by the DADDR.

The argument DADDR is an unsigned binary value which specifies the digit displacement of the micro-operator string relative to the data segment zero base. The container size of DADDR is DISPB.

EDIT WITH EXPLICIT MASK

EDTE

OP: 21

Format:

```
*****
* EDTE OPND1, COPX1, MASK *
*****
```

Function:

Move data from the source field denoted by OPND1 to the destination field denoted by COPX1 under the control of the micro-operator string immediately following COPX1. The format of the explicit micro-operator string is the same as a literal and is as follows:

LTYPE (2)	LLGTH1 (3)	LLGTH2 (8)	MICRO-OPERATOR STRING (variable)
		↓ Present if LLGTH1 equals zero	
	↓ Length of the micro-operator string in 8-bit units. If length is greater than or equal to eight units, the length is encoded in LLGTH2 and LLGTH1 is set to zero.		
↓ 01: Unsigned 8-bit format			

Appendix D (Cont)

EDIT Micro-Operators

The edit micro-operators used in an edit instruction are:

<u>OPERATOR</u>	<u>MNEMONIC</u>	<u>OPERATION</u>
0000 "R"	MVD	MOVE DIGITS
0001 "R"	MVC	MOVE CHARACTERS
0010 "R"	MVS	MOVE SUPPRESS
0011 "R"	FIL	FILL SUPPRESS
0100 "N"	SRD	SKIP REVERSE DESTINATION
0101 "T"	INU	INSERT UNCONDITIONALLY
0110 "T"	INM	INSERT ON MINUS
0111 "T"	INS	INSERT SUPPRESS
1000 "T"	INF	INSERT FLOAT
1001 "T"	EFM	END FLOAT MODE
1010 0000	ENZ	END NON-ZERO
1010 0001	EOM	END OF MASK
1010 0010	SZS	START ZERO SUPPRESS
1010 0011	CCP	COMPLEMENT CHECK PROTECT
OTHERS		UNDEFINED

"R" indicates a 4-bit binary value used as a repeat count. The value 0000 represents no repeat, do it once.

"N" indicates a 4-bit binary value used to skip over a number of destination 8-bit units. The value 0000 represents no skip.

"T" indicates a 4-bit binary value which is:

- a. used to index into a table of editing constants
- b. used to indicate a conditional selection between two table constants
- c. used to indicate an editing constant in line with the edit-operator string.

The next edit-operator follows the constant.

Table D-3 indicates the normal table editing constants as well as the conditional and unconditional selection of constants associated with the value "T".

Table D-3. Editing Constants and Toggles

EDITING CONSTANTS			
"T" VALUE	ENTRY NUMBER	TABLE ENTRY OR CONSTANT	MNEMONIC
0000	0	."+	PLU (PLUS)
0001	1	"-"	MIN (MINUS)
0010	2	"*"	AST (ASTERISK)
0011	3	."	DPT (DECIMAL POINT)
0100	4	","	CMA (COMMA)
0101	5	"\$"	CUR (CURRENCY)
0110	6	"0"	ZRO (ZERO)
0111	7	" "	BLK (BLANK)
1000	-	ENTRY 0 or 1	SPM (SIGN PLUS OR MINUS)
1001	-	ENTRY 7 or 1	SBM (SIGN BLANK OR MINUS)
1010	-	IN-LINE 8-BIT CONSTANT	LIT (LITERAL)
EDITING TOGGLES			
<u>TOGGLE</u>	<u>USE</u>		
"S" (SIGN)	Set to zero if the source field sign is positive, otherwise set to one.		
"Z" (ZERO SUPPRESS)	Set and reset by the individual micro-operators		
"P" (CHECK PROTECT)	Set and reset by the individual micro-operators		

Associated with the edit instructions are three toggles denoted as "S" for sign, "Z" for zero suppress and "P" for check protect. Initially the "Z" and the "P" toggles are assumed to be set to the zero state. They are set and reset as specified by the description of the individual micro-operators. The "S" toggle is set to zero if the source field sign is positive and to one otherwise. Unsigned fields are considered positive.

Operations are detailed below; explanations of the individual micro-operators follow the definitions.

Move Digit. Set "Z" to "1", ending the zero suppress state. Move an appropriate unit (4-bit digit or 8-bit character) from the source field to the destination field. If a 4-bit unit is moved, append the four bits 1111 to the left before storing in the destination. If an 8-bit unit is moved, the four bits 1111 are substituted for the leftmost four bits of the 8-bit unit.

Move Character. Set "Z" to "1", ending the zero suppress state. Move an appropriate unit (4-bit digit or 8-bit character) from the source field to the destination field. If a 4-bit unit is moved, append the four bits 1111 to the left before storing in the destination. If an 8-bit unit is moved, it is moved unchanged.

Move Suppress. The micro-operator "MOVE DIGIT" is performed if the 4-bit unit, or the rightmost four bits of the 8-bit unit, of the source field is not equal to 0000.

If the appropriate four bits of the source field unit are equal to 0000, the suppress toggle "Z" is inspected. If "Z" equals "1", indicating non-suppress mode, the micro-operator "MOVE DIGIT" is performed. If the suppress toggle "Z" equals "0", the check protect toggle "P" is inspected. If "P" = "0", indicating non-check protect mode, move the table entry containing the 8-bit code for blank to the destination field. If "P" = "1", move the table entry containing the 8-bit code for asterisk to the destination field.

Summary

```

SOURCE NOT = 0 MOVE DIGIT
Z=1 SOURCE = 0 MOVE DIGIT
Z=0 P=0 SOURCE = 0 MOVE TABLE ENTRY 7 (BLANK)
Z=0 P=1 SOURCE = 0 MOVE TABLE ENTRY 2 (ASTERISK)

```

Fill Suppress. If "P" = "0", indicating non-check protect mode, move the table entry containing the 8-bit code for blank to the destination field. If "P" = "1", move the table entry containing the 8-bit code for asterisk to the destination field.

Summary

```

P = 0 MOVE TABLE ENTRY 7 (BLANK)
P = 1 MOVE TABLE ENTRY 2 (ASTERISK)

```

Skip Reverse Destination. Adjust the address pointer of the destination field to skip backward (lower address) "N" 8-bit units.

Insert Unconditionally. Move the table entry "T" as indicated below to the destination field.

```

T=0...7 MOVE TABLE ENTRY "T"
S=0 T=8 MOVE TABLE ENTRY 0 (PLUS)
S=1 T=8 MOVE TABLE ENTRY 1 (MINUS)
S=0 T=9 MOVE TABLE ENTRY 7 (BLANK)
S=1 T=9 MOVE TABLE ENTRY 1 (MINUS)
T=10 MOVE IN-LINE CONSTANT

```

Insert on Minus. Move the table entry "T" as indicated below to the destination field.

```
S=1      T=0...7      MOVE TABLE ENTRY "T"
*   P=0   T=0...7,10  MOVE TABLE ENTRY 7 (BLANK)
*   P=1   T=0...7,10  MOVE TABLE ENTRY 2 (ASTERISK)
S=1      T=8          MOVE TABLE ENTRY 1 (MINUS)
*       T=8          MOVE TABLE ENTRY 0 (PLUS)
S=1      T=9          MOVE TABLE ENTRY 1 (MINUS)
*       T=9          MOVE TABLE ENTRY 7 (BLANK)
S=1      T=10         MOVE IN-LINE CONSTANT
```

*: S = 0 or only source digits/characters equal to zero (minus zero) have been moved.

