

Cromemco
COBOL
Instruction Manual
and User's Guide

Twenty Dollars

Cromemco

COBOL

Instruction Manual and User's Guide

Copyright © 1978 by Cromemco Inc. All rights reserved.





Acknowledgment

Any organization interested in reproducing the COBOL report and specifications in whole or in 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 part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention, 'COBOL' in acknowledgment of the source, but need not quote this entire section.

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

No warranty, expressed or implied, is made by any contributor or by the 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 procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (Trademark of Sperry Rand Corporation),
Programming for the UNIVAC (R) I and II, Data Automation
Systems copyrighted 1958, 1959, by Sperry Rand Corporation;
IBM Commercial Translator, Form No. F28-8013, copyrighted
1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by
Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specification in programming manuals or similar publications.

--from the ANSI COBOL STANDARD
(X3.23-1974)

CONTENTS

PAGE

CROMEMCO COBOL REFERENCE MANUAL

Introduction	6
CHAPTER 1: Fundamental Concepts of COBOL	8
1.1 Character Set	8
1.2 Punctuation	9
1.3 Word Formation	10
1.4 Format Notation	10
1.5 Level Numbers and Data-Names	12
1.6 File-Names	14
1.7 Condition-Names	15
1.8 Mnemonic-Names	15
1.9 Literals	15
1.10 Figurative Constants	17
1.11 Structure of a Program	18
1.12 Coding Rules	21
1.13 Qualification of Names	22
1.14 COPY Statement	22
CHAPTER 2: Identification and Environment Divisions	23
2.1 Identification Division	23
2.2 Environment Division	24
2.2.1 Configuration Section	24
2.2.2 Input-Output Section	25
2.2.2.1 File-Control Entry	26
2.2.2.2 I-O Control Paragraph	27
CHAPTER 3: Data Division	28
3.1 Data Items	28
3.1.1 Group Items	28
3.1.2 Elementary Items	28
3.1.3 Numeric Items	29
3.2 Data Description Entry	30
3.3 Formats for Elementary Items	31
3.4 USAGE Clause	33
3.5 PICTURE Clause	34
3.6 VALUE Clause	40
3.7 REDEFINES Clause	41
3.8 OCCURS Clause	42
3.9 SYNCHRONIZED Clause	44
3.10 BLANK WHEN ZERO Clause	44
3.11 JUSTIFIED Clause	44
3.12 SIGN Clause	45
3.13 File Section, FD Entries (Sequential I-O Only)	46
3.13.1 LABEL Clause	46

3.13.2	VALUE OF Clause	46
3.13.3	DATA RECORDS Clause	47
3.13.4	BLOCK Clause	47
3.13.5	RECORD Clause	48
3.13.6	CODE-SET Clause	48
3.14	Working-Storage Section	49
3.15	Linkage Section	49
3.16	Level 88 Condition Names	49
CHAPTER 4: Procedure Division		51
4.1	Statements, Sentences, Procedures-Names	51
4.2	Organization of the Procedure Division	52
4.3	MOVE Statement	53
4.4	INSPECT Statement	55
4.5	Arithmetic Statements	58
4.5.1	SIZE ERROR Option	59
4.5.2	ROUNDED Option	59
4.5.3	GIVING Option	60
4.5.4	ADD Statement	60
4.5.5	SUBTRACT Statement	61
4.5.6	MULTIPLY Statement	62
4.5.7	DIVIDE Statement	62
4.5.8	COMPUTE Statement	63
4.6	GO TO Statement	65
4.7	STOP Statement	65
4.8	ACCEPT Statement	66
4.9	DISPLAY Statement	66
4.10	PERFORM Statement	67
4.11	EXIT Statement	68
4.12	ALTER Statement	69
4.13	IF Statement	69
4.13.1	Conditions	70
4.14	OPEN Statement (Sequential I-O)	73
4.15	READ Statement (Sequential I-O)	74
4.16	WRITE Statement (Sequential I-O)	75
4.17	CLOSE Statement (Sequential I-O)	77
4.18	REWRITE Statement (Sequential I-O)	77
4.19	General Note on I/O Error Handling	78
4.20	ACCEPT DATE/DAY/TIME	79
4.21	STRING Statement	80
4.22	UNSTRING Statement	81
4.23	Dynamic Debugging Statements	83
CHAPTER 5: Inter-Program Communication		85
5.1	USING List Appendage to Procedure Header	85
5.2	CALL Statement	86
5.3	EXIT PROGRAM Statement	86

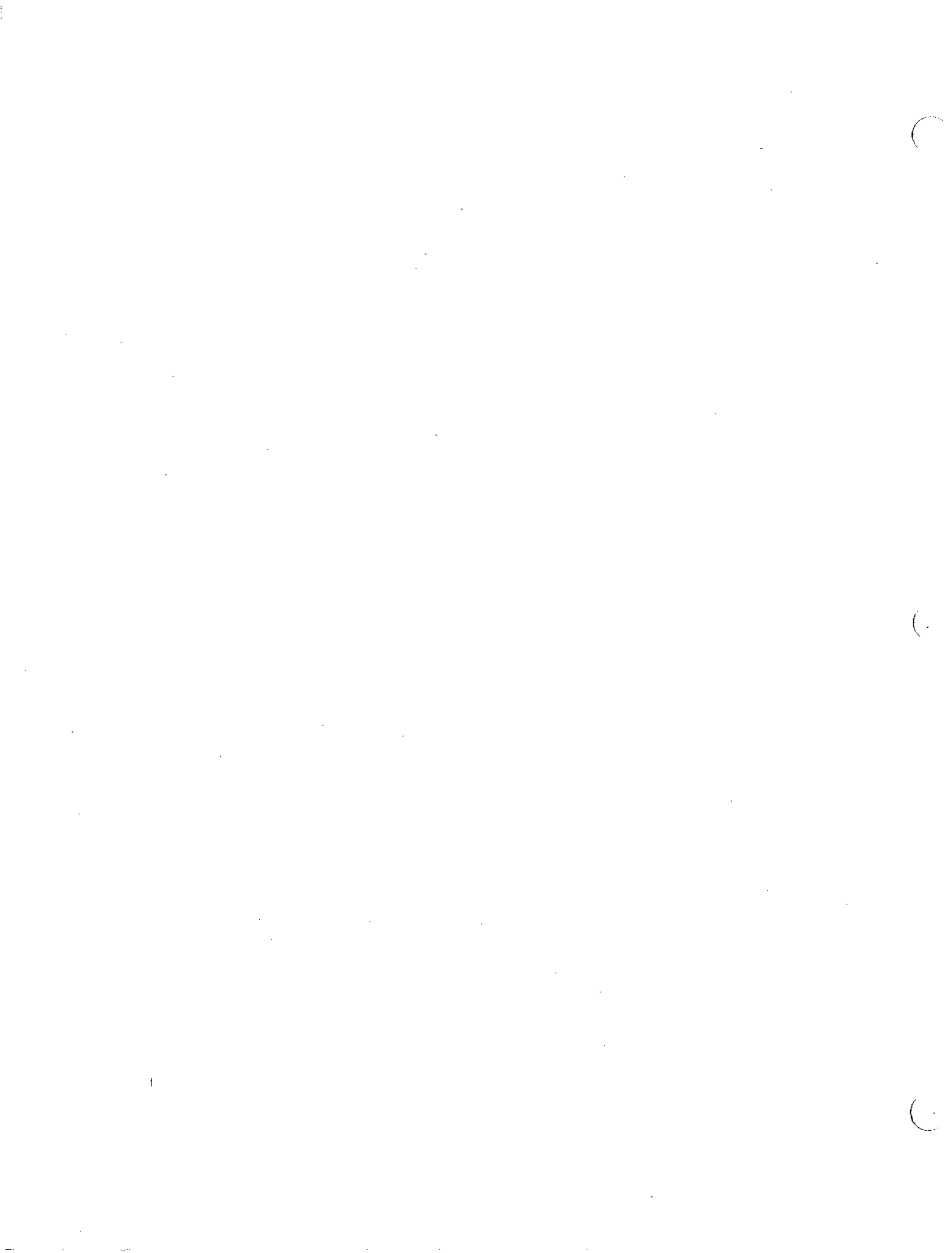
CHAPTER 6:	Table Handling by the Indexing Method . . .	87
6.1	Index-Names and Index Items	87
6.2	SET Statement	87
6.3	Relative Indexing	88
6.4	SEARCH Statement - Format 1	89
6.5	SEARCH Statement - Format 2	90
CHAPTER 7:	Indexed Files	94
7.1	Definition of Indexed File Organization	94
7.2	Syntax Considerations	94
7.2.1	RECORD KEY Clause	95
7.2.2	File Status Reporting	95
7.3	Procedure Division Statements for Indexed Files	96
7.4	READ Statement	97
7.5	WRITE Statement	98
7.6	REWRITE Statement	99
7.7	DELETE Statement	99
7.8	START Statement	100
CHAPTER 8:	Relative Files	101
8.1	Definition of Relative File Organization	101
8.2	Syntax Considerations	101
8.2.1	RELATIVE KEY Clause	102
8.3	Procedure Division Statements for Relative Files	102
8.4	READ Statement	102
8.5	WRITE Statement	103
8.6	REWRITE Statement	104
8.7	DELETE Statement	104
8.8	START Statement	105
CHAPTER 9:	DECLARATIVES and the USE Sentence	106
Appendix I:	Evaluation Rules for Compound Conditions	108
Appendix II:	Table of Permissible MOVE Operands	111
Appendix III:	Nesting of IF Statements	112
Appendix IV:	ASCII Character Set	114
Appendix V:	Reserved Word List	115
Appendix VI:	PERFORM with VARYING and AFTER Clauses	119

CROMEMCO COBOL USER'S GUIDE

SECTION 1	Compiling COBOL Programs	121
1.1	Cromemco COBOL Command Scanner	121
1.1.1	Format of Commands	121
1.1.2	Cromemco COBOL Compilation Switches	124
1.2	Output Listings and Error Messages	124
1.3	Files Used by Cromemco COBOL	126
SECTION 2	Linking and Loading COBOL Programs	127
2.1	Command Format	127
2.2	LINK Switches	128
2.2.1	E (Exit to CDOS)	128
2.2.2	G (Go - start execution)	129
2.2.3	M (Map all symbols)	129
2.2.4	R (Reset Linker)	129
2.2.5	S (Search file)	129
2.2.6	U (list all Undefined globals)	130
2.3	Format of Link-Compatible Object files	130
2.4	Link Error Messages	132
2.4.1	Fatal Errors	132
2.4.2	Warnings	133
2.5	Examples of Linking Modules	133
SECTION 3	Runtime Execution	137
3.1	Printer File Handling	137
3.2	Disk File Handling	137
3.3	Runtime Errors	138

NOTE

The last pages of this manual contain errata and supplemental notes.



Introduction

Cromemco COBOL is based upon American National Standard X3.23-1974. Elements of the COBOL language are allocated to twelve different functional processing "modules".

Each module of the COBOL Standard has two non-null "levels" --level 1 represents a subset of the full set of capabilities and features contained in level 2.

In order for a given system to be called COBOL, it must provide at least level 1 of the Nucleus, Table Handling and Sequential I-O Modules.

The following summary specifies the content of Cromemco COBOL with respect to the Standard.

<u>Module</u>	<u>Features Available</u>
Nucleus	All of level 1, plus these features of level 2:

CONDITIONS:

- Level 88 conditions with value series or range
- Use of logical AND/OR/NOT in conditions
- Use of algebraic relational symbols for equality or inequalities
- Implied subject, or both subject and relation, in relational conditions
- Sign test
- Nested IF statements; parentheses in conditions

VERBS:

- ACCEPTance of data from DATE/DAY/TIME STRING and UNSTRING statements
- COMPUTE with multiple receiving fields
- PERFORM -- all formats from standard level 2

IDENTIFIERS:

- Mnemonic-names for ACCEPT or DISPLAY devices
- Procedure-names consisting of digits only

Qualification of Names (Procedure
Division only)

Sequential, Relative and Indexed I/O	All of level 1 plus these features of level 2: RESERVE clause Multiple operands in OPEN and CLOSE, with individual options per file
Sequential I/O	EXTEND mode for OPEN
Relative and Indexed I/O	DYNAMIC access mode (with READ NEXT) START (with key relations EQUAL, GREATER, or NOT LESS)
Library	Level 1
Inter-Program Communication	Level 1
Table Handling	All of level 1, plus full level 2 formats for SEARCH statement
Debugging	Special extensions to ANS-74 standard providing convenient trace-style debugging. Conditional compilation: lines with "D in column 7" are bypassed unless <u>WITH DEBUGGING MODE</u> is given in <u>SOURCE- COMPUTER</u> paragraph.

CHAPTER 1

Fundamental Concepts of COBOL

1.1 Character Set

The COBOL source language character set consists of the following characters:

Letters A through Z
Blank or space
Digits 0 through 9
Special characters:
+ Plus sign
- Minus sign
* Asterisk
= Equal sign
> Relational sign (greater than)
< Relational sign (less than)
\$ Dollar sign
, Comma
; Semicolon
. Period or decimal point
" Quotation mark
(Left parenthesis
) Right parenthesis
' Apostrophe (alternate of quotation mark)
/ Slash

Of the previous set, the following characters are used for words:

0 through 9
A through Z
- (hyphen)

The following characters are used for punctuation:

(Left parenthesis
) Right parenthesis
, Comma
. Period
; Semicolon

The following relation characters are used in simple conditions:

>
<
=

In the case of non-numeric (quoted) literals, comment entries, and comment lines, the COBOL character set is expanded to include the computer's entire character set.

1.2 Punctuation

The following general rules of punctuation apply in writing source programs:

1. As punctuation, a period, semicolon, or comma should not be preceded by a space, but must be followed by a space.
2. At least one space must appear between two successive words and/or literals. Two or more successive spaces are treated as single space, except in non-numeric literals.
3. Relation characters should always be preceded by a space and followed by another space.
4. When the period, comma, plus, or minus characters are used in the PICTURE clause, they are governed solely by rules for report items.
5. A comma may be used as a separator between successive operands of a statement, or between two subscripts.
6. A semicolon or comma may be used to separate a series of statements or clauses.

1.3 Word Formation

User-defined and reserved words are composed of a combination of not more than 30 characters, chosen from the following set of 37 characters:

0 through 9 (digits)
A through Z (letters)
- (hyphen)

A word must begin with a letter; it may not end with a hyphen. A word is ended by a space or by proper punctuation. A word may contain more than one embedded hyphen; consecutive embedded hyphens are also permitted. All words are either reserved words, which have preassigned meanings, or programmer-supplied names. If a programmer-supplied name is not unique, there must be a unique method of reference to it by use of name qualifiers, e.g., TAX-RATE IN STATE-TABLE. Primarily, a non-reserved word identifies a data item or field and is called a data-name. Other cases of non-reserved words are file-names, condition-names, mnemonic-names, and procedure-names. (Procedure-names may begin with a digit).

1.4 Format Notation

Throughout this publication, "general formats" are prescribed for various clauses and statements to guide the programmer in writing statements. They are presented in a uniform system of notation, explained in the following paragraphs.

1. All words printed entirely in capital letters are reserved words. These are words that have preassigned meanings. In all formats, words in capital letters represent actual occurrences of those words.
2. All underlined reserved words are required unless the portion of the format containing them is itself optional. These are key words. If any key word is missing or is incorrectly spelled, it is considered an error in the program. Reserved words not underlined may be included or omitted at the option of the

programmer. These words are optional words; they are used solely for improving readability of the program.

3. The characters < > = (although not underlined) are required when such formats are used.
4. All punctuation and other special characters represent actual occurrences of those characters. Punctuation is essential where it is shown. Additional punctuation can be inserted, according to the rules for punctuation specified in Section 1.2. In general, terminal periods are shown in formats in the manual because they are required; semicolons and commas are not usually shown because they are optional. To be separators, all commas, semicolons and periods must be followed by a space (or blank).
5. Words printed in lower-case letters in formats represent generic terms (e.g., data-names) for which the user must insert a valid entry in the source program.
6. Any part of a statement or data description entry that is enclosed in brackets is optional. Parts between matching braces ({ }) represent a choice of mutually exclusive options.
7. Certain entries in the formats consist of a capitalized word(s) followed by the word "Clause" or "Statement." These designate clauses or statements that are described in other formats, in appropriate sections of the text.
8. In order to facilitate reference to lower-case words in the explanatory text, some of them are followed by a hyphen and a digit or letter. This modification does not change the syntactical definition of the word.
9. Alternate options may be explained by separating the mutually exclusive choices by a vertical stroke, e.g.:

AREA | AREAS is equivalent to $\left\{ \begin{array}{l} \text{AREA} \\ \text{AREAS} \end{array} \right\}$

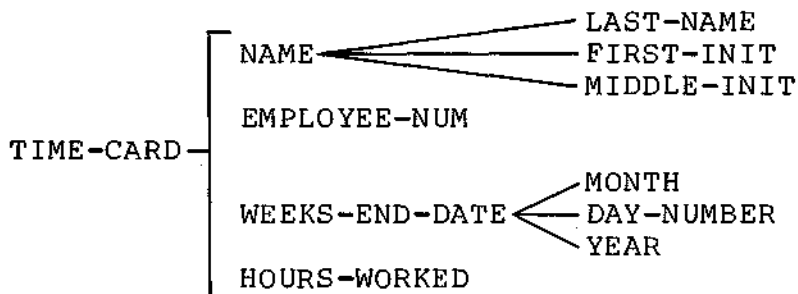
10. The ellipsis (...) indicates that the immediately preceding unit may occur once, or any number of times in succession. A unit means either a single lower-case word, or a group of lower-case words and one or more reserved words enclosed in brackets or braces. If a term is enclosed in brackets or braces, the entire unit of which it is part must be repeated when repetition is specified.
11. Optional elements may be indicated by parentheses instead of brackets, provided the lack of formality represents no substantial bar to clarity.
12. Comments, restrictions, and clarification on the use and meaning of every format are contained in the appropriate sections of this manual.

1.5 Level Numbers and Data-Names

For purposes of processing, the contents of a file are divided into logical records, with level number 01 initiating a logical record description. Subordinate data items that constitute a logical record are grouped in a hierarchy and identified with level numbers 02 to 49, not necessarily consecutive. Additionally, level number 77 identifies a "stand alone" item in Working Storage or Linkage Sections; that is, it does not have subordinate elementary items as does level 01. Level 88 is used to define condition-names and associated conditions. A level number less than 10 may be written as a single digit.

Levels allow specification of subdivisions of a record necessary for referring to data. Once a subdivision is specified, it may be further subdivided to permit more detailed data reference. This is illustrated by the following weekly timecard record, which is divided into four major items: name, employee-number, date and hours, with more specific information appearing for name and date.

CROMEMCO COBOL REFERENCE MANUAL
 Fundamental Concepts of COBOL



Subdivisions of a record that are not themselves further subdivided are called elementary items. Data items that contain subdivisions are known as group items. When a Procedure statement makes reference to a group item, the reference applies to the area reserved for the entire group. All elementary items must be described with a PICTURE or USAGE IS INDEX clause. Consecutive logical records (Ø1) subordinate to any given file represent implicit redefinitions of the same area whereas in the Working-Storage section, each record (Ø1) is the definition of its own memory area.

Less inclusive groups are assigned numerically higher level numbers. Level numbers of items within groups need not be consecutive. A group whose level is k includes all groups and elementary items described under it until a level number less than or equal to k is encountered.

Separate entries are written in the source program for each level. To illustrate level numbers and group items, the weekly timecard record in the previous example may be described (in part) by Data Division entries having the following level numbers, data-names and PICTURE definitions.

```

Ø1  TIME-CARD.
    Ø2  NAME.
        Ø3  LAST-NAME      PICTURE X(18).
        Ø3  FIRST-INIT     PICTURE X.
        Ø3  MIDDLE-INIT    PICTURE X.
    Ø2  EMPLOYEE-NUM      PICTURE 99999.
    Ø2  WEEKS-END-DATE.
        Ø5  MONTH          PIC 99.
        Ø5  DAY-NUMBER     PIC 99.
        Ø5  YEAR           PIC 99.
    Ø2  HOURS-WORKED      PICTURE 99V9.
  
```

A data-name is a word assigned by the user to identify a data item used in a program. A data-name always refers to a region of data, not to a particular value. The item referred to often assumes a number of different values during the course of a program.

A data-name must begin with an alphabetic character. A data-name or the key word FILLER must be the first word following the level number in each Record Description entry, as shown in the following general format:

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

This data-name is the defining name of the entry and is used to refer to the associated data area (containing the value of a data item).

If some of the characters in a record are not used in the processing steps of a program, then the data description of these characters need not include a data-name. In this case, FILLER is written in lieu of a data-name after the level number.

1.6 File Names

A file is a collection of data records, such as a printed listing or a region of floppy disk, containing individual records of a similar class or application. A file-name is defined by an FD entry in the Data Division's File Section. FD is a reserved word which must be followed by a unique programmer-supplied word called the file-name. Rules for composition of the file-name word are identical to those for data-names (see Section 1.3). References to a file-name appear in Procedure statements OPEN, CLOSE and READ, as well as in the Environment Division. CAUTION: File names are not to be confused with file IDs as described in section 3.13.2.

1.7 Condition-Names

A condition-name is defined in level 88 entries within the Data Division. It is a name assigned to a specific value, set or range of values, within the complete set of values that a data item may assume. Rules for formation of name words are specified in Section 1.3. Explanations of condition-name declarations and procedural statements employing them are given in the chapters devoted to Data and Procedure Divisions.

1.8 Mnemonic-Names

A mnemonic-name is assigned in the Environment Division for reference in ACCEPT or DISPLAY statements. It assigns a user-defined word to an implementor-chosen name, such as PRINTER. A mnemonic-name is composed according to the rules in Section 1.3.

1.9 Literals

A literal is a constant that is not identified by a data-name in a program, but is completely defined by its own identity. A literal is either non-numeric or numeric.

Non-Numeric Literals

A non-numeric literal must be bounded by matching quotation marks or apostrophes and may consist of any combination of characters in the ASCII set, except quotation marks or apostrophe, respectively. All spaces enclosed by the quotation marks are included as part of the literal. A non-numeric literal must not exceed 120 characters in length.

The following are examples of non-numeric literals:

"ILLEGAL CONTROL CARD"

'CHARACTER-STRING'

"DO's & DON'T'S"

Each character of a non-numeric literal (following

the introductory delimiter) may be any character other than the delimiter. That is, if the literal is bounded by apostrophes, then quotation (") marks may be within the literal, and vice versa. Length of a non-numeric literal excludes the delimiters; minimum length is one.

A succession of two "delimiters" within a literal is interpreted as a single representation of the delimiter within the literal.

Non-numeric literals may be "continued" from one line to the next. When a non-numeric literal is of a length such that it cannot be contained on one line, the following rules apply to the next line of coding (continuation line):

1. A hyphen is placed in column 7 of the continuation line.
2. A delimiter is placed in Area B preceding the continuation of the literal.
3. All spaces at the end of the previous line and any spaces following the delimiter in the continuation line and preceding the final delimiter of the literal are considered to be part of the literal.
4. On any continuation line, Area A should be blank.

Numeric Literals

A numeric literal must contain at least one and not more than 18 digits. A numeric literal may consist of the characters 0 through 9 (optionally preceded by a sign) and the decimal point. It may contain only one sign character and only one decimal point. The sign, if present, must appear as the leftmost character in the numeric literal. If a numeric literal is unsigned, it is assumed to be positive.

A decimal point may appear anywhere within the numeric literal, except as the rightmost character. If a numeric literal does not contain a decimal point, it is considered to be an integer.

The following are examples of numeric literals:

72 +1011 3.14159 -6 -.333 0.5

By use of the Environment specification DECIMAL-POINT IS COMMA, the functions of characters period and comma are interchanged, putting the "European" notation into effect. In this case, the value of "pi" would be 3,1416 when written as a numeric literal.

1.10 Figurative Constants

A figurative constant is a special type of literal. It represents a value to which a standard data-name has been assigned. A figurative constant is not bounded by quotation marks.

ZERO may be used in many places in a program as a numeric literal. Other figurative constants are available to provide non-numeric data; the reserved words representing various characters are as follows:

SPACE	the blank character represented by "octal" 40
LOW-VALUE	the character whose "octal" representation is 00
HIGH-VALUE	the character whose "octal" representation is 177
QUOTE	the quotation mark, whose "octal" representation is 42 (7-8 in punched cards)
ALL literal	one or more instances of the literal, which must be a one character non-numeric or figurative constant (other than ALL literal), in which case ALL is redundant but serves for readability.

The plural forms of these figurative constants are acceptable to the compiler but are equivalent in effect. A figurative constant represents as many instances of the associated character as are

required in the context of the statement.

A figurative constant may be used anywhere a literal is called for in a "general format" except that whenever the literal is restricted to being numeric, the only figurative constant permitted is ZERO.

1.11 Structure of a Program

Every COBOL source program is divided into four divisions. Each division must be placed in its proper sequence, and each must begin with a division header.

The four divisions, listed in sequence, and their functions are:

IDENTIFICATION DIVISION, which names the program.

ENVIRONMENT DIVISION, which indicates the computer equipment and features to be used in the program.

DATA DIVISION, which defines the names and characteristics of data to be processed.

PROCEDURE DIVISION, which consists of statements that direct the processing of data at execution time.

The following skeletal coding defines program component structure and order.

IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.

[AUTHOR. comment-entry ...]

[INSTALLATION. comment-entry ...]

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

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

[SECURITY. comment-entry ...]

ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.]

[SOURCE-COMPUTER. entry]

[OBJECT-COMPUTER. entry]

[SPECIAL-NAMES. entry]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. entry ...]

[I-O-CONTROL. entry ...]

DATA DIVISION.

[FILE SECTION.

[file description entry

record description entry ...]...]

[WORKING-STORAGE SECTION.

[data item description entry ...]...]

[LINKAGE SECTION.

[data item description entry ...]...]

PROCEDURE DIVISION [USING identifier-1 ...].

[DECLARATIVES.

{section-name SECTION. USE Sentence.

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

END DECLARATIVES.]

{[section-name SECTION.]

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

1.12 Coding Rules

Since Cromemco COBOL is a subset of American National Standards Institute (ANSI) COBOL, programs may be written on standard COBOL coding sheets, and the following rules are applicable.

1. Each line of code should have a six-digit sequence number in columns 1-6. These numbers should be in ascending order. Blanks are permitted in columns 1-6.
2. Reserved words for division, section, and paragraph headers must begin in Area A (columns 8-11). Procedure-names must also appear in Area A (at the point where they are defined). Level numbers may appear in Area A. Level numbers 01, 77 and level indicator "FD" must begin in Area A.
3. All other program elements should be confined to columns 12-72, governed by the other rules of statement punctuation.
4. Columns 73-80 are ignored by the compiler. These columns may be used to contain program identification.
5. Explanatory comments may be inserted on any line within a source program by placing an asterisk in column 7 of the line. The line will be produced on the source listing but serves no other purpose. If a slash (/) appears in column 7, the associated line is treated as a comment line and will be printed at the top of a new page when the compiler lists the program.
6. Any program element may be "continued" on the following line of a source program. The rules for continuation of a non-numeric ("quoted") literal are explained in Section 1.9. Any other word or literal or other program element is continued by placing a hyphen in the column 7 position of the continuation line. The effect is concatenation of successive word parts, exclusive of all trailing spaces of the last predecessor word and all leading spaces of the first successor word on the continuation line. On a continuation line, Area A must be blank.