Insert Suppress. Move the table entry "T" as indicated below to the destination field.

```
Z=1      T=0...7      MOVE TABLE ENTRY "T"
Z=0 P=0          MOVE TABLE ENTRY 7 (BLANK)
Z=0 P=1          MOVE TABLE ENTRY 2 (ASTERISK)
Z=1      S=0 T=8    MOVE TABLE ENTRY 0 (PLUS)
Z=1      S=1 T=8    MOVE TABLE ENTRY 1 (MINUS)
Z=1      S=0 T=9    MOVE TABLE ENTRY 7 (BLANK)
Z=1      S=1 T=9    MOVE TABLE ENTRY 1 (MINUS)
Z=1      T=10       MOVE IN-LINE CONSTANT
```

Insert Float. Move the table entry "T" and/or perform the micro-operator "MOVE DIGIT" as indicated below.

```
Z=1          MOVE DIGIT
Z=0 SOURCE   =0 P=0    MOVE TABLE ENTRY 7 (BLANK)
Z=0 SOURCE   =0 P=1    MOVE TABLE ENTRY 2 (ASTERISK)
Z=0 SOURCE NOT=0 T=0..7  MOVE TABLE ENTRY "T", THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=8 S=0  MOVE TABLE ENTRY 0 (PLUS) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=8 S=1  MOVE TABLE ENTRY 1 (MINUS) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=9 S=0  MOVE TABLE ENTRY 7 (BLANK) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=9 S=1  MOVE TABLE ENTRY 1 (MINUS) THEN MOVE DIGIT
Z=0 SOURCE NOT=0 T=10    MOVE IN-LINE CONSTANT, THEN MOVE DIGIT
```

Appendix D (Cont)

End Float Mode. Move the table entry "T" as indicated below to the destination field.

Z=0		T=0...7	MOVE TABLE ENTRY "T"
Z=0	S=0	T=8	MOVE TABLE ENTRY 0 (PLUS)
Z=0	S=1	T=8	MOVE TABLE ENTRY 1 (MINUS)
Z=0	S=0	T=9	MOVE TABLE ENTRY 7 (BLANK)
Z=0	S=1	T=9	MOVE TABLE ENTRY 1 (MINUS)
Z=0		T=10	MOVE IN-LINE CONSTANT
Z=1	NO OPERATION		

End Non-Zero. Terminate the micro-operator operations if any non-zero source character/digit has been moved; otherwise continue with the next in-line operator.

End of Mask. Terminate the micro-operator operations.

Start Zero Suppress. Set "Z" to the "0" state.

Complement Check Protect. Complement the state of "P".

MICR FORMAT**MICF**

OP: 48

Format:

```
*****
* MICF COPX1, COPX2 *
*****
```

Function:

Format the data from the source field denoted by COPX1 into the destination field denoted by COPX2.

The data type of both the source and the destination fields is assumed to be unsigned 8-BIT.

The field length of the destination MODULO 20 must equal zero. The destination field is considered to be composed of a number of 20 character subfields. Data movement is right to left beginning with the rightmost character of the source field and beginning with the rightmost character position of the destination field.

In the discussion that follows, the following definitions apply:

- a. Transfer characters are characters that are automatically transferred from the source field into the current destination subfield. They never occupy the rightmost control character position of a destination subfield. They are the numeric "0" through "9" and the HYPHEN "-". The HYPHEN is not expected to occur for OCR input.
- b. Defined control characters are characters that cause some specific action to be taken, depending on the character. They are: the END-OF-DOCUMENT "", the MICR CANT-READ "*" and the OCR CANT-READ @3F@.
- c. Default control characters are characters other than transfer and defined control characters. They are expected to be, but are not limited to, the MICR AMOUNT "#", TRANSIT "@" and ON-US ":" and the OCR HOOK "<", FORK "=", CHAIR ">", VERTICAL BAR "|", BLANK @40@ and PLUS "+".

Appendix D (Cont)

Operation is as follows:

- a. Begin formatting into a subfield by fetching a source field character, unless the source field is exhausted, and then proceeding to step a1.
 1. If the source field is exhausted, assume an END-OF-DOCUMENT (') character and proceed to step a2.
 2. If the source field character is an END-OF-DOCUMENT character, move it to the rightmost position of the current subfield, blank-fill the rest of the destination field and then terminate the operation.
 3. If the source field character is other than a default control or END-OF-DOCUMENT character, move a blank to the rightmost position of the current subfield, then move the source character and proceed to step b1.
 4. If the source field character is a default control character, move it to the rightmost position of the current subfield and then proceed to step b1.
- b. Continue formatting into the current subfield by fetching a new source character and then proceeding to step b1.
 1. If the source field is exhausted, blank-fill the rest of the current subfield, assume an END-OF-DOCUMENT source character and proceed to step a2.
 2. If the source field character is an END-OF-DOCUMENT character, blank-fill the rest of the current subfield, save the source field character and proceed to step a2.
 3. If the source field character is other than a default control or END-OF-DOCUMENT character, store the character in the destination and proceed to step b1.
 4. If the source field character is a default control character that is equal to the character in the rightmost position of the current subfield, move it to the next position of the current subfield, blank-fill the rest of the current subfield and then proceed to step a1.
 5. If the source field character is a default control character, but it is not equal to the character in the rightmost position of the current subfield, the rest of the current subfield is blank-filled and the control character is used in step a4, to which we now proceed.

NOTES

1. If any attempt is made to exceed the size of any subfield or of the entire destination field, the overflow toggle is set to one, the operation is terminated and the contents of the destination field are undefined.
2. If any individual subfield contains a CANT-READ ("*" or "@3F@) character, then the high order (leftmost) position of the subfield is set to 1101 0001; otherwise, it is set to a blank (0100 0000).

MICR EDIT



OP: 49

Format:

```
*****  
* MICE COPX1, COPX2, COPX3 *  
*****
```

Function:

Move data from the source field denoted by COPX1 to the destination field denoted by COPX2 deleting all characters except numeric characters ("0" through "9") and CANT-READ characters ("*" and @3F@).

The moved characters are right justified in the destination field and zero filled on the left, if necessary, to fill the remaining destination area. If the destination field is lesser in size than the moved data, the source data is truncated on the left.

A decimal count of all numeric characters moved is provided in the special COBOL register "TALLY" denoted by COPX3.

The data type of the source field must be unsigned 8-bit. The data type of the destination field must be unsigned 4-bit or 8-bit. The data type of the "TALLY" field must be unsigned 4-bit and its length is assumed to be five.

BRANCHING OPERANDS AND INSTRUCTIONS**INTRODUCTION**

A branch address argument "BADDR" has the following format:

DISPLACEMENT (BDISPB)	BTYPE (1)	SEGMENT NUMBER (7)
		present if BTYPE = 1
	0:	Relative to the current code segment base (intra-segment branch)
	1:	Relative to a new code segment base (inter-segment branch)

Displacement is an unsigned binary value which specifies the bit displacement of an instruction relative to a segment base. The container size of the displacement and BTYPE combined is a program parameter (BDISPB1).

BRANCH UNCONDITIONALLY

BUN

OP: BUN

Format:

```
*****  
* BUN BADDR *  
*****
```

Function:

Obtain the next instruction from the location specified by BADDR.

BRANCH ON OVERFLOW

BOFL

OP: 23

Format:

```
*****  
* BOFL V, BADDR *  
*****
```

Function:

If the overflow toggle equals V, a transfer to the address (BADDR) given in the instruction occurs, otherwise control is passed to the next sequential instruction.

The overflow toggle is unchanged. The length of V is one bit.

SET OVERFLOW TOGGLE

SOFL

OP: 07

Format:

```
*****  
* SOFL V *  
*****
```

Function:

Set the overflow toggle to V.

The length of V is one bit.

NOTE

The overflow toggle is set to one if a "DIVIDE BY ZERO" is encountered in the DIVIDE SPECIAL S-operator or if a field overflow is attempted in the MICR FORMAT S-operator.

PERFORM ENTER**PERF**

OP: 06

Format:

```

*****
* PERF K, BADDR *
*****

```

Function:

Create a stack entry with the following format:

DISPLACEMENT	SEGMENT NO.	K
(24)	(7)	(8)

Insert a displacement value, relative to the active code segment base and pointing to the next sequential S-instruction, into the stack.