1.13 Qualification of Names

When a data-name, condition-name or paragraph name is not unique, procedural reference thereto may be accomplished uniquely by use of qualifier names. For example, if there were two or more items named YEAR, the qualified reference

YEAR OF HIRE-DATE

might differentiate between year fields in HIRE-DATE and TERMINATION-DATE.

Qualifiers are preceded by the word OF or IN; successive data-name or condition-name qualifiers must designate lesser-level-numbered groups that contain all preceding names in the composite reference, i.e., HIRE-DATE must be a group item (or file-name) containing an item called YEAR. Paragraph-names may be qualified by a section-name.

The maximum number of qualifiers is one for a paragraph-name, five for a data-name or condition-name. File-names and mnemonic-names must be unique.

A qualified name may only be written in the Procedure Division. A reference to a multiply-defined paragraph-name need not be qualified when referred to from within the same section.

1.14 COPY Statement

The statement COPY text-name incorporates into a source program a body of standard COBOL code maintained in a "COPY Library" as a distinctly named (text-name) entity. A COPY statement must be terminated by a period. A COPY statement may appear anywhere except within the copied entity itself.

The effect of copying is to augment the source stream processed by the compiler by insertion of the copied entity in place of the COPY statement, and then to resume processing of the primary source of input at the end of the copied entity.

After the text-name operand of a COPY statement, the remainder of the source card must be blank (through column 72).

CHAPTER 2

Identification and Environment Divisions

2.1 Identification Division

Every COBOL program begins with the header: IDENTIFICATION DIVISION. This division is divided into paragraphs having preassigned names:

PROGRAM-ID.	program-name.
AUTHOR.	comments.
INSTALLATION.	comments.
DATE-WRITTEN.	comments.
DATE-COMPILED.	comments.
SECURITY.	comments.

Only the PROGRAM-ID paragraph is required, and it must be the first paragraph. Program-name is any alphanumeric string of characters, the first of which must be alphabetic. Only the first 6 characters of program-name are retained by the compiler. The program-name identifies the object program and is contained in headings on compilation listings.

The contents of any other paragraphs are of no consequence, serving only as documentary remarks.

2.2 Environment Division

The Environment Division specifies a standard method of expressing those aspects of a COBOL program that are dependent upon physical characteristics of a specific computer. It is required in every program.

The general format of the Environment Division is:

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

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

OBJECT-COMPUTER. Computer-name
[MEMORY SIZE integer WORDS | CHARACTERS | MODULES]
[PROGRAM COLLATING SEQUENCE IS ASCII].

SPECIAL-NAMES. [PRINTER IS mnemonic-name] ASCII IS (STANDARD-1)
(NATIVE)
[CURRENCY SIGN IS literal]
[DECIMAL-POINT IS COMMA].

INPUT-OUTPUT SECTION.

FILE-CONTROL. {file-control-entry}...

I-O-CONTROL.

[SAME AREA FOR file-name-2...]...

2.2.1 CONFIGURATION SECTION

The CONFIGURATION SECTION, which has three possible paragraphs, is optional. The three paragraphs are SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES. The contents of the first two paragraphs are treated as commentary, except for the clause WITH DEBUGGING MODE, if present (see Section 4.23). The third paragraph, SPECIAL-NAMES, relates implementor names to user-defined names and changes default editing characters. The PRINTER IS phrase allows definition of a name to be used in the DISPLAY statement with UPON.

In case the currency symbol is not supposed to be the Dollar Sign, the user may specify a single

character non-numeric literal in the CURRENCY SIGN clause. However, the designated character may not be a quote mark, nor any of the characters defined for Picture representations, nor digits (0-9).

The "European" convention of separating integer and fraction positions of numbers with the comma character is specified by employment of the clause DECIMAL-POINT IS COMMA.

Note that the reserved word IS is required in entries for currency sign definition and decimal-point convention specification.

The entry ASCII IS NATIVE/STANDARD-1 specifies that data representation adheres to the American Standard Code for Information Interchange. However, this convention is assumed even if the ASCII-entry is not specifically present. In this compiler, NATIVE and STANDARD-1 are identical, and refer to the character set representation specified in Appendix IV.

2.2.2 INPUT-OUTPUT SECTION

The second section of the Environment Division is mandatory unless the program has no data files; it begins with the header:

INPUT-OUTPUT SECTION.

This section has two paragraphs: FILE-CONTROL and I-O-CONTROL. In this section, the programmer defines the file assignment parameters, including specification of buffering.

2.2.2.1 FILE-CONTROL ENTRY (SELECT ENTRY)

For each file having records described in the Data Division's File Section, a Sentence-Entry (beginning with the reserved word SELECT) is required in the FILE-CONTROL paragraph. The format of a Select Sentence-Entry for a sequential file is:

```
SELECT file-name ASSIGN TO DISK | PRINTER  
  
[RESERVE integer AREAS | AREA]  
  
[FILE STATUS IS data-name-1]  
  
[ACCESS MODE IS SEQUENTIAL] [ORGANIZATION IS SEQUENTIAL].
```

All phrases after "SELECT filename" can be in any order. Both the ACCESS and ORGANIZATION clauses are optional for sequential input-output processing. For Indexed or Relative files, alternate formats are available for this section, and are explained in the chapters on Indexed and Relative files.

If the RESERVE clause is not present, the compiler assigns buffer areas. An integer number of buffers specified by the Reserve clause may be from 1 to 7, but any number over 2 is treated as 2.

In the FILE STATUS entry, data-name-1 must refer to a two-character Working-Storage or Linkage item of category alphanumeric into which the run-time data management facility places status information after an I-O statement. The left-hand character of data-name-1 assumes the values:

- '0' for successful completion
- '1' for End-of-File condition
- '2' for Invalid Key (only
for Indexed and Relative files)
- '3' for a non-recoverable (I-0) error
- '9' for implementor-related errors
(see User's Guide)

The right-hand character of data-name-1 is set to '0' if no further status information exists for the previous I-O operation. The following combinations of values are possible:

<u>File</u> <u>Status</u> <u>Left</u>	<u>File</u> <u>Status</u> <u>Right</u>	<u>Meaning</u>
'0'	'0'	O.K.
'1'	'0'	EOF
'3'	'0'	Permanent error
'3'	'4'	Disk space full

For values of status-right when status-left has a value of '2', see the chapters on Indexed or Relative files.

2.2.2.2 I-O-CONTROL PARAGRAPH

The SAME AREA specification is optional. It permits the programmer to enumerate files that are open only at mutually exclusive times, in order that they may share the same I-O buffer areas and conserve the utilization of memory space.

The format of the SAME AREA entry (which designates files that all share a common I-O area) is:

SAME AREA FOR file-name-2 file-name-3...

Files named in a given SAME AREA clause need not all have the same organization or access. However, no file may be listed in more than one SAME AREA clause.

CHAPTER 3

Data Division

The Data Division, which is one of the required divisions in a program, is subdivided into three sections: File Section, Working-Storage Section and Linkage Section. Each is discussed in Sections 3.13-3.15, but first, aspects of data specification that apply in all sections will be described.

3.1 Data Items

Several types of data items can be described in COBOL programs. These data items are described in the following paragraphs.

3.1.1 Group Items

A group item is defined as one having further subdivisions, so that it contains one or more elementary items. In addition, a group item may contain other groups. An item is a group item if, and only if, its level number is less than the level number of the immediately succeeding item. If an item is not a group item, then it is an elementary item. The maximum size of a group item is 4095 characters.

3.1.2 Elementary Items

An elementary item is a data item containing no subordinate items.

Alphanumeric Item: An alphanumeric item consists of any combination of characters, making a "character string" data field. If the associated picture contains "editing" characters, it is an alphanumeric edited item.

Report (Edited) Item: A report item is an edited "numeric" item containing only digits and/or special editing characters. It must not exceed 30 characters in length. A report item can be used only as a receiving field for numeric data. It is

designed to receive a numeric item but cannot be used as a numeric item itself.

3.1.3 Numeric Items

Numeric items are elementary items intended to contain numeric data only.

External Decimal Item: An external data item is an item in which one computer character (byte) is employed to represent one digit. A maximum number of 18 digits is permitted; the exact number of digit positions is defined by writing a specific number of 9-characters in the PICTURE description. For example, PICTURE 999 defines a 3-digit item. That is, the maximum decimal value of the item is nine hundred ninety-nine.

If the PICTURE begins with the letter S, then the item also has the capability of containing an "operational sign." An operational sign does not occupy a separate character (byte), unless the "SEPARATE" form of SIGN clause is included in the item's description. Regardless of the form of representation of an operational sign, its purpose is to provide a sign that functions in the normal algebraic manner.

The USAGE of an external decimal item is DISPLAY (see USAGE clause, Section 3.4).

Internal Decimal Item: An internal decimal item is stored in packed decimal format. It is attained by inclusion of the COMPUTATIONAL-3 USAGE clause.

A packed decimal item defined by n 9's in its PICTURE occupies 1/2 of (n + 2) bytes in memory. All bytes except the rightmost contain a pair of digits, and each digit is represented by the binary equivalent of a valid digit value from 0 to 9. The item's low order digit and the operational sign are found in the rightmost byte of a packed item. For this reason, the compiler considers a packed item to have an arithmetic sign, even if the original PICTURE lacked an S-character.

Binary Item: A binary item uses the base 2 system to represent an integer in the range -32768 to 32767. It occupies one 16 bit word composed of two

eight bit bytes. The leftmost bit of the reserved area is the operational sign. A binary item is specified by USAGE IS COMPUTATIONAL.

Index Data-Item: An index-data item has no PICTURE; USAGE IS INDEX. (Refer to Chapter 6, Table Handling by the Indexing Method.")

3.2 DATA DESCRIPTION ENTRY

A Data Description entry specifies the characteristics of each field (item) in a data record. Each item must be described in a separate entry in the same order in which the items appear in the record. Each Data Description entry consists of a level number, a data-name, and a series of independent clauses followed by a period.

The general format of a Data Description entry is:

level-number { data-name }
 { FILLER } (REDEFINES-clause) (JUSTIFIED-clause)
(PICTURE-clause) (USAGE-clause) (SYNCHRONIZED-clause)
(OCCURS-clause) (BLANK-clause) (VALUE-clause) (SIGN-clause).

When this format is applied to specific items of data, it is limited by the nature of the data being described. The format allowed for the description of each data type appears below. Clauses that are not shown in a format are specifically forbidden in that format. Clauses that are mandatory in the description of certain data items are shown without parentheses. The clauses may appear in any order except that a REDEFINES-clause, if used, should come first.

Group Item Format

level-number { data-name }
 { FILLER } (REDEFINES-clause) (USAGE-clause)
 (OCCURS-clause) (SIGN-clause).

Example:

```
Ø1  GROUP-NAME.  
Ø2  FIELD-B PICTURE X.  
Ø2  FIELD-C PICTURE X.
```

NOTE

The USAGE clause may be written at a group level to avoid repetitious writing of it at the subordinate element level.

3.3 FORMATS FOR ELEMENTARY ITEMS

ALPHANUMERIC ITEM (also called a character-string item)

level-number { data-name }
 { FILLER } (REDEFINES-clause) (OCCURS-clause)
PICTURE IS an-form (USAGE IS DISPLAY) (JUSTIFIED-clause)
(VALUE IS non-numeric-literal) (SYNCHRONIZED-clause).

Examples:

```
Ø2  MISC-1 PIC X(53).  
Ø2  MISC-2 PICTURE BXXXBXXB.
```

REPORT ITEM (also called a numeric-edited item)

level-number { data-name }
 { FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS report-form (BLANK WHEN ZERO) (USAGE IS DISPLAY)
(VALUE IS non-numeric literal) (SYNCHRONIZED-clause).

Example:

Ø2 XTOTAL PICTURE \$999,999.99-.

DECIMAL ITEM

level-number { data-name }
 { FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS numeric-form (SIGN-clause)
(USAGE-clause) (VALUE IS numeric-literal) (SYNCHRONIZED-clause).

Examples:

Ø2 HOURS-WORKED PICTURE 99V9, USAGE IS DISPLAY.
Ø2 HOURS-SCHEDULED PIC S99V9, SIGN IS TRAILING.

11 TAX-RATE PIC S99V999 VALUE 1.375, COMPUTATIONAL-3.

BINARY ITEM

level-number { data-name }
 { FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS numeric-form

USAGE IS COMPUTATIONAL|COMP|INDEX

(VALUE IS numeric-literal) (SYNCHRONIZED-clause).

NOTE

A PICTURE or VALUE must not be given for an INDEX Data Item.

Examples:

Ø2 SUBSCRIPT COMP, VALUE ZERO.
Ø2 YEAR-TO-DATE COMPUTATIONAL.

3.4 USAGE CLAUSE

The USAGE clause specifies the form in which numeric data is represented.

The USAGE clause may be written at any level. If USAGE is not specified, the item is assumed to be in "DISPLAY" mode. The general format of the USAGE clause is:

USAGE IS { COMPUTATIONAL
 INDEX
 DISPLAY
 COMPUTATIONAL-3 }

INDEX is explained in Chapter 6, Table Handling. COMPUTATIONAL, which may be abbreviated COMP, usage defines an integer binary field. COMPUTATIONAL-3, which may be abbreviated COMP-3, defines a packed (internal decimal) field.

If a USAGE clause is given at a group level, it applies to each elementary item in the group. The USAGE clause for an elementary item must not contradict the USAGE clause of a group to which the item belongs.

3.5 PICTURE CLAUSE

The PICTURE clause specifies a detailed description of an elementary level data item and may include specification of special report editing. The reserved word PICTURE may be abbreviated PIC.

The general format of the PICTURE clause is:

PICTURE IS { an-form
 numeric-form }
 report-form }

There are three possible types of pictures: An-form, Numeric-form and Report-form.

An-Form Option: This option applies to alphanumeric (character string) items. The PICTURE of an alphanumeric item is a combination of data description characters X, A or 9 and, optionally, editing characters B, Ø and /. An X indicates that the character position may contain any character from the computer's ASCII character set. A Picture that contains at least one of the combinations:

- (a) A and 9, or
- (b) X and 9, or
- (c) X and A

in any order is considered as if every 9, A or X character were X. The characters B, Ø and / may be used to insert blanks or zeros or slashes in the item. This is then called an alphanumeric-edited item.

If the string has only A's and B's, it is considered alphabetic; if it has only 9's, it is numeric (see below).

Numeric-Form Option: The PICTURE of a numeric item may contain a valid combination of the following characters:

- 9 The character 9 indicates that the actual or conceptual digit position contains a numeric character. The maximum number of 9's in a PICTURE is 18.

- V The optional character V indicates the position of an assumed decimal point. Since a numeric item cannot contain an actual decimal point, an assumed decimal point is used to provide the compiler with information concerning the scaling alignment of items involved in computations. Storage is never reserved for the character V. Only one V is permitted in any single PICTURE, and is redundant if it is the rightmost character.
- S The optional character S indicates that the item has an operational sign. It must be the first character of the PICTURE. See also, SIGN clause, Section 3.12.
- P The character P indicates an assumed decimal scaling position. It is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character P is not counted in the size of the data item; that is, memory is not reserved for these positions. However, scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or in items that appear as operands in arithmetic statements. The scaling position character P may appear only to the left or right of the other characters in the string as a continuous string of P's within a PICTURE description. The sign character S and the assumed decimal point V are the only characters which may appear to the left of a leftmost string of P's. Since the scaling position character P implies an assumed decimal point (to the left of the P's if the P's are leftmost PICTURE characters and to the right of the P's if the 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.

Report-Form Option: This option describes a data item suitable as an "edited" receiving field for

presentation of a numeric value. The editing characters that may be combined to describe a report item are as follows:

9 V . Z CR DB , \$ + * B O -P /

The characters 9, P and V have the same meaning as for a numeric item. The meanings of the other allowable editing characters are described as follows:

. The decimal point character specifies that an actual decimal point is to be inserted in the indicated position and the source item is to be aligned accordingly. Numeric character positions to the right of an actual decimal point in a PICTURE must consist of characters of one type. The decimal point character must not be the last character in the PICTURE character string. Picture character 'P' may not be used if '.' is used.

Z * The characters Z and * are called replacement characters. Each one represents a digit position. During execution, leading zeros to be placed in positions defined by Z or * are suppressed, becoming blank or *. Zero suppression terminates upon encountering the decimal point (. or V) or a non-zero digit. All digit positions to be modified must be the same (either Z or *), and contiguous starting from the left. Z or * may appear to the right of an actual decimal point only if all digit positions are the same.

CR DB CR and DB are called credit and debit symbols and may appear only at the right end of a PICTURE. These symbols occupy two character positions and indicate that the specified symbol is to appear in the indicated positions if the value of a source item is negative. If the value is positive or zero, spaces will appear instead. CR and DB and + and - are mutually exclusive.

The comma specifies insertion of a comma between digits. Each insertion character

is counted in the size of the data item, but does not represent a digit position. The comma may also appear in conjunction with a floating string, as described below. It must not be the last character in the PICTURE character string.

A floating string is defined as a leading, continuous series of one of either \$ or + or -, or a string composed of one such character interrupted by one or more insertion commas and/or decimal points. For example:

```

$$,$$$,$$$
++++
--,---,--
+(8).++
$$,$$$.$$

```

A floating string containing N + 1 occurrences of \$ or + or - defines N digit positions. When moving a numeric value into a report item, the appropriate character floats from left to right, so that the developed report item has exactly one actual \$ or + or - immediately to the left of the most significant nonzero digit, in one of the positions indicated by \$ or + or - in the PICTURE. Blanks are placed in all character positions to the left of the single developed \$ or + or -. If the most significant digit appears in a position to the right of positions defined by the floating string, then the developed item contains \$ or + or - in the rightmost position of the floating string, and non-significant zeros may follow. The presence of an actual or implied decimal point in a floating string is treated as if all digit positions to the right of the point were indicated by the PICTURE character 9. In the following examples, b represents a blank in the developed items.

PICTURE	Numeric Value	Developed Item
\$\$\$999	14	bb\$014
--,---,999	-456	bbbbbb-456
\$\$\$\$\$\$	14	bbb\$14

A floating string need not constitute the entire PICTURE of a report item, as shown in the preceding

examples. Restrictions on characters that may follow a floating string are given later in the description.

When a comma appears to the right of a floating string, the string character floats through the comma in order to be as close to the leading digit as possible.

- + The character + or - may appear in a PICTURE either singly or in a floating string. As a fixed sign control character, the + or - must appear as the last symbol in the PICTURE. The plus sign indicates that the sign of the item is indicated by either a plus or minus placed in the character position, depending on the algebraic sign of the numeric value placed in the report field. The minus sign indicates that blank or minus is placed in the character position, depending on whether the algebraic sign of the numeric value placed in the report field is positive or negative, respectively.
- B Each appearance of B in a Picture represents a blank in the final edited value.
- / Each slash in a Picture represents a slash in the final edited value.
- Ø Each appearance of Ø in a Picture represents a position in the final edited value where the digit zero will appear.

Other rules for a report (edited) item PICTURE are:

1. The appearance of one type of floating string precludes any other floating string.
2. There must be at least one digit position character.
3. The appearance of a floating sign string or fixed plus or minus insertion character precludes the appearance of any other of the sign control insertion character, namely, +, -, CR, DB.

4. The characters to the right of a decimal point up to the end of a PICTURE, excluding the fixed insertion characters +, -, CR, DB (if present), are subject to the following restrictions:
 - a. Only one type of digit position character may appear. That is, Z * 9 and floating-string digit position characters \$ + - are all 6, mutually exclusive.
 - b. If one of the numeric character positions to the right of a decimal point is represented by + or - or \$ or Z, then all the numeric character positions in the PICTURE must be represented by the same character.
5. The PICTURE character 9 can never appear to the left of a floating string, or replacement character.

Additional notes on the PICTURE Clause:

1. A PICTURE clause must only be used at the elementary level.
2. An integer enclosed in parentheses and following X 9 \$ Z P * B - or + indicates the number of consecutive occurrences of the PICTURE character.
3. Characters V and P are not counted in the space allocation of a data item. CR and DB occupy two character positions.
4. A maximum of 30 character positions is allowed in a PICTURE character string. For example, PICTURE X(89) consists of five PICTURE characters.
5. A PICTURE must consist of at least one of the characters A Z * X 9 or at least two consecutive appearances of the + or - or \$ characters.
6. The characters '.' S V CR and DB can appear only once in a PICTURE.
7. When DECIMAL-POINT IS COMMA is specified, the explanations for period and comma are understood to apply to comma and period, respectively.

The following examples illustrate the use of PICTURE to edit data. In each example, a movement of data is implied, as indicated by the column headings. (Data value shows contents in storage; scale factor of this source data area is given by the Picture.)

Source Area		Receiving Area	
PICTURE	Data Value	PICTURE	Edited Data
9(5)	12345	\$\$\$, \$\$9.99	\$12,345.00
9(5)	00123	\$\$\$, \$\$9.99	\$123.00
9(5)	00000	\$\$\$, \$\$9.99	\$0.00
9(4)V9	12345	\$\$\$, \$\$9.99	\$1,234.50
V9(5)	12345	\$\$\$, \$\$9.99	\$0.12
S9(5)	00123	-----99	123.00
S9(5)	-00001	-----99	-1.00
S9(5)	00123	++++++99	+123.00
S9(5)	00001	-----99	1.00
9(5)	00123	++++++99	+123.00
9(5)	00123	-----99	123.00
S9(5)	12345	*****99CR	**12345.00
S999V99	02345	ZZZVZZ	2345
S999V99	00004	ZZZVZZ	04

3.6 VALUE CLAUSE

The VALUE clause specifies the initial value of working-storage items. The format of this clause is:

VALUE IS literal

The VALUE clause must not be written in a Data Description entry that also has an OCCURS or REDEFINES clause, or in an entry that is subordinate to an entry containing an OCCURS or REDEFINES clause. Furthermore, it cannot be used in the File or Linkage Sections, except in level 88 condition descriptions.

The size of a literal given in a VALUE clause must be less than or equal to the size of the item as given in the PICTURE clause. The positioning of the literal within a data area is the same as would result from specifying a MOVE of the literal to the

data area, except that editing characters in the PICTURE have no effect on the initialization, nor do BLANK WHEN ZERO or JUSTIFIED clauses. The type of literal written in a VALUE clause depends on the type of data item, as specified in the data item formats earlier in this text. For edited items, values must be specified as non-numeric literals, and must be presented in edited form. A figurative constant may be given as the literal.

When an initial value is not specified, no assumption should be made regarding the initial contents of an item in Working-Storage.

The VALUE clause may be specified at the group level, in the form of a correctly sized non-numeric literal, or a figurative constant. In these cases the VALUE clause cannot be stated at the subordinate levels with the group. However, the value clause should not be written for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED and USAGE (other than USAGE IS DISPLAY). (A form used in level 88 items is explained in Section 3.16)

3.7 REDEFINES CLAUSE

The REDEFINES clause specifies that the same area is to contain different data items, or provides an alternative grouping or description of the same data. The format of the REDEFINES clause is:

REDEFINES data-name-2

When written, the REDEFINES clause should be the first clause following the data-name that defines the entry. The data description entry for data-name-2 should not contain a REDEFINES clause, nor an OCCURS clause.

When an area is redefined, all descriptions of the area remain in effect. Thus, if B and C are two separate items that share the same storage area due to Redefinition, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same physical area would receive Y according to the description of C.

For purposes of discussion of Redefinition, data-name-1 is termed the subject, and data-name-2 is called the object. The levels of the subject and object are denoted by s and t, respectively. The following rules must be obeyed in order to establish a proper redefinition.

1. s must equal t, but must not equal 88.
2. The object must be contained in the same record (Ø1 group level item), unless s=t=Ø1.
3. Prior to definition of the subject and subsequent to definition of the object there can be no level numbers that are numerically less than s.

The length of data-name-1, multiplied by the number of occurrences of data-name-1, may not exceed the length of data-name-2, unless the level of data-name-1 is Ø1 (permitted only outside the File Section). Entries giving the new description must not contain any value clauses, except in level 88. In the File Section, multiple level Ø1 entries subordinate to any given FD represent implicit redefinitions of the same area.

3.8 OCCURS CLAUSE

The OCCURS clause is used in defining related sets of repeated data, such as tables, lists and arrays. It specifies the number of times that a data item with the same format is repeated. Data Description clauses associated with an item whose description includes an OCCURS clause apply to each repetition of the item being described. When the OCCURS clause is used, the data name that is the defining name of the entry must be subscripted or indexed whenever it appears in the Procedure Division. If this data-name is the name of a group item, then all data-names belonging to the group must be subscripted or indexed whenever they are used.

The OCCURS clause must not be used in any Data Description entry having a level number Ø1 or 77. The OCCURS clause has the following format:

OCCURS integer TIMES [INDEXED BY index-name...]

Subscripting: Subscripting provides the facility for referring to data items in a table or list that have not been assigned individual data-names. Subscripting is determined by the appearance of an OCCURS clause in a data description. If an item has an OCCURS clause or belongs to a group having an OCCURS clause, it must be subscripted or indexed whenever it is used. See the chapter on Table Handling for explanations on Indexing and Index Usage. (Exception: the table-name in a SEARCH statement must be referenced without subscripts.)

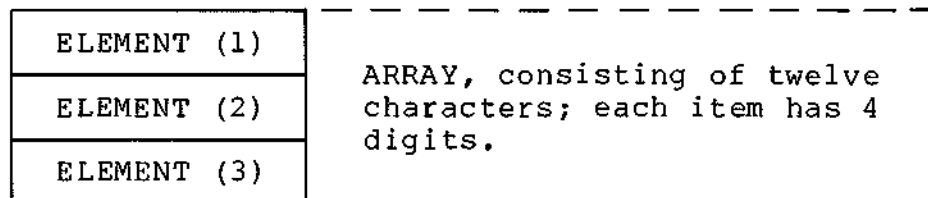
A subscript is a positive nonzero integer whose value determines an element to which a reference is being made within a table or list. The subscript may be represented either by a literal or a data-name that has an integer value. Whether the subscript is represented by a literal or a data-name, the subscript is enclosed in parentheses and appears after the terminal space of the name of the element. A subscript must be a decimal or binary item. (The latter is strongly recommended, for the sake of efficiency.)

At most, three OCCURS clauses may govern any data item. Consequently, one, two or three subscripts may be required. When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the data organization. Multiple subscripts are separated by commas, viz. ITEM (I, J).

Example:

```
Ø1 ARRAY.  
Ø3 ELEMENT, OCCURS 3, PICTURE 9(4).
```

The above example would be allocated storage as shown below.



A data-name may not be subscripted if it is being used for:

1. a subscript
2. the defining name of a data description entry
3. data-name-2 in a REDEFINES clause
4. a qualifier

3.9 SYNCHRONIZED CLAUSE

The SYNCHRONIZED clause was designed in order to allocate space for data in an efficient manner, with respect to the computer central "memory". However, in this compiler, the SYNCHRONIZED specification is treated as commentary only.

The format of this clause is:

SYNC | SYNCHRONIZED [LEFT | RIGHT]

3.10 BLANK WHEN ZERO CLAUSE

The BLANK WHEN ZERO clause specifies that a report (edited) field is to contain nothing except blanks if the numeric value moved to it has a value of zero. When this clause is used with a numeric picture, the field is considered a report field.