Insert the current code segment number into the stack. Insert the value of K from the instruction into the stack.

Adjust the stack pointer to point to the next possible entry.

Obtain the next instruction from the location specified by BADDR.

PERFORM EXIT

PXIT

OP: 34

Format:

```
*****  
* PXIT K *  
*****
```

Function:

Compare the K contained in the instruction to the K in the current stack entry and if unequal proceed to the next in-line S-instruction. If equal, adjust the stack pointer to point to the previous entry and obtain the next S-instruction from the information contained in the removed stack entry.

ENTER

NTR

OP: 18

Format:

```
*****  
* NTR BADDR *  
*****
```

Function:

Same function as "PERF". K is assumed equal to zero.

Appendix D (Cont)

EXIT



OP: 19

Format:

```
*****  
* XIT *  
*****
```

Function:

Same function as "PXIT". K is assumed equal to zero.

GO TO DEPENDING**GOTD**

OP: 39

Format:

```
*****
* GOTD COPX1, L, DBADDR0, ..., DBADDRL *
*****
```

Function:

Compare the ten bit binary value L with the variable specified by COPX1. The variable is first converted to a binary value, MODULO 2 to the 24th power.

If the binary value of the variable is less than zero or greater than L, the next instruction is obtained from the location specified by DBADDR0. Note that the variable can be signed.

If the binary value of the variable is in the range zero through L, it is used as an index to select from the list of DBADDR's the appropriate DBADDR to be used to obtain the next instruction.

DBADDR and BADDR have the same format with the exception that DBADDR always contains the segment number. Although segment number is unnecessary for those DBADDR's with BTYPE equal to zero, in order to index into the list of DBADDR's, all of the DBADDR's must be of equal length. The container size of DBADDR is BDISPB1 + 7.

ALTERED GO TO PARAGRAPH

GP**AR**

OP: 35

Format:

```
*****  
* GPAR DADDR *  
*****
```

Function:

Obtain the next instruction from the location specified by the address "ACON".

The address constant "ACON" has the same format as a BADDR.

The argument DADDR is an unsigned binary value which specifies the digit displacement of the "ACON" relative to the data segment zero base.

The container size of DADDR is DISPB.

ALTER**ALTER**

OP: 36

Format:

```
*****  
* ALTR DADDR, ACON *  
*****
```

Function:

Copy the address constant "ACON" into the data area specified by the argument DADDR.

The address constant "ACON" has the same format as a BADDR.

The argument DADDR is an unsigned binary value which specifies the digit displacement of the "ACON" relative to the data segment zero base.

The container size of DADDR is DISPB.

CONDITIONAL BRANCH OPERANDS AND INSTRUCTIONS

INTRODUCTION

If the condition "A (R) B" is true a transfer to the address (BADDR) given in the instruction occurs, otherwise control is passed to the next sequential instruction. The relation (R) is defined as follows:

000	UNDEFINED
001	GTR
010	LSS
011	NEQ
100	EQL
101	GEQ
110	LEQ
111	UNDEFINED

Overlap of fields is permitted. "A" is the first operand denoted in the instruction. If an instruction has only one operand, then the assumed field is the "A" field.

COMPARE ALPHANUMERIC**CMPA**

OP: 04

Format:

```

*****
* CMPA OPND1, COPX1, R, BADDR *
*****

```

Function:

Compare the two operand fields according to their binary values.

The comparison is performed left to right with any shorter operand assumed to be right-filled with blank characters (0100 0000 if EBCDIC or 0010 0000 if ASCII).

The fields are considered equal when the equal size portions are equal and the longer (if one is longer) field has trailing blanks.

8-BIT data format is assumed for both fields with no checking to verify otherwise. Signed fields have their most significant four bits, i.e., their sign, modified to the appropriate numeric zone (1111 for EBCDIC, 0011 for ASCII) before being compared. This modification is not permanent and is done so that sign does not affect the result of an alphanumeric comparison.

COMPARE NUMERIC



OP: 05

Format:

```
*****  
* CMPN OPND1, COPX1, R, BADDR *  
*****
```

Function:

Compare the two operand fields according to the algebraic values, considering the two fields to be comprised of decimal integers.

When the field sizes are different, the longer is tested for leading zeros (0000). There is no restriction as to data type. In comparing an 8-bit character only the rightmost four bits of the character are considered; the other bits are ignored.

Two fields of all zeros are equal regardless of sign.

Unsigned fields are considered positive. Sign conventions are the same as for arithmetic operands.

Results generated by invalid digit values are undefined.

COMPARE FOR ZEROS**CMPZ**

OP: 27

Format:

```
*****  
* CMPZ COPX1, R, BADDR *  
*****
```

Function:

Compare two operand fields according to their algebraic values, assuming the first field to be comprised of all zeros (0000).

There is no restriction as to data type. In comparing an 8-bit character only the rightmost four bits of the character are considered. The other bits are ignored.

Two fields of all zeros are equal regardless of sign.

Unsigned fields are considered positive. Sign conventions are the same as for arithmetic operands.

Results generated by invalid digit values are undefined.

COMPARE FOR SPACES

CMPS

OP: 37

Format:

```
*****  
* CMPX COPX1, R, BADDR *  
*****
```

Function:

Compare two operand fields according to their binary values, assuming the first field to be comprised of all spaces (0100 0000 if EBCDIC or 0010 0000 if ASCII).

The comparison is performed left to right.

Unsigned 8-bit format is assumed with no checking to verify otherwise.

COMPARE FOR CLASS

CMPC

OP: 38

Format:

```
*****
* CMPC COPX1, C, BADDR *
*****
```

Function:

Compare the operand field and determine whether the field is:

```
C=00 COMPLETELY ALPHABETIC
C=01 COMPLETELY NUMERIC
C=10 NOT COMPLETELY ALPHABETIC
C=11 NOT COMPLETELY NUMERIC
```

If the condition being tested is true, a transfer to the address BADDR given in the instruction occurs, otherwise control is passed to the next sequential instruction.

In the alphabetic test, each character is range-checked for 1100 0001 through 1100 1001, 1101 0001 through 1101 1001, 1110 0010 through 1110 1001 and for 0100 0000. Unsigned 8-bit format is assumed with no checking to verify otherwise.

In the numeric test each character is range-checked for 1111 0000 through 1111 1001. Signed or unsigned 8-bit format is permitted. The four bits in the sign position of a signed 8-bit field are ignored. The sign position is the leftmost four bits of the most significant character.

COMPARE REPEAT

CMPR

OP: 45

Format:

```
*****  
* CMPR OPND1, COPX1, R, BADDR *  
*****
```

Function:

Compare the two operand fields according to their binary value.

Comparison proceeds from left to right.

The field lengths are considered equal by repeating OPND1.

Both fields are assumed to have unsigned 8-bit data type.

The size of OPND1 must divide evenly into the size of COPX1; otherwise, the results of the compare may be erroneous.

MISCELLANEOUS INSTRUCTIONS

COMMUNICATE

COMM

OP: 33

Format:

```
*****  
* COMM COPX1 *  
*****
```

Function:

Move the length and address fields from the COPX1 entry to the RS.COMMUNICATE. MSR.PTR field located in this program's RS.NUCLEUS, converting them enroute. The origin field is unchanged.

The length is converted from a digit or character length to a bit length. The address is stored as an absolute bit address.

Appendix D (Cont)

LOAD COMMUNICATE REPLY

LDCR

OP: 41

Format:

```
*****  
* LDCR DADDR *  
*****
```

Function:

Move the last 24 bits of information from the RS.REPLY area of the RS.NUCLEUS to the location specified by DADDR.

See 'MAKE PRESENT' for definition of DADDR.

CONVERT

CONV

OP: 40

Format:

```
*****  
* CONV COPX1 DADDR *  
*****
```

Function:

Convert the operand denoted by COPX1 from a decimal value to an unsigned 24 bit binary value, truncating or zero filling on the left if necessary. Place the result at the location specified by DADDR.

The operand must be either unsigned 4-bit or unsigned 8-bit units.

See 'MAKE PRESENT' for definition of DADDR.

MAKE PRESENT



OP: 42

Format:

```
*****  
* MAKP COPX1, DADDR *  
*****
```

Function:

Load the data segment specified by COPX1 and place the base relative address of the data area specified by COPX1 into the 24 bit location specified by DADDR.

DADDR is an unsigned binary value which specifies a digit displacement from the data segment zero base.

The container size of DADDR is DISPB.

HARDWARE MONITOR



OP: 43

Format:

```
*****  
* HMON OPND1 *  
*****
```

Function:

The low order eight bits of the field described by OPND1 are used as the input to the monitor micro-operator.

The length of the field described by OPND1 must be greater than or equal to eight bits.

ALPHABETIC INDEX

ADD.....	D-14	CONDITIONAL BRANCHING.....	D-11
ADD THREE ADDRESS.....	D-14	CONTAINER SIZE.....	D-3
ADD TWO ADDRESS.....	D-16	CONV.....	D-65
ALTER.....	D-55	CONVERT.....	D-65
ALTERED GO TO PARAGRAPH.....	D-54	CURRENT OPERAND INDEX (COPX)....	D-4
ALTR.....	D-55	CURRENT OPERAND TABLE (COP)....	D-5
ARITHMETIC.....	D-10	DATA LENGTH.....	D-6
ARITHMETIC OPERANDS AND		DATA MOVEMENT.....	D-10
INSTRUCTIONS.....	D-12	DATA MOVEMENT OPERANDS AND	
ASCII FLAG.....	D-7	INSTRUCTIONS.....	D-23
BOFL.....	D-47	DATA TYPE.....	D-6
BRANCH ON OVERFLOW.....	D-47	DEC.....	D-17
BRANCH UNCONDITIONALLY.....	D-46	DECREMENT BY ONE.....	D-22
BRANCHING.....	D-11	DECL.....	D-22
BRANCHING OPERANDS AND		DISPLACEMENT.....	D-6
INSTRUCTIONS.....	D-45	DIV.....	D-19
BUN.....	D-46	DIVIDE.....	D-19
CAT.....	D-28	DIVIDE SPECIAL.....	D-20
CMPA.....	D-57	DIVS.....	D-20
CMPC.....	D-61	EDIT.....	D-34
CMPN.....	D-58	EDIT INSTRUCTIONS AND EDIT	
CMPR.....	D-62	MICRO-OPERATORS.....	D-33
CMPS.....	D-60	EDIT MICRO-OPERATORS.....	D-36
CMPZ.....	D-59	EDIT WITH EXPLICIT MASK.....	D-35
COBOL PROGRAM LAYOUT		EDITING CONSTANTS.....	D-37
(TABLE D-1).....	D-2	EDTE.....	D-35
COMM.....	D-63	END FLOAT MODE.....	D-40
COMMUNICATE.....	D-63	END NON-ZERO.....	D-40
COMPARE ALPHANUMERIC.....	D-57	END OF MASK.....	D-40
COMPARE FOR CLASS.....	D-61	ENTER.....	D-51
COMPARE FOR SPACES.....	D-60	EXAM.....	D-31
COMPARE FOR ZEROS.....	D-59	EXAMINE.....	D-31
COMPARE NUMERIC.....	D-58	EXIT.....	D-52
COMPARE REPEAT.....	D-62	FILL SUPPRESS.....	D-38
COMPLEMENT CHECK PROTECT.....	D-40	GENERAL.....	D-1
CONCATENATE.....	D-28	GO TO DEPENDING.....	D-53
CONDITIONAL BRANCH OPERANDS		GOTD.....	D-53
AND INSTRUCTIONS.....	D-56	GPAR.....	D-54

HARDWARE MONITOR.....	D-67	MVA.....	D-24
HMON.....	D-67	MVN.....	D-26
IN-LINE COP INFORMATION.....	D-8	MVS.....	D-25
INC.....	D-16	MVT.....	D-30
INCREMENT BY ONE.....	D-21	MVZ.....	D-27
INCL.....	D-21	NTR.....	D-51
INDEXING.....	D-8	NUMBER OF SUBSCRIPTS	
INSERT FLOAT.....	D-39	OR INDEXES.....	D-7
INSERT ON MINUS.....	D-39	OPND.....	D-3
INSERT SUPPRESS.....	D-39	PERF.....	D-49
INSERT UNCONDITIONALLY.....	D-38	PERFORM ENTER.....	D-49
INSTRUCTION SET.....	D-10	PERFORM EXIT.....	D-50
LDCR.....	D-64	PROGRAM PARAMETERS.....	D-3
LITERAL STRING.....	D-4	PXIT.....	D-50
LOAD COMMUNICATE REPLY.....	D-64	RELATED PUBLICATIONS.....	D-1
MAKE PRESENT.....	D-66	S-INSTRUCTION FORMAT.....	D-3
MAKP.....	D-66	S-LANGUAGE PROGRAMS.....	D-1
MICE.....	D-44	S-OPERATORS.....	D-3
MICF.....	D-41	SCALED MOVE NUMERIC.....	D-29
MICR EDIT.....	D-44	SEGMENT NUMBER.....	D-6
MICR FORMAT.....	D-41	SET OVERFLOW TOGGLE.....	D-48
MISCELLANEOUS.....	D-11	SKIP REVERSE DESTINATION.....	D-38
MISCELLANEOUS		SMVN.....	D-29
INSTRUCTIONS.....	D-63	SOFL.....	D-48
MOVE ALPHANUMERIC.....	D-24	SPECIAL REGISTERS	
MOVE CHARACTER.....	D-38	(TABLE 1-2).....	D-2
MOVE DIGIT.....	D-37	START ZERO SUPPRESS.....	D-40
MOVE NUMERIC.....	D-26	SUB.....	D-15
MOVE SPACES.....	D-25	SUBSCRIPT FACTORS.....	D-7
MOVE SUPPRESS.....	D-38	SUBSCRIPT-OR-INDEX-FLAG.....	D-6
MOVE TRANSLATE.....	D-30	SUBTRACT THREE ADDRESS.....	D-15
MOVE ZEROS.....	D-27	SUBTRACT TWO ADDRESS.....	D-17
MULT.....	D-18	TABLE BOUND.....	D-8
MULTIPLY	D-18	XIT.....	D-52

APPENDIX **E**

COBOL GRAPHICS

Table E-1 is a listing of pertinent graphics, the related EBCDIC and ASCII internal codes, the EBCDIC 80-column card code and the BCD 96-column card codes.

Standard convention is followed for the 8-bit internal codes, which are presented as pairs of hexadecimal numbers that are translated as shown in these examples:

	<u>8</u> <u>4</u> <u>2</u> <u>1</u>	<u>8</u> <u>4</u> <u>2</u> <u>1</u>
39 =	0 0 1 1	1 0 0 1
BE =	1 0 1 1	1 1 1 0
OF =	0 0 0 0	1 1 1 1

The 80-column card code is shown in terms of the card positions punched (12,11,0,9,...,1).

The 96-column card code is presented as 6-position zone and numeric punches (BA8421), with the punched positions represented by their symbol and the non-punched positions represented by dashes.