3.11 JUSTIFIED CLAUSE

The JUSTIFIED RIGHT clause is only applicable to unedited alphanumeric (character string) items. It signifies that values are stored in a right-to-left fashion, resulting in space fill on the left when a short field is moved to a longer Justified field, or in truncation on the left when a long field is moved to a shorter JUSTIFIED field. The JUSTIFIED clause is effective only when the associated field is employed as the "receiving" field in a MOVE statement.

The word JUST is a permissible abbreviation of JUSTIFIED.

3.12 SIGN CLAUSE

For an external decimal item, there are four possible manners of representing an operational sign; the choice is controlled by inclusion of a particular form of the SIGN clause, whose general form is:

[SIGN IS] TRAILING | LEADING [SEPARATE CHARACTER]

The following chart summarizes the effect of four possible forms of this clause.

SIGN Clause	Sign Representation
TRAILING	Embedded in rightmost byte
LEADING	Embedded in leftmost byte
TRAILING SEPARATE	Stored in separate rightmost byte
LEADING SEPARATE	Stored in separate leftmost byte

When the above forms are written, the PICTURE must begin with S. If no S appears, the item is not signed (and is capable of storing only absolute values), and the SIGN clause is prohibited. When S appears at the front of a PICTURE but no SIGN clause is included in an item's description, the "default" case SIGN IS TRAILING is assumed.

The SIGN clause may be written at a group level; in this case the clause specifies the sign's format on any signed subordinate external decimal item. The SEPARATE CHARACTER phrase increases the size of the data item by 1 character. The entries to which the SIGN clause apply must be implicitly or explicitly described as USAGE IS DISPLAY.

(Note: When the CODE-SET clause is specified for a file, all signed numeric data for that file must be described with the SIGN IS SEPARATE clause.)

3.13 FILE SECTION, FD ENTRIES (SEQUENTIAL I-O ONLY)

In the FILE SECTION of the Data Division, an FD entry (file definition) must appear for every Selected file. This entry precedes the descriptions of the file's record structure(s).

The general format of an FD entry is:

FD file name LABEL-clause [VALUE-OF-clause]
[DATA-RECORD(S)-clause] [BLOCK-clause] [RECORD-clause]
[CODE-SET-clause].

After "FD filename," the order of the clauses is immaterial.

3.13.1 LABEL CLAUSE

The format of this required FD-entry clause is:

LABEL RECORD | RECORDS IS | ARE OMITTED | STANDARD

The OMITTED option specifies that no labels exist for the file; this must be specified for files assigned to PRINTER.

The STANDARD option specifies that labels exist for the file and that the labels conform to system specifications; this must be specified for files assigned to DISK.

3.13.2 VALUE OF CLAUSE

The VALUE OF clause appears in any FD entry for a DISK-assigned file, and contains a file ID expressed as a COBOL-type "quoted" literal. The general form is:

VALUE OF FILE-ID IS "literal"

Example:

VALUE OF FILE-ID "A:MASTER.ASM"

A reminder: if a file is ASSIGNED to PRINTER, it is unlabeled and the VALUE clause must not be included

in the associated FD. If a file is ASSIGNED to DISK, it is necessary to include both LABEL RECORDS STANDARD and VALUE clauses in the associated FD.

3.13.3 DATA RECORD(S) CLAUSE

The optional DATA RECORDS clause identifies the records in the file by name. This clause is documentary only, in this and all COBOL systems. Its general format is:

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

The presence of more than one data-name indicates that the file contains more than one type of data record. That is, two or more record descriptions may apply to the same storage area. The order in which the data-names are listed is not significant.

data-name-1, data-name-2, etc., are the names of data records, and each must be preceded in its record description entry by the level number 01, in the appropriate file declaration (FD) in the File Section.

3.13.4 BLOCK CLAUSE

The BLOCK CONTAINS clause is used to specify characteristics of physical records in relation to the concept of logical records. The general format is:

BLOCK CONTAINS integer-2 { CHARACTERS }
 { RECORDS }

Files assigned to PRINTER must not have a BLOCK clause in the associated FD entry. Furthermore, the BLOCK clause has no effect on disk files in this COBOL system, but it is examined for correct syntax. It is normally applicable to tape files, which are not supported by this COBOL.

When used, the size is usually stated in RECORDS, except when the records are variable in size or

exceed the size of a physical block; in these cases the size should be expressed in CHARACTERS. If multiple record sizes exist, and if blocking is specified, then the physical block will contain multiple logical records, each of which is terminated by a carriage-return line-feed.

When the BLOCK CONTAINS clause is omitted, it is assumed that records are not blocked. When neither the CHARACTERS nor the RECORDS option is specified, the CHARACTERS option is assumed. When the RECORDS option is used, the compiler assumes that the block size provides for integer-2 records of maximum size and then provides additional space for any required control characters.

3.13.5 RECORD CLAUSE

Since the size of each data record is defined fully by the set of data description entries constituting the record (level 01) declaration, this clause is always optional and documentary. The format of this clause is:

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

Integer-2 should be the size of the biggest record in the file declaration. If the records are variable in size, Integer-1 must be specified and equal the size of the smallest record. The sizes are given as character positions required to store the logical records.

3.13.6 CODE-SET CLAUSE

The format of this clause is:

CODE-SET IS ASCII

The CODE-SET clause, which should be specified only for non-mass-storage files, serves only the purposes of documentation in this compiler, reflecting the fact that both internal and external data are represented in ASCII code. However, any signed numeric data description entries in the file's record should include the SIGN IS SEPARATE clause and all data in the file should have DISPLAY USAGE.

3.14 WORKING-STORAGE SECTION

The second section of the DATA DIVISION begins with the header WORKING-STORAGE SECTION. This section describes records and other data which are not part of external data files but which are developed and processed internally.

Data description entries in this section may employ level numbers 01-49, as in the File section, as well as 77. Value clauses, prohibited in the File section (except for level 88), are permitted throughout the Working-storage section.

3.15 LINKAGE SECTION

The third section of the Data Division is defined by the header LINKAGE SECTION. In this section, the user describes data by name and attribute, but storage space is not allocated. Instead, these "dummy" descriptions are applied (through the mechanism of the USING list on the Procedure Division header) to data whose addresses are passed into a subprogram by a call upon it from a separately compiled program. Consequently, VALUE clauses are prohibited in the Linkage Section, except in level 88 condition-name entries. Refer to Chapter 5, Inter-Program Communication, for further information.

3.16 LEVEL 88 CONDITION-NAMES

The level 88 condition-name entry specifies a value, list of values, or a range of values that an elementary item may assume, in which case the named condition is true, otherwise false. The format of a level 88 item's value clause is

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

A level 88 entry must be preceded either by another level 88 entry (in the case of several consecutive condition-names pertaining to an elementary item) or by an elementary item (which may be FILLER). INDEX data items should not be followed by level 88 items.

Every condition-name pertains to an elementary item in such a way that the condition-name may be qualified by the name of the elementary item and the elementary item's qualifiers. A condition-name is used in the Procedure Division in place of a simple relational condition. A condition-name may pertain to an elementary item (a conditional variable) requiring subscripts. In this case, the condition-name, when written in the Procedure Division, must be subscripted according to the same requirements as the associated elementary item. The type of literal in a condition-name entry must be consistent with the data type of the conditional variable. In the following example, PAYROLL-PERIOD is the conditional variable. The picture associated with it limits the value of the 88 condition-name to one digit.

```
Ø2 PAYROLL-PERIOD PICTURE IS 9.  
   88 WEEKLY VALUE IS 1.  
   88 SEMI-MONTHLY VALUE IS 2.  
   88 MONTHLY VALUE IS 3.
```

Using the above description, the following procedural condition-name test may be written:

```
IF MONTHLY GO TO DO-MONTHLY
```

An equivalent statement is:

```
IF PAYROLL-PERIOD = 3 GO TO DO-MONTHLY.
```

For an edited elementary item, values in a condition-name entry must be expressed in the form of non-numeric literals.

A VALUE clause may not contain both a series of literals and a range of literals.

CHAPTER 4

Procedure Division

In this chapter, the basic concepts of the Procedure Division are explained. Advanced topics (such as Indexing of tables, Indexed file accessing, interprogram communication and Declaratives) are discussed in subsequent chapters.

4.1 STATEMENTS, SENTENCES, PROCEDURE-NAMES

The Procedure portion of a source program specifies those procedures needed to solve a given EDP problem. These steps (computations, logical decisions, etc.) are expressed in statements similar to English, which employ the concept of verbs to denote actions, and statements and sentences to describe procedures. The Procedure portion must begin with the words PROCEDURE DIVISION.

A statement consists of a verb followed by appropriate operands (data-names or literals) and other words that are necessary for the completion of the statement. The two types of statements are imperative and conditional.

Imperative Statements

An imperative statement specifies an unconditional action to be taken by the object program. An imperative statement consists of a verb and its operands, excluding the IF and SEARCH conditional statements and any statement which contains an INVALID KEY, AT END, SIZE ERROR, or OVERFLOW clause.

Conditional Statements

A conditional statement stipulates a condition that is tested to determine whether an alternate path of program flow is to be taken. The IF and SEARCH statements provide this capability. Any I/O statement having an INVALID KEY or AT END clause is also considered to be conditional. When an arithmetic statement possesses a SIZE ERROR suffix, the statement is considered to be conditional rather

than imperative. STRING or UNSTRING statements having an OVERFLOW clause are also conditional.

Sentences

A sentence is a single statement or a series of statements terminated by a period and followed by a space. If desired, a semi-colon or comma may be used between statements in a sentence.

Paragraphs

A paragraph is a logical entity consisting of zero, one or more sentences. Each paragraph must begin with a paragraph-name.

Paragraph-names and section-names are procedure-names. Procedure-names follow the rules for name-formation. In addition, a procedure-name may consist only of digits. An all-digit procedure-name may not consist of more than 18 digits; if it has leading zeros, they are all significant.

Sections

A section is composed of one or more successive paragraphs, and must begin with a section-header. A section header consists of a section-name conforming to the rules for procedure-name formation, followed by the word SECTION and a period. A section header must appear on a line by itself. Each section-name must be unique.

4.2 ORGANIZATION OF THE PROCEDURE DIVISION

The PROCEDURE part of a program may be subdivided in three possible ways:

1. The Procedure Division consists only of paragraphs.
2. The Procedure Division consists of a number of paragraphs followed by a number of sections (each section subdivided into one or more paragraphs).
3. The Procedure Division consists of a DECLARATIVES portion and a series of sections (each section subdivided into one or more

paragraphs).

The DECLARATIVES portion of the Procedure Division is optional; it provides a means of designating a procedure to be invoked in the event of an I/O error. If Declaratives are utilized, only possibility 3 may be used. Refer to Chapter 9 for a complete discussion.

4.3 MOVE STATEMENT

The MOVE statement is used to move data from one area of main storage to another and to perform conversions and/or editing on the data that is moved. The MOVE statement has the following format:

```
MOVE      { data-name-1 } TO data-name-2  [data-name-3...]  
           { literal      }
```

The data represented by data-name-1 or the specified literal is moved to the area designated by data-name-2. Additional receiving fields may be specified (data-name-3 etc.). When a group item is a receiving field, characters are moved without regard to the level structure of the group involved and without editing.

Subscripting or indexing associated with data-name-2 is evaluated immediately before data is moved to the receiving field. The same is true for other receiving fields (data-name-3, etc., if any). But for the source field, subscripting or indexing (associated with data-name-1) is evaluated only once, before any data is moved.

To illustrate, consider the statement

```
MOVE A (B) TO B, C (B),
```

which is equivalent to

```
MOVE A (B) TO temp  
MOVE temp TO B  
MOVE temp TO C (B)
```

where temp is an intermediate result field assigned automatically by the compiler.

The following considerations pertain to moving

items:

1. Numeric (external or internal decimal, binary, numeric literal, or ZERO) or alphanumeric to numeric or report:
 - a. The items are aligned by decimal points, with generation of zeros or truncation on either end, as required. If source is alphanumeric, it is treated as an unsigned integer and should not be longer than 31 characters.
 - b. When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place. Alphanumeric source items are treated as unsigned integers with Usage Display.
 - c. The items may have special editing performed on them with suppression of zeros, insertion of a dollar sign, etc., and decimal point alignment, as specified by the receiving area.
 - d. One should not move an item whose PICTURE declares it to be alphabetic or alphanumeric edited to a numeric or report item, nor is it possible to move a numeric item of any sort to an alphabetic item though numeric integers and numeric report items can be moved to alphanumeric items with or without editing, but operational signs are not moved in this case even if "SIGN IS SEPARATE" has been specified.
2. Non-numeric source and destinations:
 - a. The characters are placed in the receiving area from left to right, unless JUSTIFIED RIGHT applies.
 - b. If the receiving field is not completely filled by the data being moved, the remaining positions are filled with spaces.
 - c. If the source field is longer than the receiving field, the move is terminated as soon as the receiving field is filled.

3. When overlapping fields are involved, results are not predictable.
4. Appendix II shows, in tabular form, all permissible combinations of source and receiving field types.
5. An item having USAGE IS INDEX cannot appear as an operand of a MOVE statement. See SET in Chapter 6, Table Handling.

Examples of Data Movement (b represents blank):

Source Field		Receiving Field		
PICTURE	Value	PICTURE	Value before MOVE	Value after MOVE
99V99	1234	S99V99	9876-	1234+
99V99	1234	99V9	987	123
S9V9	12-	99V999	98765	01200+
XXX	A2B	XXXXX	Y9X8W	A2Bbb
9V99	123	99.99	87.65	01.23

4.4 INSPECT STATEMENT

The INSPECT statement enables the programmer to examine a character-string item. Options permit various combinations of the following actions:

1. counting appearances of a specified character
2. replacing a specified character with another
3. limiting the above actions by requiring the appearance of other specific characters

The format of the INSPECT statement is:

INSPECT data-name-1 [TALLYING-clause] [REPLACING-clause]

where TALLYING-clause has the format

TALLYING data-name-2 FOR { CHARACTERS
ALL | LEADING operand-3 }

[BEFORE | AFTER INITIAL operand-4]

and REPLACING-clause has the format

REPLACING { CHARACTERS
ALL | LEADING | FIRST operand-5 } BY operand-6

[BEFORE | AFTER INITIAL operand-7]

Because data-name-1 is to be treated as a string of characters by INSPECT, it must not be described by USAGE IS INDEX, COMP, or COMP-3. Data-name-2 must be a numeric data item.