Table E-1. Graphics and Related Codes

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
NULL	00	12-0-9-8-1	00	
SOH	01	12-9-1	01	
STX	02	12-9-2	02	
ETX	03	12-9-3	03	
	04	12-9-4		
HT	05	12-9-5	09	
	06	12-9-6		
DEL	07	12-9-7	7F	
	08	12-9-8		
	09	12-9-8-1		
	0A	12-9-8-2		
VT	0B	12-9-8-3	0B	
FF	0C	12-9-8-4	0C	
CR	0D	12-9-8-5	0D	
SO	0E	12-9-8-6	0E	
SI	0F	12-9-8-7	0F	
DLE	10	12-11-9-8-1	10	
DC1	11	11-9-1	11	
DC2	12	11-9-2	12	
DC3	13	11-9-3	13	
	14	11-9-4		
NL	15	11-9-5		
BS	16	11-9-6	08	
	17	11-9-7		
CAN	18	11-9-8	18	
EM	19	11-9-8-1	19	
	1A	11-9-8-2		
	1B	11-9-8-3		
FS	1C	11-9-8-4	1C	
GS	1D	11-9-8-5	1D	
RS	1E	11-9-8-6	1E	

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD	
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE	
US	1F	11-9-8-7	1F		
	20	11-0-9-8-1			
	21	0-9-1			
	22	0-9-2			
	23	0-9-3			
	24	0-9-4			
	LF	0-9-5			0A
	ETB	0-9-6			17
	ESC	0-9-7			1B
	28	0-9-8			
29	0-9-8-1				
2A	0-9-8-2				
2B	0-9-8-3				
2C	0-9-8-4				
ENQ	2D	0-9-8-5	05		
ACK	2E	0-9-8-6	06		
BEL	2F	0-9-8-7	07		
30	12-11-0-9-8-1				
31	9-1				
SYN	32	9-2	16		
	33	9-3			
	34	9-4			
	35	9-5			
	36	9-6			
EOT	37	9-7	04		
	38	9-8			
	39	9-8-1			
	3A	9-8-2			
	3B	9-8-3			
DC4	3C	9-8-4	14		
NAK	3D	9-8-5	15		
	3E	9-8-6			
SUB	3F	9-8-7	1A		

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
SPACE	40		20	
	41	12-0-9-1		
	42	12-0-9-2		
	43	12-0-9-3		
	44	12-0-9-4		
	45	12-0-9-5		
	46	12-0-9-6		
	47	12-0-9-7		
	48	12-0-9-8		
	49	12-8-1		
[4A	12-8-2	5B	BA-42-
.	4B	12-8-3	2E	BA8-21
<	4C	12-8-4	3C	BA84--
(4D	12-8-5	28	BA84-1
+	4E	12-8-6	2B	BA842-
!	4F	12-8-7	21	BA8421
&	50	12	26	-A8-2-
	51	12-11-9-1		
	52	12-11-9-2		
	53	12-11-9-3		
	54	12-11-9-4		
	55	12-11-9-5		
	56	12-11-9-6		
	57	12-11-9-7		
	58	12-11-9-8		
	59	11-8-1		
]	5A	11-8-2	5D	B-8-2-
\$	5B	11-8-3	24	B-8-21
*	5C	11-8-4	2A	B-84--
)	5D	11-8-5	29	B-84-1
;	5E	11-8-6	3B	B-842-
-	5F	11-8-7	5E	B-8421
-	60	11	2D	B-----

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
/	61	0-1	2F	-A---1
	62	11-0-9-2		
	63	11-0-9-3		
	64	11-0-9-4		
	65	11-0-9-5		
	66	11-0-9-6		
	67	11-0-9-7		
	68	11-0-9-8		
	69	0-8-1		
	6A	12-11	7C	
,	6B	0-8-3	2C	-A8-21
%	6C	0-8-4	25	-A84--
Underscore	6D	0-8-5	5F	-A84-1
>	6E	0-8-6	3E	-A842-
?	6F	0-8-7	3F	-A8421
	70	12-11-0		
	71	12-11-0-9-1		
	72	12-11-0-9-2		
	73	12-11-0-9-3		
	74	12-11-0-9-4		
	75	12-11-0-9-5		
	76	12-11-0-9-6		
	77	12-11-0-9-7		
	78	12-11-0-9-8		
	79	8-1	60	
:	7A	8-2	3A	--8-2-
#	7B	8-3	23	--8-21
@	7C	8-4	40	--84--
'	7D	8-5	27	--84-1
=	7E	8-6	3D	--842-
"	7F	8-7	22	--8421
	80	12-0-8-1		
a	81	12-0-1	61	
b	82	12-0-2	62	

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
c	83	12-0-3	63	
d	84	12-0-4	64	
e	85	12-0-5	65	
f	86	12-0-6	66	
g	87	12-0-7	67	
h	88	12-0-8	68	
i	89	12-0-9	69	
	8A	12-0-8-2		
	8B	12-0-8-3		
	8C	12-0-8-4		
	8D	12-0-8-5		
	8E	12-0-8-6		
	8F	12-0-8-7		
	90	12-11-8-1		
j	91	12-11-1	6A	
k	92	12-11-2	6B	
l	93	12-11-3	6C	
m	94	12-11-4	6D	
n	95	12-11-5	6E	
o	96	12-11-6	6F	
p	97	12-11-7	70	
q	98	12-11-8	71	
r	99	12-11-9	72	
	9A	12-11-8-2		
	9B	12-11-8-3		
	9C	12-11-8-4		
	9D	12-11-8-5		
	9E	12-11-8-6		
	9F	12-11-8-7		
	A0	11-0-8-1		
	A1	11-0-1	7E	
s	A2	11-0-2	73	
t	A3	11-0-3	74	
u	A4	11-0-4	75	

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
v	A5	11-0-5	76	
w	A6	11-0-6	77	
x	A7	11-0-7	78	
y	A8	11-0-8	79	
z	A9	11-0-9	7A	
	AA	11-0-8-2		
	AB	11-0-8-3		
	AC	11-0-8-4		
	AD	11-0-8-5		
	AE	11-0-8-6		
	AF	11-0-8-7		
	B0	12-11-0-8-1		
	B1	12-11-0-1		
	B2	12-11-0-2		
	B3	12-11-0-3		
	B4	12-11-0-4		
	B5	12-11-0-5		
	B6	12-11-0-6		
	B7	12-11-0-7		
	B8	12-11-0-8		
	B9	12-11-0-9		
	BA	12-11-0-8-2		
	BB	12-11-0-8-3		
	BC	12-11-0-8-4		
	BD	12-11-0-8-5		
	BE	12-11-0-8-6		
	BF	12-11-0-8-7		
(+)PZ	C0	12-0	7B	
A	C1	12-1	41	BA---1
B	C2	12-2	42	BA--2-
C	C3	12-3	43	BA--21
D	C4	12-4	44	BA-4--
E	C5	12-5	45	BA-4-1
F	C6	12-6	46	BA-42-

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
G	C7	12-7	47	BA-421
H	C8	12-8	48	BA8---
I	C9	12-9	49	BA8--1
	CA	12-0-9-8-2		
	CB	12-0-9-8-3		
	CC	12-0-9-8-4		
	CD	12-0-9-8-5		
	CE	12-0-9-8-6		
	CF	12-0-9-8-7		
(!)MZ	D0	11-0	7D	
J	D1	11-1	4A	B----1
K	D2	11-2	4B	B---2-
L	D3	11-3	4C	B--21
M	D4	11-4	4D	B-4--
N	D5	11-5	4E	B-4-1
O	D6	11-6	4F	B-42-
P	D7	11-7	50	B-421
Q	D8	11-8	51	B-8---
R	D9	11-9	52	B-8--1
	DA	12-11-9-8-2		
	DB	12-11-9-8-3		
	DC	12-11-9-8-4		
	DD	12-11-9-8-5		
	DE	12-11-9-8-6		
	DF	12-11-9-8-7		
	E0	0-8-2	5C	
	E1	11-0-9-1		
S	E2	0-2	53	-A--2-
T	E3	0-3	54	-A--21
U	E4	0-4	55	-A-4--
V	E5	0-5	56	-A-4-1
W	E6	0-6	57	-A-42-
X	E7	0-7	58	-A-421
Y	E8	0-8	59	-A8---