In the above formats, operand-n may be a quoted literal of length one, a figurative constant signifying a single character, or a data-name of an item whose length is one.

TALLYING-clause and REPLACING-clause may not both be omitted; if both are present, TALLYING-clause must be first.

TALLYING-clause causes character-by-character comparison, from left to right, of data-name-1, incrementing data-name-2 by one each time a match is found. When an AFTER INITIAL operand-4 subclause is present, the counting process begins only after detection of a character in data-name-1 matching operand-4. If BEFORE INITIAL operand-4 is specified, the counting process terminates upon encountering a character in data-name-1 which matches operand-4. Also going from left to right, REPLACING-clause causes replacement of characters under conditions specified by the REPLACING-clause. If BEFORE INITIAL operand-7 is present, replacement does not continue after detection of a character in data-name-1 matching operand-7. If AFTER INITIAL operand-7 is present, replacement does not commence until detection of a character in data-name-1 matching operand-7.

With bounds on data-name-1 thus determined, TALLYING and REPLACING is done on characters as specified by the following:

1. "CHARACTERS" implies that every character in the bounded data-name-1 is to be TALLYed or REPLACed.
2. "All operand" means that all characters in the bounded data-name-1 which match the "operand" character are to participate in TALLYING/REPLACING.
3. "LEADING operand" specifies that only characters matching "operand" from the leftmost portion of the bounded data-name-1 which are contiguous (such as leading zeros) are to participate in TALLYING or REPLACING.
4. "FIRST operand" specifies that only the first-encountered character matching "operand" is to participate in REPLACING. (This option is unavailable in TALLYING.)

When both TALLYING and REPLACING clauses are present, the two clauses behave as if two INSPECT statements were written, the first containing only a TALLYING-clause and the second containing only a REPLACING-clause.

In developing a TALLYING value, the final result in data-name-2 is equal to the tallied count plus the initial value of data-name-2. In the first example below, the item COUNTX is assumed to have been set to zero initially elsewhere in the program.

```
INSPECT ITEM TALLYING COUNTX FOR ALL "L" REPLACING  
LEADING "A" BY "E" AFTER INITIAL "L"
```

Original (ITEM):	SALAMI	ALABAMA
Result (ITEM):	SALEMI	ALEBAMA
Final (COUNTX):	1	1

```
INSPECT WORK-AREA REPLACING ALL DELIMITER BY TRANSFORMATION
```

Original (WORK-AREA):	NEW YORK N Y	(length 16)
Original (DELIMITER):	(space)	
Original (TRANSFORMATION):	. (period)	
Result (WORK-AREA):	NEW.YORK..N.Y...	

NOTE

If any data-name-1 or operand-n is described as signed numeric, it is treated as if it were unsigned.

4.5 ARITHMETIC STATEMENTS

There are five arithmetic statements: ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE. Any arithmetic statement may be either imperative or conditional. When an arithmetic statement includes an ON SIZE ERROR specification, the entire statement is termed conditional, because the size-error condition is data-dependent.

An example of a conditional arithmetic statement is:

```
ADD 1 TO RECORD-COUNT, ON SIZE ERROR MOVE ZERO TO  
RECORD-COUNT, DISPLAY "LIMIT 99 EXCEEDED".
```

Note that if a size error occurs (in this case, it is apparent that RECORD-COUNT has Picture 99, and cannot hold a value of 100), both the MOVE and DISPLAY statements are executed.

The three statement components that may appear in arithmetic statements (GIVING option, ROUNDED option, and SIZE ERROR option) are discussed in detail later in this section.

Basic Rules for Arithmetic Statements

1. All data-names used in arithmetic statements must be elementary numeric data items that are defined in the Data Division of the program, except that operands of the GIVING option may be report (numeric edited) items. Index-names and index-items are not permissible in these arithmetic statements (see Chapter 6).
2. Decimal point alignment is supplied automatically throughout the computations.
3. Intermediate result fields generated for the evaluation of arithmetic expressions assure the accuracy of the result field, except where high-

order truncation is necessary.

4.5.1 SIZE ERROR OPTION

If, after decimal-point alignment and any low-order rounding, the value of a calculated result exceeds the largest value which the receiving field is capable of holding, a size error condition exists.

The optional SIZE ERROR clause is written immediately after any arithmetic statement, as an extension of the statement. The format of the SIZE ERROR option is:

ON SIZE ERROR imperative statement ...

If the SIZE ERROR option is present, and a size error condition arises, the value of the resultant data-name is unaltered and the series of imperative statements specified for the condition is executed.

If the SIZE ERROR option has not been specified and a size error condition arises, no assumption should be made about the final result.

An arithmetic statement, if written with SIZE ERROR option, is not an imperative statement. Rather, it is a conditional statement and is prohibited in contexts where only imperative statements are allowed.

4.5.2 ROUNDED OPTION

If, after decimal-point alignment, the number of places in the fraction of the result is greater than the number of places in the fractional part of the data item that is to be set equal to the calculated result, truncation occurs unless the ROUNDED option has been specified.

When the ROUNDED option is specified, the least significant digit of the resultant data-name has its value increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

Rounding of a computed negative result is performed by rounding the absolute value of the computed result and then making the final result negative.

The following chart illustrates the relationship between a calculated result and the value stored in an item that is to receive the calculated result, with and without rounding.

Calculated Result	Item to Receive Calculated Result		
	PICTURE	Value After Rounding	Value After Truncating
-12.36	S99V9	-12.4	-12.3
8.432	9V9	8.4	8.4
35.6	99V9	35.6	35.6
65.6	S99V	66	65
.0055	SV999	.006	.005

Illustration of Rounding

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

4.5.3 GIVING OPTION

If the GIVING option is written, the value of the data-name that follows the word GIVING is made equal to the calculated result of the arithmetic operation. The data-name that follows GIVING is not used in the computation and may be a report (numeric edited) item.

4.5.4 ADD STATEMENT

The ADD statement adds two or more numeric values and stores the resulting sum.

The ADD statement general format is:

```

ADD      { numeric-literal }
         { data-name-1 }      ...

         { TO
         { GIVING } } data-name-n [ROUNDED] [SIZE-ERROR-clause]
  
```


When the TO option is used, the values of all the data-names (including data-name-n) and literals in the statements are added, and the resulting sum replaces the value of data-name-n. At least two data-names and/or numeric literals must follow the word ADD when the GIVING option is written.

The following are examples of proper ADD statements:

```
ADD INTEREST, DEPOSIT TO BALANCE ROUNDED  
ADD REGULAR-TIME OVERTIME GIVING GROSS-PAY.
```

The first statement would result in the sum of INTEREST, DEPOSIT, and BALANCE being placed at BALANCE, while the second would result in the sum of REGULAR-TIME and OVERTIME earnings being placed in item GROSS-PAY.

4.5.5 SUBTRACT STATEMENT

The SUBTRACT statement subtracts one or more numeric data items from a specified item and stores the difference.

The SUBTRACT statement general format is:

```
SUBTRACT { data-name-1  
            numeric-literal-1 } ... FROM  
  
            { data-name-m [GIVING data-name-n]  
            numeric literal-m GIVING data-name-n }  
  
            [ROUNDED] [SIZE-ERROR-clause]
```

The effect of the SUBTRACT statement is to sum the values of all the operands that precede FROM and subtract that sum from the value of the item following FROM.

The result (difference) is stored in data-name-n, if there is a GIVING option. Otherwise, the result is stored in data-name-m.

4.5.6 MULTIPLY STATEMENT

The MULTIPLY statement multiplies two numeric data items and stores the product.

The general format of the MULTIPLY statement is:

```
MULTIPLY      { data-name-1  
                { numeric-literal-1 }  
  
BY           { data-name-2 [GIVING data-name-3]  
                { numeric-literal-2 GIVING data-name-3 }  
  
                [ ROUNDED ] [SIZE-ERROR-clause]
```

When the GIVING option is omitted, the second operand must be a data-name; the product replaces the value of data-name-2. For example, a new BALANCE value is computed by the statement MULTIPLY 1.03 BY BALANCE. (Since this order might seem somewhat unnatural, it is recommended that GIVING always be written.)

4.5.7 DIVIDE STATEMENT

The DIVIDE statement divides two numeric values and stores the quotient. The general format of the DIVIDE statement is:

```
DIVIDE      { data-name-1  
                { numeric-literal-1 } } { BY } { data-name-2  
                { numeric-literal-2 } } { INTO }  
  
                [ GIVING data-name-3 ] [ ROUNDED ] [SIZE-ERROR-clause]
```

The BY-form signifies that the first operand (data-name-1 or numeric-literal-1) is the dividend (numerator), and the second operand (data-name-2 or numeric-literal-2) is the divisor (denominator). If GIVING is not written in this case, then the first operand must be a data-name, in which the quotient is stored.

The INTO-form signifies that the first operand is the divisor and the second operand is the dividend. If GIVING is not written in this case, then the second operand must be a data-name, in which the quotient is stored.

Division by zero always causes a size-error

condition.

4.5.8 COMPUTE STATEMENT

The COMPUTE statement evaluates an arithmetic expression and then stores the result in a designated numeric or report (numeric edited) item.

The general format of the COMPUTE statement is:

COMPUTE data-name-1 [ROUNDED]....=

{ data-name-2
numeric-literal
arithmetic-expression } [SIZE-ERROR-clause]

An example of such a statement is:

```
COMPUTE GROSS-PAY ROUNDED = BASE-SALARY *  
(1 + 1.5* (HOURS -40) / 40).
```

An arithmetic expression is a proper combination of numeric literals, data-names, arithmetic operators and parentheses. In general, the data-names in an arithmetic expression must designate numeric data. Consecutive data-names (or literals) must be separated by an arithmetic operator, and there must be one or more blanks on either side of the operator. The operators are:

+ for addition
-for subtraction
* for multiplication
/ for division
** for exponentiation to an integral power.

When more than one operation is to be executed using a given variable or term, the order of precedence is:

1. Unary (involving one variable) plus and minus
2. Exponentiation
3. Multiplication and Division
4. Addition and Subtraction

Parentheses may be used when the normal order of operations is not desired. Expressions within parentheses are evaluated first; parentheses may be nested to any level. Consider the following expression.

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

Evaluation of the above expression is performed in the following ordered sequence:

1. Compute the product D times E, considered as intermediate result R1.
2. Compute intermediate result R2 as the difference C - R1.
3. Divide B by R2, providing intermediate result R3.
4. The final result is computed by addition of A to R3.

Without parentheses, the expression

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

is evaluated as:

```
R1 = B / C
R2 = A + R1
R3 = D * E
final result = R2 - R3
```

When parentheses are employed, the following punctuation rules should be used:

1. A left parenthesis is preceded by one or more spaces.
2. A right parenthesis is followed by one or more spaces.

The expression A - B - C is evaluated as (A - B) - C. Unary operators are permitted, e.g.:

```
COMPUTE A = +C + -4.6
COMPUTE X = -Y
COMPUTE A, B(I) = -C - D(3)
```

4.6 GO TO STATEMENT

The GO TO statement transfers control from one portion of a program to another. It has the following general format:

GO TO procedure-name [...DEPENDING ON data-name]

The simple form GO TO procedure-name changes the path of flow to a designated paragraph or section. If the GO statement is without a procedure-name, then that GO statement must be the only one in a paragraph, and must be altered (see 4.12) prior to its execution.

The more general form designates N procedure-names as a choice of N paths to transfer to, if the value of data-name is 1 to N, respectively. Otherwise, there is no transfer of control and execution proceeds in the normal sequence. Data-name must be a numeric elementary item and have no positions to the right of the decimal point.

If a GO (non-DEPENDING) statement appears in a sequence of imperative statements, it must be the last statement in that sequence.

4.7 STOP STATEMENT

The STOP statement is used to terminate or delay execution of the object program.

The format of this statement is:

STOP { RUN }
 { literal }

STOP RUN terminates execution of a program, returning control to the operating system. If used in a sequence of imperative statements, it must be the last statement in that sequence.

The form STOP literal displays the specified literal on the console and suspends execution.

Execution of the program is resumed only after operator intervention. Presumably, the operator performs a function suggested by the content of the literal, prior to resuming program execution. For

more information, see the COBOL User's Guide.

4.8 ACCEPT STATEMENT

The ACCEPT statement is used to enter data into the computer on a low volume basis, from operator key-in at the computer console. The format of the ACCEPT statement is:

ACCEPT data-name

One line is read, and as many characters as necessary (depending on the size of the named data field) are moved, without change, to the indicated field. If the input is shorter than the receiving field, the extra positions are filled with spaces (blanks).

When input is to be accepted from the console, execution is suspended. After the operator enters a response, the program stores the acquired data in the field designed by data-name, and normal execution proceeds. A form of the ACCEPT statement used to acquire the current date, day or time is explained in Section 4.20.

4.9 DISPLAY STATEMENT

The DISPLAY statement provides a simple means of outputting low-volume data without the complexities of File Definition; the maximum number of characters to be output per line is 132. The format of the DISPLAY statement is:

DISPLAY {data-name} [UPON mnemonic-name]
{literal}...

When the UPON suffix is omitted, it is understood that output is destined to be printed on the console. Use of the suffix UPON mnemonic-name directs that output to the printer. Mnemonic-name must be assigned to PRINTER in the SPECIAL-NAMES paragraph.