Table E-1. Graphics and Related Codes (Cont)

EBCDIC			USASCII-7	BCD
GRAPHIC	8-BIT INTERNAL CODE	80-COLUMN CARD CODE	8-BIT INTERNAL CODE	96-COLUMN CARD CODE
Z	E9	0-9	5A	-A8--1
	EA	11-0-9-8-2		
	EB	11-0-9-8-3		
	EC	11-0-9-8-4		
	ED	11-0-9-8-5		
	EE	11-0-9-8-6		
	EF	11-0-9-8-7		
0	F0	0	20	-A----
1	F1	1	31	-----1
2	F2	2	32	-----2-
3	F3	3	33	-----21
4	F4	4	34	----4--
5	F5	5	35	----4-1
6	F6	6	36	---42-
7	F7	7	37	---421
8	F8	8	38	--8---
9	F9	9	39	--8--1
	FA	12-11-0-9-8-2		
	FB	12-11-0-9-8-3		
	FC	12-11-0-9-8-4		
	FD	12-11-0-9-8-5		
	FE	12-11-0-9-8-6		
	FF	12-11-0-9-8-7		

INDEX

- abbreviated compound conditions, 7-24
- ACCEPT, 7-31
- ACCESS, 5-9, 5-12, 5-13, 13-1, 13-8, 13-9
 - RANDOM, 5-9, 5-12, 5-13, 13-1, 13-8, 13-9
 - SEQUENTIAL, 5-9, 5-12, 5-13, 13-1, 13-8, 13-9
 - DYNAMIC, 13-1, 13-8, 13-9
- ACTUAL KEY, 5-9, 5-12, 5-13
- ADD, 7-32
- ALL, 2-7, 4-2, 4-3, 7-50, 7-83
- ALL-AT-OPEN, 5-9, 5-11
- alphabetic items, 6-49
- alphanumeric edited items, 6-49
- alphanumeric items, 6-49
- ALTER, 7-35, 7-53
- ALTERNATE, 5-9, 5-11, 5-15, 5-16
- APPLY, 8-5
- AREA, 5-9, 5-11, 5-15, 5-16
- arithmetic expressions, 7-14
 - formation and evaluation rules, 7-14
- arithmetic operators, 7-14
- arithmetic verbs, 7-29
 - ADD, 7-32
 - COMPUTE, 7-41
 - DIVIDE, 7-47
 - MULTIPLY, 7-66
 - SUBTRACT, 7-96
- ASCENDING, 6-33, 6-43, 6-46, 7-90
- ASCII, 6-16, 6-32, 6-68
- ASSIGN, 5-9, 5-10, 5-11, 13-8, 13-9
- ASSIGN, READER-SORTER, 8-4
- AT END, 5-12, 7-78, 9-1, 13-2, 13-4, 13-5
- AUTHOR, 4-1

- BACKUP, 5-9, 5-10
- BATCH-COUNT, 8-8, 8-9
- BLANK WHEN ZERO, 6-33, 6-34, 6-35, 6-52
- BLOCK, 6-16, 6-17, 6-19
- braces, 2-10
- brackets, 2-10

- character set, 2-1
- characters, 2-1
 - editing, 2-2
 - formulas, 2-2
 - MICR, 8-1
 - relational, 2-2
 - word, 2-1
 - punctuation, 2-2
- class conditions, 7-20
- CLOSE, 7-36, 8-7, 9-1, 13-11, 13-12, 13-13
- COBOL compiler control, 11-1, 13-6
- COBOL source program syntax, 12-1
- coding form, 3-1
- COMMA, 5-6
- comparison of operands, 7-19
- compilation card deck, 11-1
- compile card, 11-2
- compiler-directing sentence, 7-4
- compiler-directing statement, 7-3
- compiler-directing verbs, 7-29
 - COPY, 7-42
 - MONITOR, 4-2
 - NOTE, 7-67
 - USE, 7-98, 8-15, 13-22
- compound conditions, 7-22
- COMPUTATIONAL, 6-32, 6-67
- COMPUTATIONAL-1, 6-32
- COMPUTATIONAL-3, 6-32, 6-67
- COMPUTE, 7-41
- concepts, 6-3
 - file, 6-3
 - record, 6-4
 - level numbers, 6-5
- condition-name, 2-4, 6-33, 6-36, 6-72, 7-21
- condition-name condition, 7-21
- conditional sentence, 7-4
- conditional statement, 7-3
- conditional verb, 7-29
 - IF, 7-3, 7-5, 7-55
- conditions, 7-17
 - abbreviated compound, 7-24
 - class, 7-20
 - compound, 7-22
 - condition-name, 7-21
 - evaluation rules, 7-21
 - relation, 7-17
 - sign, 7-20
 - simple, 7-22
- CONFIGURATION SECTION, 5-1, 5-2
- connectives, 2-9
- constant, figurative, 2-6
- continuation indicator, 3-1
- CONTROL, 8-8
- control cards, 11-1
- control relationship between procedures, 7-7
- COPY, 5-3, 5-4, 5-6, 5-9, 5-15, 6-16, 6-17, 6-32, 7-13, 7-42
- CORE-INDEX, 13-10, 13-11
- CORRESPONDING, 7-27, 7-32, 7-34, 7-61, 7-96

INDEX (Cont)

CURRENCY SIGN, 5-6, 6-53
 Current Record Pointer, 13-2

DATA-BASE SECTION, 12-1
 data classes of, 6-49
 data communications, 9-1
 DATA DIVISION, 6-1, 8-5, 12-1, 13-10
 DATA DIVISION, structure, 6-2
 DATA MANAGEMENT, 12-1
 data manipulation, verbs, 7-29
 EXAMINE, 7-50
 FORMAT, 8-10, D-41
 MICR-EDIT, 8-12, D-44
 MOVE, 7-61
 data-name, 2-3, 6-39
 DATA RECORDS, 6-16, 6-17, 6-21
 DATA SEGMENT-LIMIT, 5-4
 DATE, julian, 2-8
 DATE-COMPILED, 4-1, 4-2
 DATE-WRITTEN, 4-1
 debugging verbs, 7-30
 DUMP, 7-49
 TRACE, 7-97
 DECIMAL-POINT, 5-6, 6-52
 DECLARATIVES, 7-1, 7-2, 7-7, 7-13, 8-6
 definition of words, 2-3
 DEPENDING, 4-2, 6-32, 6-43, 6-44, 7-53
 DESCENDING, 6-33, 6-43, 6-46
 DISPLAY, 6-32, 6-52, 6-66, 7-46
 DIVIDE, 7-47
 DIVISIONS, 1-2
 DATA, 1-2, 6-1, 8-5, 12-1, 13-10
 ENVIRONMENT, 1-2, 5-1, 8-4, 13-8
 IDENTIFICATION, 1-2, 4-1
 PROCEDURE, 1-2, 7-1, 8-6, 12-2, 13-11
 DUMP, 7-49

editing, 6-50
 floating insertion, 6-56
 fixed insertion, 6-54
 insertion, 6-54
 replacement, 6-57
 simple insertion, 6-54
 special insertion, 6-54
 suppression, 6-56
 editing characters, 2-2
 editing rules, 6-54
 editing symbols, 6-50
 elementary items, 6-5
 elementary MOVE, 7-61
 ellipsis, 2-11
 END, 5-9, 5-12
 ending verb, 7-29

STOP, 7-95
 END-OF-JOB, 7-2, 7-7, A-3, B-8
 END-OF-PAGE, 7-103
 ENVIRONMENT DIVISION, 5-1, 8-4, 12-1, 13-8
 ENVIRONMENT DIVISION, structure, 5-1
 evaluation of conditions, 7-21
 EXAMINE, 7-50
 execution of PROCEDURE DIVISION, 7-2
 execution, sentence, 7-5
 EXIT, 7-52
 EXTEND, 7-68, B-13

FD, 6-16, 8-5, 13-10
 figurative constant, 2-8
 file concept, 6-3
 FILE-CONTROL, 5-1, 5-9, 13-8
 FILE-CONTROL, READER-SORTER, 8-4
 file description, 6-2, 6-16, 13-10
 FILE-LIMIT, 5-9, 5-11, 5-12
 file-name, 2-3
 FILE SECTION, 6-1, 6-2, 6-16
 FILLER, 6-32, 6-39
 fixed insertion editing, 6-54
 floating insertion editing, 6-56
 FORM, 5-9, 5-11
 FORMAT, 8-10, D-41
 FROM, 7-81, 7-103, 7-104, 13-24

generic terms, 2-10
 GIVING, 7-32, 7-33, 7-47, 7-56, 7-59, 7-66, 7-90, 7-93, 7-94, 7-96
 GO, 7-35
 group items, 6-5
 group MOVE, 7-63

HEADERS, 6-22, 6-23
 HIGH-VALUE(s), 2-6

IDENTIFICATION DIVISION, 4-1, 12-1
 IDENTIFICATION DIVISION, structure, 4-1
 identification field, 3-3
 identifier, 6-15, 7-1
 IF, 7-55
 imperative sentence, 7-4
 imperative statement, 7-3
 INDEX, 6-32
 index data items, MOVE, 7-64
 index-name, 2-4, 6-15, 7-90
 INDEXED BY, 6-14, 6-33, 6-43, 6-46
 indexing, 6-14, 6-15
 initial value, 6-72
 INPLACE, 7-90
 INPUT, OPEN, 7-68, 8-13, 13-14

INDEX (Cont)

INPUT-OUTPUT, OPEN, 7-68, 13-14
 INPUT-OUTPUT SECTION, 5-1, 5-8
 INPUT PROCEDURE, 7-90
 input-output verbs, 7-29
 ACCEPT, 7-31
 CLOSE, 7-36, 8-7, 13-11, 13-12
 CONTROL, 8-8
 DISPLAY, 7-46
 OPEN, 7-68, 8-13, 13-13, 13-14
 READ, 7-78, 8-14, 13-16
 REWRITE, 13-19
 SEEK, 7-87
 START, 13-21
 WRITE, 7-103, 13-21, 13-23, 13-24
 insertion editing, 6-54
 INSTALLATION, 4-1
 internal program switches, 7-26
 inter-program communication, 10-1
 INTERPRET, OPEN, 7-68
 INTO, 7-78, 7-79, 8-14, 13-16, 13-17
 INVALID KEY, 5-12, 7-78, 7-103, 13-4,
 13-16, 13-19, 13-21, 13-24
 items, 6-48
 alphabetic, 6-49
 alphanumeric, 6-49
 alphanumeric edited, 6-49
 numeric, 6-49
 numeric edited, 6-49
 I-O-CONTROL, 5-1, 5-15, 6-27, 8-5
 I-O Status, 13-2

 JUSTIFIED, 6-33, 6-34, 6-40

 key words, 2-9, 2-10

 LABEL, 6-16, 6-24
 label equation card, 5-14, 11-1
 language description notation, 2-10
 language formation, 2-1
 LEADING, 7-50
 level-number, 6-42
 level-number concept, 6-5
 library, 3-1, 7-42, 7-43, 7-44, 7-45
 literals, 2-4
 numeric, 2-4
 non-numeric, 2-5
 undigit, 2-6
 LOCK ACCESS, OPEN, 7-68, 7-71, 13-14
 LOCK, CLOSE, 7-36, 13-12
 LOCK, OPEN, 7-68, 7-71, 13-14
 logical control verbs, 7-29
 IF, 7-55
 logical operators, 7-17
 logical record, 6-3
 LOW-VALUE(s), 2-6

 margin A, 3-3
 margin B, 3-3
 MCP label card, 11-2
 MEMORY, 5-4
 MERGE, 7-56
 MESSAGE QUEUES, 10-1
 MICR character type, 8-21
 MICR-EDIT, 8-12, D-44
 MICR RESULT DESCRIPTOR, 8-5, 8-14,
 8-16
 OPERATION-COMPLETE, 8-16
 EXCEPTION-CONDITION, 8-16
 NOT-READY, 8-2, 8-17
 UNENCODED-DOCUMENT, 8-17
 CANNOT-READ, 8-17
 DOUBLE-DOCUMENT, 8-3, 8-18
 TOO-LATE-TO-READ, 8-2, 8-18
 JAM, 8-2, 8-18
 MISSORT, 8-2, 8-19
 BLACK-BAND, 8-2, 8-3, 8-19
 HALT-FLOW-STOPPED, 8-8, 8-20
 TOO-LATE-TO-POCKET-SELECT, 8-2,
 8-3, 8-20
 POCKET-NUMBER-SELECTED, 8-21, 8-22
 mnemonic-name, 2-4
 MOD, 7-47, 7-48
 MONITOR, 4-2
 MOVE, 7-61
 elementary, 7-61
 group, 7-63
 index data items, 7-64
 MOVE, valid statement combinations,
 7-58
 MSG-TYPE, 5-13
 MULTIPLE, 5-9, 5-15
 MULTIPLE FILE, 5-15, 5-16, 5-17
 MULTIPLE REEL, 5-9, 5-11
 MULTIPLY, 7-66

 NEXT SENTENCE, 7-55
 NO, 5-9, 5-11
 NO REWIND, CLOSE, 7-36
 NO REWIND, OPEN, 7-68
 non-contiguous WORKING-STORAGE, 7-78
 non-numeric literal, 2-5
 NON-STANDARD, 6-16, 6-26
 NOTE, 7-67
 nouns, 2-3
 condition-name, 2-4
 data-name, 2-3
 figurative constant, 2-6
 file-name, 2-3
 index-name, 2-4
 literals, 2-4
 mnemonic-name, 2-4

INDEX (Cont)

- procedure-name, 2-4
- record-name, 2-3
- special registers, 2-7
- numeric edited items, 6-49
- numeric items, 6-49
- numeric literal, 2-4

- OBJECT-COMPUTER, 5-1, 5-4
- object program, 1-3
- OCCURS, 6-32, 6-43
- OCR, 8-1
- O-I, OPEN, 7-68
- OMITTED, 6-16, 6-24
- OPEN, 5-10, 5-16, 7-68, 8-13, 13-14
- optional control card, 11-3
- OPTIONAL, 5-9, 5-10, 5-17, 7-79
- optional words, 2-9, 2-10
- OUTPUT, OPEN, 7-68, 13-14
- OUTPUT PROCEDURE, 7-90

- paragraph, definition, 7-1
- structure, 7-7
- paragraph NOTE, 7-67
- PERFORM, 7-72
- period, 2-11
- Physical Files, 13-6
- physical record, 6-3
- PICTURE, 5-7, 6-32, 6-48
- precedence, 6-58
- POSITION, 5-15, 5-17
- precedence, 6-58
- PICTURE, 6-58
- PRINT128, OPEN, 7-68
- priority number, 7-10
- procedure branching verbs, 7-30
- ALTER, 7-35
- EXIT, 7-52
- GO, 7-53
- PERFORM, 7-72
- ZIP, 7-106
- PROCEDURE DIVISION, 7-1, 8-6, 12-1, 13-11
- PROCEDURE DIVISION, body, 7-2
- PROCEDURE DIVISION, execution, 7-2
- PROCEDURE DIVISION, READER-SORTER, 8-6
- procedure formation, rules of, 7-1
- procedure-name, 2-4, 7-1
- POCKET, 8-8
- POCKET-LIGHT, 8-8, 8-15
- PROGRAM-ID, 4-1
- program organization, 1-2
- program segments, 7-9
- PUNCH, OPEN, 7-68
- punctuation, 3-4
- punctuation characters, 2-2
- punctuation, sentence, 7-5
- PURGE, CLOSE, 7-36, 13-12