Values output are either literals, figurative constants (one character), or data fields. If a data item operand is packed, it is displayed as a series of digits followed by a separate trailing

sign.

4.10 PERFORM STATEMENT

The PERFORM statement permits the execution of a separate body of program steps. Two formats of the PERFORM statement are available:

Option 1

PERFORM range [{integer }
 {data-name} TIMES]

Option 2

PERFORM range VARYING {index-name}
 {data-name} FROM
 amount-1 BY amount-2 UNTIL condition.

(A more extensive version of option 2 is available for varying 2 or 3 items concurrently, as explained in Appendix VI.)

In the above syntactical presentation, the following definitions are assumed:

1. Range is a paragraph-name, a section-name, or the construct procedure-name-1 THRU procedure-name-2. (THROUGH is synonymous with THRU.) If only a paragraph-name is specified, the return is after the paragraph's last statement. If only a section-name is specified, the return is after the last statement of the last paragraph of the section. If a range is specified, control is returned after the appropriate last sentence of a paragraph or section. These return points are valid only when a PERFORM has been executed to set them up; in other cases, control will pass right through.
2. The generic operands amount-1 and amount-2 may be a numeric literal, index-name, or data-name. In practice, these amount specifications are frequently integers, or data-names that contain integers, and the specified data-name is used as a subscript within the range.

In Option 1, the designated range is performed a fixed number of times, as determined by an integer or by the value of an integer data-item. If no "TIMES" phrase is given, the range is performed once. When any PERFORM has finished, execution proceeds to the next statement following the PERFORM.

In Option 2, the range is performed a variable number of times, in a step-wise progression, varying from an initial value of data-name = amount-1, with increments of amount-2, until a specified condition is met, at which time execution proceeds to the next statement after the PERFORM.

The condition in an Option 2 PERFORM is evaluated prior to each attempted execution of the range. Consequently, it is possible to not PERFORM the range, if the condition is met at the outset. Similarly, in Option 1, if data-name ≤ 0 , the range is not performed at all.

At run-time, it is illegal to have concurrently active PERFORM ranges whose terminus points are the same.

4.11 EXIT STATEMENT

The EXIT statement is used where it is necessary to provide an endpoint for a procedure.

The format for the EXIT statement is:

paragraph-name. EXIT.

EXIT must appear in the source program as a one-word paragraph preceded by a paragraph-name. An exit paragraph provides an end-point to which preceding statements may transfer control if it is decided to bypass some part of a section.

4.12 ALTER STATEMENT

The ALTER statement is used to modify a simple GO TO statement elsewhere in the Procedure Division, thus changing the sequence of execution of program statements.

The ALTER statement general format is:

ALTER paragraph TO [PROCEED TO] procedure-name

Paragraph (the first operand) must be a COBOL paragraph that consists of only a simple GO TO statement; the ALTER statement in effect replaces the former operand of that GO TO by procedure-name. Consider the ALTER statement in the context of the following program segment.

```
GATE.      GO TO MF-OPEN.
MF-OPEN.   OPEN INPUT MASTER-FILE.
           ALTER GATE TO PROCEED TO NORMAL.
NORMAL.    READ MASTER-FILE, AT END GO TO
           EOF-MASTER.
```

Examination of the above code reveals the technique of "shutting a gate", providing a one-time initializing program step.

4.13 IF STATEMENT

The IF statement permits the programmer to specify a series of procedural statements to be executed in the event a stated condition is true. Optionally, an alternative series of statements may be specified for execution if the condition is false. The general format of the IF statement is:

IF condition { NEXT SENTENCE } [ELSE statement(s)-2]
 { statement(s)-1 } [ELSE NEXT SENTENCE]

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

Examples of IF statements:

1. IF BALANCE = 0 GO TO NOT-FOUND.
2. IF T LESS THAN 5 NEXT SENTENCE ELSE GO TO T-1-4.

3. IF ACCOUNT-FIELD = SPACES OR NAME = SPACES ADD 1
TO SKIP-COUNT ELSE GO TO BYPASS.

The first series of statements is executed only if the designated condition is true. The second series of statements (ELSE part) is executed only if the designated condition is false. The second series (ELSE part) is terminated by a sentence-ending period unless it is "ELSE NEXT SENTENCE", in which case more statements may be written before the period. If there is no ELSE part to an IF statement, then the first series of statements must be terminated by a sentence-ending period. Refer to Appendix III for discussion of nested IF statements.

Regardless of whether the condition is true or false, the next sentence is executed after execution of the appropriate series of statements, unless a GO TO is contained in the imperatives that are executed, or unless the nominal flow of program steps is superseded because of an active PERFORM statement.

4.13.1 Conditions

A condition is either a simple condition or a compound condition. The four simple conditions are the relational, class, condition-name, and sign condition tests. A simple relational condition has the following structure:

operand-1 relation operand-2

where "operand" is a data-name, literal, or figurative-constant.

A compound condition may be formed by connecting two conditions, of any sort, by the logical operator AND or OR, e.g., A < B OR C = D. Refer to Appendix I for further permissible forms involving parenthesization, NOT, or "abbreviation".

The simplest "simple relations" have three basic forms, expressed by the relational symbols equal to, less than, or greater than (i.e., = or < or >).

Another form of simple relation that may be used involves the reserved word NOT, preceding any of the three relational symbols. In summary, the six

simple relations in conditions are:

<u>Relation</u>	<u>Meaning</u>
=	equal to
<	less than
>	greater than
NOT =	not equal to
NOT <	greater than or equal to
NOT >	less than or equal to

It is worthwhile to briefly discuss how relation conditions can be compounded. The reserved words AND or OR permit the specification of a series of relational tests, as follows:

1. Individual relations connected by AND specify a compound condition that is met (true) only if all the individual relationships are met.
2. Individual relations connected by OR specify a compound condition that is met (true) if any one of the individual relationships is met.

The following is an example of a compound relation condition containing both AND and OR connectors. Refer to Appendix I for formal specification of evaluation rules.

IF X = Y AND FLAG = 'Z' OR SWITCH = 0 GOTO PROCESSING.

In the above example, execution will be as follows, depending on various data values.

Data Value				Does Execution Go to PROCESSING?
X	Y	FLAG	SWITCH	
10	10	'Z'	1	Yes
10	11	'Z'	1	No
10	11	'Z'	0	Yes
10	10	'P'	1	No
6	3	'P'	0	Yes
6	6	'P'	1	No

Usages of reserved word phrasings EQUAL TO, LESS THAN, and GREATER THAN are accepted equivalents of = < > respectively. Any form of the relation may be preceded by the word IS, optionally.

Before discussing class-test, sign-test, and condition-name-test conditions, methods of performing comparisons will be discussed.

Numeric Comparisons: The data operands are compared after alignment of their decimal positions. The results are as defined mathematically, with any negative values being less than zero, which in turn is less than any positive value. An index-name or index item (see Chapter 6) may appear in a comparison. Comparison of any two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses, and regardless of length.

Character Comparisons: Non-equal-length comparisons are permitted, with spaces being assumed to extend the length of the shorter item, if necessary. Relationships are defined in the ASCII code; in particular, the letters A-Z are in an ascending sequence, and digits are less than letters. Group items are treated simply as characters when compared. Refer to Appendix IV for all ASCII character representations. If one operand is numeric and the other is not, it must be an integer and have an implicit or explicit USAGE IS DISPLAY.

Returning to our discussion of simple conditions, there are three additional forms of a simple condition, in addition to the relational form, namely: class test, condition-name test (88), and sign test.

A class test condition has the following syntactical format:

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

This condition specifies an examination of the data item content to determine whether all characters are proper digit representations regardless of any operational sign (when the test is for NUMERIC), or only alphabetic or blank space characters (when the test is for ALPHABETIC). The NUMERIC test is valid

only for a group, decimal, or character item (not having an alphabetic PICTURE). The ALPHABETIC test is valid only for a group or character item (Picture an-form).

A sign test has the following syntactical format:

data-name IS [NOT] NEGATIVE | ZERO | POSITIVE

This test is equivalent to comparing data-name to zero in order to determine the truth of the stated condition.

In a condition-name test, a conditional variable is tested to determine whether its value is equal to one of the values associated with the condition-name. A condition-name test is expressed by the following syntactical format:

condition-name

where condition-name is defined by a level 88 data division entry.

4.14 OPEN STATEMENT (Sequential I-O)

The OPEN statement must be executed prior to commencing file processing. The general format of an OPEN statement is:

$$\left\{ \begin{array}{l} \text{OPEN} \\ \left\{ \begin{array}{l} \text{INPUT} \\ \text{I-O} \\ \text{OUTPUT} \\ \text{EXTEND} \end{array} \right\} \text{file-name...} \end{array} \right\} \dots$$

For a sequential INPUT file, opening initiates reading the file's first records into memory, so that subsequent READ statements may be executed without waiting.

For an OUTPUT file, opening makes available a record area for development of one record, which will be transmitted to the assigned output device upon the execution of a WRITE statement. An existent file which has the same name will be superceded by the file created with OPEN OUTPUT.

An I-O opening is valid only for a DISK file; it permits use of the REWRITE statement to modify

records which have been accessed by a READ statement. The WRITE statement may not be used in I-O mode for files with sequential organization. The file must exist on disk at OPEN time; it cannot be created by OPEN I-O.

When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file will add records to the end of the file. Thus, processing proceeds as though the file had been opened with the OUTPUT phrase and positioned at its end. EXTEND can be used only for sequential files.

Failure to precede (in terms of time sequence) file reading or writing by the execution of an OPEN statement is an execution-time error which will cause abnormal termination of a program run. See User's Guide. Furthermore, a file cannot be opened if it has been CLOSED "WITH LOCK."

Sequential files opened for INPUT or I-O access must have been written in the appropriate format described in the User's Guide for such files.

4.15 READ STATEMENT (Sequential I-O)

The READ statement makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data item, if one was specified. The general format of a READ statement is:

```
READ file-name RECORD [INTO data-name] [AT END  
imperative statement...]
```

Since at some time the end-of-file will be encountered, the user should include the AT END clause. The reserved word END is followed by any number of imperative statements, all of which are executed only if the end-of-file situation arises. The last statement in the AT END series must be followed by a period to indicate the end of the sentence. If end-of-file occurs but there is no AT END clause on the READ statement, an applicable Declarative procedure is performed. If neither AT END nor Declarative exists and no FILE STATUS item

is specified for the file, a run-time I/O error is processed.

When a data record to be read exists, successful execution of the READ statement is immediately followed by execution of the next sentence.

When more than one Ø1-level item is subordinate to a file definition, these records share the same storage area. Therefore, the user must be able to distinguish between the types of records that are possible, in order to determine exactly which type is currently available. This is accomplished with a data comparison, using an IF statement to test a field which has a unique value for each type of record.

The INTO option permits the user to specify that a copy of the data record is to be placed into a designated data field immediately after the READ statement. The data-name must not be defined in the file records description itself.

Also, the INTO phrase should not be used when the file has records of various sizes as indicated by their record descriptions. Any subscripting or indexing of data-name is evaluated after the data has been read but before it is moved to data-name. Afterward, the data is available in both the file record and data-name.

In the case of a blocked input file (such as disk files), not every READ statement performs a physical transmission of data from an external storage device; instead, READ may simply obtain the next logical record from an input buffer.

4.16 WRITE STATEMENT (Sequential I-O)

The general format of a WRITE statement is:

```
WRITE record-name FROM data-name-1
{ AFTER } ADVANCING { operand LINE(S) }
{ BEFORE }           { PAGE }
```

Ignoring the ADVANCING option for the moment, we proceed to explain the main functions of the WRITE statement.

In COBOL, file output is achieved by execution of the WRITE statement. Depending on the device assigned, "written" output may take the form of printed matter or magnetic recording on a floppy disk storage medium. The user is reminded also that you READ file-name, but you WRITE record-name. The associated file must be open in the OUTPUT mode at time of execution of a WRITE statement.

Record-name must be one of the level 01 records defined for an output file, and may be qualified by the filename. The execution of the WRITE statement releases the logical record to the file and updates its FILE STATUS item, if one is specified.

If the data to be output has been developed in Working-Storage or in another area (for example, in an input file's record area), the FROM suffix permits the user to stipulate that the designated data (data-name-1) is to be copied into the record-name area and then output from there. Record-name and data-name-1 must refer to separate storage areas.

The ADVANCING option is restricted to line printer output files, and permits the programmer to control the line spacing on the paper in the printer. Operand is either an unsigned integer literal or data-name; values from 0 to 60 are permitted:

<u>Integer</u>	<u>Carriage Control Action</u>
0	No spacing
1	Normal single spacing
2	Double spacing
3	Triple spacing
.	.
:	:
.	.

Single spacing (i.e., "after advancing 1 line") is assumed if there is no BEFORE or AFTER option in the WRITE statement.

Use of the key word AFTER implies that the carriage control action precedes printing a line, whereas use of BEFORE implies that writing precedes the carriage control action. If PAGE is specified, the data is printed BEFORE or AFTER the device is repositioned

to the next physical page.

When an attempt is made to write beyond the externally defined boundaries of a sequential file, a Declarative procedure will be executed (if available) and the FILE STATUS (if available) will indicate a boundary violation. If neither is available, a runtime error occurs.

4.17 CLOSE STATEMENT (Sequential I-O)

Upon completion of the processing of a file, a CLOSE statement must be executed, causing the system to make the proper disposition of the file. Whenever a file is closed, or has never been opened, READ, REWRITE, or WRITE statements cannot be executed properly; a runtime error would occur, aborting the run.

The general format of the CLOSE statement is:

CLOSE {file-name [WITH LOCK]} ...

If the LOCK suffix is used, the file is not re-openable during the current job. If LOCK is not specified immediately after a file-name, then that file may be re-opened later in the program, if the program logic dictates the necessity.

An attempt to execute a CLOSE statement for a file that is not currently open is a runtime error, and causes execution to be discontinued.