- qualification, 6-8
- qualifier, 2-9
- Q-EMPTY, 7-99, 10-3
- Q-FULL, 7-99, 10-3
- Q-MAX-MESSAGES, 6-16, 6-31, 10-2, 10-3
- QUEUE, 6-16
- QUEUE, files, 10-1, 10-3

- RANDOM, 5-12
- READ, 7-78, 8-14, 13-16
- READ-OK, 7-100
- READER-SORTER, 8-1
- ENVIRONMENT DIVISION, 8-4
- DATA DIVISION, 8-5
- PROCEDURE DIVISION, 8-6
- RECORD, 5-15, 5-16, 6-16, 6-17, 6-25
- record concept, 6-4
- record description, 6-2, 6-32
- record-name, 2-3
- RECORDING, 6-16, 6-26
- REDEFINES, 6-32, 6-33, 6-62
- REEL, CLOSE, 7-36, 7-40
- relation characters, 2-2
- relation condition, 7-17
- relational operators, 7-19
- RELEASE, 7-81
- RELEASE, CLOSE, 7-36, 13-12
- REMARKS, 4-1
- REMOVE, CLOSE, 7-36, 13-12
- RENAMES, 6-33, 6-42, 6-64
- replacement editing, 6-57
- REPLACING, 5-3, 5-4, 5-6, 5-8, 5-15, 6-16, 6-17, 7-34, 7-42, 7-43, 7-50
- RESERVE, 5-9, 5-11
- reserved words, 2-9
- RESTART, 7-90
- RETURN, 7-82
- REVERSED, OPEN, 7-68
- REWRITE, 13-19
- right margin, 3-3
- Rough Tables, 13-7
- ROUNDED, 7-32, 7-33, 7-41, 7-47, 7-66, 7-96

- SAME, 5-15, 5-16, 5-17
- SAVE-FACTOR, 6-16, 6-27, 6-30
- SD, 6-17, 7-90
- SEARCH, 7-83
- SECTION, 7-1, 7-7
- CONFIGURATION, 5-2

INDEX (Cont)

- definition, 7-1
- FILE, 6-16
- INPUT-OUTPUT, 5-8
- structure, 7-7
- WORKING-STORAGE, 6-72
- SECURITY, 4-1
- SEEK, 7-87
- segmentation, 7-9
- segment classification, 7-9
- SEGMENT-LIMIT, 5-4, 7-9
- SELECT, 5-9, 5-10, 13-8
- SELECT, READER-SORTER, 8-4
- sentence, definition, 7-1
- sentence, 7-1, 7-4
 - compiler-directing, 7-4
 - conditional, 7-4
 - imperative, 7-4
- sentence NOTE, 7-67
- sentence punctuation, 7-5
- sequence field, 3-1
- SEQUENTIAL, 5-12, 5-13, 13-8
- SET, 7-88
- sign condition, 7-20
- simple conditions, 7-22
- simple insertion editing, 6-54
- SINGLE, 5-9, 5-11
- SIZE ERROR, 7-32, 7-33, 7-41, 7-47, 7-66, 7-96, 8-10
- SORT, 5-4, 5-10, 5-11, 5-14, 7-90
- sort verbs, 7-30
 - RELEASE, 7-81
 - RETURN, 7-82
 - SORT, 7-90
- SOURCE-COMPUTER, 5-1, 5-3
- source data card, 11-6
- source program, 1-3
- special insertion editing, 6-54
- SPECIAL-NAMES, 5-1, 5-6
- special registers, 2-7
 - DATE, julian, 2-8
 - TALLY, 2-7
 - TIME, 2-8
 - TODAYS-DATE, 2-8
 - TODAYS-NAME, 2-8
- SPO-INPUT-PRESENT, 7-100
- STANDARD, 6-16, 6-24, 6-26
- START, 13-21
- statement, 7-1, 7-3
 - compiler-directing, 7-3
 - conditional, 7-3
 - imperative, 7-3
- STATION, 6-22
- STATION-RSN, 5-13
- STATUS KEY, 13-2, 13-3, 13-4
- STOP, 7-95
- STOP-FLOW, 8-8, 8-20
- STOP RUN, 7-3, 7-95
- SUBTRACT, 7-96
- subscripting, 6-12
- suppression editing, 6-56
- switches, internal program, 7-26
- SYNCHRONIZED, 6-33, 6-34
- tables, 6-11, 6-43
- table manipulation verbs, 7-30
 - SEARCH, 7-83
 - SET, 7-88
- Tag Files, 13-5
- TAG-KEY, 7-90
- TAG-SEARCH, 7-90
- TALLY, 2-7
- TALLYING, 7-50
- TAPES, 5-10, 5-14
- TEXT-LENGTH, 5-13
- THROUGH, 6-33, 6-36
- TIME, 2-8
- timing requirements, READER-SORTER, 8-2
- TODAYS-DATE, 2-8
- TODAYS-NAME, 2-8
- TRACE, 7-97
- translation of data, MOVE, 7-63
- types of words, 2-3
 - nouns, 2-3
 - verbs, 2-8
 - reserved, 2-9
- undigit literal, 2-6
- UNTIL FIRST, 7-50
- USAGE, 6-32, 6-66
- USE declarative, 7-13, 7-98, 8-15, 9-2, 13-22
- VALUE, 6-16, 6-24, 6-27, 6-33, 6-36, 6-69
- verb formats, data communication, 9-1
- verbs, 2-8, 7-29
- verbs, arithmetic, 7-29
 - ADD, 7-32
 - COMPUTE, 7-41
 - DIVIDE, 7-47
 - MULTIPLY, 7-66
 - SUBTRACT, 7-96
- verbs, compiler-directing, 7-29
 - COPY, 7-42
 - MONITOR, 4-2
 - NOTE, 7-67
 - USE, 7-98, 8-15, 9-2, 13-22
- verbs, conditional, 7-29
 - IF, 7-55
- verbs, data manipulation, 7-29
 - EXAMINE, 7-50

INDEX (Cont)

- FORMAT, 8-10, D-41
- MICR-EDIT, 8-12, D-44
- MOVE, 7-61
- verbs, debugging, 7-30
 - DUMP, 7-49
 - TRACE, 7-97
- verbs, ending, 7-29
 - STOP, 7-95
- verbs, input-output, 7-29
 - ACCEPT, 7-31
 - CLOSE, 7-36, 8-71, 13-11, 13-12
 - CONTROL, 8-8
 - DISPLAY, 7-46
 - OPEN, 7-68, 8-13, 13-13, 13-14
 - READ, 7-78, 8-14, 13-16
 - REWRITE, 13-19
 - SEEK, 7-87
 - START, 13-21
 - WRITE, 7-103, 13-21, 13-23, 13-24
- verbs, logic control, 7-29
 - IF, 7-55
- verbs, procedure branching, 7-30
 - ALTER, 7-35
 - EXIT, 7-52
 - GO, 7-53
 - PERFORM, 7-72
- ZIP, 7-106
- verbs, SORT, 7-30
 - RELEASE, 7-81
 - RETURN, 7-82
 - SORT, 7-90
- verbs, table manipulation, 7-30
 - SEARCH, 7-83
 - SET, 7-88
- words, 2-3
 - definition, 2-3
 - key, 2-9
 - optional, 2-9
 - reserved, 2-9
 - types, 2-3
 - nouns, 2-3
 - reserved, 2-9
 - verbs, 2-8, 7-29
- WAIT, 7-100
- WORK, 5-9, 5-11
- WORKING-STORAGE, 6-1, 6-2, 6-42, 6-71
- WRITE, 7-103, 13-21, 13-23, 13-24
- WRITE-OK, 7-100
- ZIP, 7-106

B 1800/B 1700 Systems COBOL
REFERENCE MANUAL

1057197

Printed in U.S.A.