Examples of CLOSE statements:

CLOSE MASTER-FILE-IN WITH LOCK, WORK-FILE;
CLOSE PRINT-FILE, TAX-RATE-FILE, JOB-PARAMETERS WITH LOCK

4.18 REWRITE STATEMENT (Sequential I-O)

The REWRITE statement replaces a logical record on a sequential DISK file. The general format is:

REWRITE record-name [FROM data-name]

Record-name is the name of a logical record in the File Section of the Data Division and may be qualified. Record-name and data-name must refer to

separate storage areas.

At the time of execution of this statement, the file to which record-name belongs must be open in the I-O mode (see OPEN, Section 4.14).

If a FROM part is included in this statement, the effect is as if MOVE data-name TO record-name were executed just prior to the REWRITE.

Execution of REWRITE replaces the record that was accessed by the most recent READ statement; said prior READ must have been completed successfully, otherwise a run-time error terminates execution.

4.19 GENERAL NOTE ON I/O ERROR HANDLING

If an I/O error occurs, the file's FILE STATUS item, if one exists, is set to the appropriate two-character code, otherwise it assumes the value "00".

If an I/O error occurs and is of the type that is pertinent to an AT END or INVALID KEY clause, then the imperative statements in such a clause, if present on the statement that gave rise to the error, are executed. But, if there is not an appropriate clause (such clauses may not appear on Open or Close, for example, and are optional for other I/O statements), then the logic of program flow is as follows:

1. If there is an associated Declaratives ERROR procedure (see Section 9), it is performed automatically; user-written logic must determine what action is taken because of the existence of the error. Upon return from the ERROR procedure, normal program flow to the next sentence (following the I/O statement) is allowed.
2. If no Declaratives ERROR procedure is applicable but there is an associated FILE STATUS item, it is presumed that the user may base actions upon testing the STATUS item, so normal flow to the next sentence is allowed.

4.21 STRING STATEMENT

The STRING statement allows concatenation of multiple sending data item values into a single receiving item. The general format of this statement is:

STRING { operand-1... DELIMITED BY { operand-2 } } ...

INTO identifier-1 [WITH POINTER identifier-2]

[ON OVERFLOW imperative-statement]

In this format, the term operand means a non-numeric literal, one-character figurative constant, or data-name. "Identifier-1" is the receiving data-item name, which must be alphanumeric without editing symbols or the JUSTIFIED clause. "Identifier-2" is a counter and must be an elementary numeric integer data item of sufficient size (plus 1) to point to positions within identifier-1.

If no POINTER phrase exists, the default value of the logical pointer is one. The logical pointer value designates the beginning position of the receiving field into which data placement begins. During movement to the receiving field, the criteria for termination of an individual source are controlled by the "DELIMITED BY" phrase:

DELIMITED BY SIZE: the entire source field is moved (unless the receiving field becomes full)

DELIMITED BY operand-2: the character string specified by operand-2 is a "Key" which, if found to match a like-numbered succession of sending characters, terminates the function for the current sending operand (and causes automatic switching to the next sending operand, if any).

If at any point the logical pointer (which is automatically incremented by one for each character stored into identifier-1) is less than one or greater than the size of identifier-1, no further data movement occurs, and the imperative statement given in the OVERFLOW phrase (if any) is executed. If there is no OVERFLOW phrase, control is

transferred to the next executable statement.

There is no automatic space fill into any position of identifier-1. That is, unaccessed positions are unchanged upon completion of the STRING statement.

Upon completion of the STRING statement, if there was a POINTER phrase, the resultant value of identifier-2 equals its original value plus the number of characters moved during execution of the STRING statement.

4.22 UNSTRING STATEMENT

The UNSTRING statement causes data in a single sending field to be separated into subfields that are placed into multiple receiving fields. The general format of the statement is:

UNSTRING identifier-1

[DELIMITED BY [ALL] operand-1 [OR [ALL] operand-2] ...]

INTO {identifier-2 [DELIMITER IN identifier-3]
[COUNT IN identifier-4]} ...

[WITH POINTER identifier-5]

[TALLYING IN identifier-6]

[ON OVERFLOW imperative-statement]

Criteria for separation of subfields may be given in the "DELIMITED BY" phrase. Each time a succession of characters matches one of the non-numeric literals, one-character figurative constants, or data-item values named by operand-i, the current collection of sending characters is terminated and moved to the next receiving field specified by the INTO-clause. When the ALL phrase is specified, more than one contiguous occurrence of operand-i in identifier-1 is treated as one occurrence.

When two or more delimiters exist, an 'OR' condition exists. Each delimiter is compared to the sending field in the order specified in the UNSTRING statement.

Identifier-1 must be a group or character string (alphanumeric) item. When a data-item is employed as any operand-i, that operand must also be a group

or character string item.

Receiving fields (identifier-2) may be any of the following types of items:

1. an unedited alphabetic item
2. a character-string (alphanumeric) item
3. a group item
4. an external decimal item (numeric, usage DISPLAY) whose PICTURE does not contain any P character.

When any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled depending on its type. If there is a "DELIMITED BY" phrase in the UNSTRING statement, then there may be "DELIMITER IN" phrases following any receiving item (identifier-2) mentioned in the INTO clause. In this case, the character(s) that delimit the data moved into identifier-2 are themselves stored in identifier-3, which should be an alphanumeric item. Furthermore, if a "COUNT IN" phrase is present, the number of characters that were moved into identifier-2 is moved to identifier-4, which must be an elementary numeric integer item.

If there is a "POINTER" phrase, then identifier-5 must be an integer numeric item, and its initial value becomes the initial logical pointer value (otherwise, a logical pointer value of one is assumed). The examination of source characters begins at the position in identifier-1 specified by the logical pointer; upon completion of the UNSTRING statement, the final logical pointer value will be copied back into identifier-5.

If at any time the value of the logical pointer is less than one or exceeds the size of identifier-1, then overflow is said to occur and control passes over to the imperative statements given in the "ON OVERFLOW" clause, if any.

Overflow also occurs when all receiving fields have been filled prior to exhausting the source field.

During the course of source field scanning (looking

for matching delimiter sequences), a variable length character string is developed which, when completed by recognition of a delimiter or by acquiring as many characters as the size of the current receiving field can hold, is then moved to the current receiving field in the standard MOVE fashion.

If there is a "TALLYING IN" phrase, identifier-6 must be an integer numeric item. The number of receiving fields acted upon, plus the initial value of identifier-6, will be produced in identifier-6 upon completion of the UNSTRING statement.

Any subscripting or indexing associated with identifier-1, 5, or 6 is evaluated only once at the beginning of the UNSTRING statement. Any subscripting associated with operands-i or identifier-2, 3, 4 is evaluated immediately before access to the data-item.

4.23 DYNAMIC DEBUGGING STATEMENTS

The execution TRACE mode may be set or reset dynamically. When set, procedure-names are printed in the order in which they are executed.

Execution of the READY TRACE statements sets the trace mode to cause printing of every section and paragraph name each time it is entered. The RESET TRACE statement inhibits such printing. A printed list of procedure-names in the order of their execution is invaluable in detection of a program malfunction; it aids in detection of the point at which actual program flow departed from the expected program flow.

Another debugging feature may be required in order to reveal critical data values at specifically designated points in the procedure. The EXHIBIT statement provides this facility.

The statement form

```
EXHIBIT NAMED { literal }  
               { data-name }...
```

produces a printout of values of the indicated literal, or data items in the format data-name = value.

Statements EXHIBIT, READY TRACE and RESET TRACE are extensions to ANS-74 standard COBOL designed to provide a convenient aid to program debugging.

Programming Note: It is often desirable to include such statements on source lines that contain D in column 7, so that they are ignored by the compiler unless WITH DEBUGGING MODE is included in the SOURCE-COMPUTER paragraph.

CHAPTER 5

Inter-Program Communication

Separately compiled COBOL program modules may be combined into a single executable program. Inter-program communication is made possible through the use of the LINKAGE Section of the Data Division (which follows the Working-Storage Section) and by the CALL statement and the USING list appendage to the Procedure Division header of a subprogram module. The Linkage section describes data made available in memory from another program module. Record description entries in the LINKAGE section provide data-names by which data-areas reserved in memory by other programs may be referenced. Entries in the LINKAGE section do not reserve memory areas because the data is assumed to be present elsewhere in memory, in a CALLing program.

Any Record Description clause may be used to describe items in the LINKAGE Section as long as the VALUE clause is not specified for other than level 88 items.

5.1 USING LIST APPENDAGE TO PROCEDURE HEADER

The Procedure Division header of a CALLable subprogram is written as

PROCEDURE DIVISION [USING data-name ...].

Each of the data-name operands is an entry in the Linkage Section of the subprogram, having level 77 or 01. Addresses are passed from an external CALL in one-to-one correspondence to the operands in the USING list of the Procedure header so that data in the calling program may be manipulated in the subprogram. No data-name may appear more than once in the USING phrase.

5.2 CALL STATEMENT

The CALL statement format is

CALL literal USING data-name ...

Literal is a subprogram name defined as the PROGRAM-ID of a separately compiled program, and is non-numeric. Data names in the USING list are made available to the called subprogram by passing addresses to the subprogram; these addresses are assigned to the Linkage Section items declared in the USING list of that subprogram. Therefore the number of data-names specified in matching CALL and Procedure Division USING lists must be identical.

NOTE

Correspondence between caller and callee lists is by position, not by identical spelling of names.

5.3 EXIT PROGRAM STATEMENT

The EXIT PROGRAM statement, appearing in a called subprogram, causes control to be returned to the next executable statement after CALL in the calling program. This statement must be a paragraph by itself.

CHAPTER 6

Table Handling by the Indexing Method

In addition to the capabilities of subscripting described in Chapter 3, COBOL provides the Indexing method of table handling.

6.1 INDEX-NAMES AND INDEX ITEMS

An index-name is declared not by the usual method of level number, name, and data description clauses, but implicitly by appearance in the "INDEXED BY index-name" appendage to an OCCURS clause. Thus, an index-name is equivalent to an index data-item (USAGE IS INDEX), although defined differently. An index-name must be uniquely named.

An index data item may only be referred to by a SET or SEARCH statement, a CALL statement's USING list or a Procedure header USING list; or used in a relation condition or as the variation item in a PERFORM VARYING statement. In all cases the process is equivalent to dealing with a binary word integer subscript. Index-name must be initialized to some value before use via SET, SEARCH or PERFORM.

6.2 SET STATEMENT

The SET statement permits the manipulation of index-names, index items, or binary subscripts for table-handling purposes. There are two formats.

Format 1:

$$\underline{\text{SET}} \quad \left\{ \begin{array}{l} \text{index-name-1} \\ \text{index-item-1} \\ \text{data-name-1} \end{array} \right\} \quad \dots \quad \underline{\text{TO}} \quad \left\{ \begin{array}{l} \text{index-name-2} \\ \text{index-item-2} \\ \text{data-name-2} \\ \text{integer-2} \end{array} \right\}$$

Format 2:

$$\underline{\text{SET}} \quad \text{index-name-3} \quad \dots \quad \left\{ \begin{array}{l} \underline{\text{UP BY}} \\ \underline{\text{DOWN BY}} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{integer-4} \\ \text{data-name-4} \end{array} \right\}$$

Format 1 is equivalent to moving the "TO" value (e.g., integer-2) to multiple receiving fields written immediately after the verb SET.

Format 2 is equivalent to reduction (DOWN) or increase (UP) applied to each of the quantities written immediately after the verb SET: the amount of the reduction or increase is specified by a name or value immediately following the word BY.

In any SET statement, data-names are restricted to integer items.

6.3 RELATIVE INDEXING

A user reference to an item in a table controlled by an OCCURS clause is expressed with a proper number of subscripts (or indexes), separated by commas. The whole is enclosed in matching parentheses, for example:

```
TAX-RATE (BRACKET, DEPENDENTS)  
XCODE (I, 2)
```

where subscripts are ordinary integer decimal data-names, or integer constants, or binary integer (COMPUTATIONAL or INDEX) items, or index-names. Subscripts may be qualified, but not, themselves, subscripted. A subscript may be signed, but if so, it must be positive. The lowest acceptable value is 1, pointing to the first element of a table. The highest permissible value is the maximum number of occurrences of the item as specified in its OCCURS clause.

A further capability exists, called relative indexing. In this case, a "subscript" is expressed as

name ± integer constant

where a space must be on either side of the plus or minus, and "name" may be any proper index-name. Example:

```
XCODE (I + 3, J - 1).
```

6.4 SEARCH STATEMENT --Format 1

A linear search of a table may be done using the SEARCH statement. The general format is:

```
SEARCH table [VARYING identifier | index-name]  
    [AT END imperative-statement-1]  
    { WHEN Condition-1      { NEXT SENTENCE  
      { imperative-statement-2 } } } ...
```

"Table" is the name of a data-item having an OCCURS clause that includes an INDEXED-BY list; "table" must be written without subscripts or indexes because the nature of the SEARCH statement causes automatic variation of an index-name associated with a particular table.

There are four possible "varying" cases:

1. NO VARYING phrase -- the first-listed index-name for the table is varied.
2. VARYING index-name-in-a-different-table -- the first-listed index-name in the table's definition is varied, implicitly, and the index-name listed in the VARYING phrase is varied in like manner, simultaneously.
3. VARYING index-name-defined-for-table -- this specific index-name is the only one varied.
4. VARYING integer-data-item-name -- both this data-item and the first-listed index-name for table are varied, simultaneously.

The term variation has the following interpretation:

1. The initial value is assumed to have been established by an earlier statement such as SET.
2. If the initial value exceeds the maximum declared in the applicable OCCURS clause, the SEARCH operation terminates at once; and if an AT END phrase exists, the associated imperative statement-1 is executed.

3. If the value of the index is within the range of valid indexes (1,2,... up to and including the maximum number of occurrences), then each WHEN-condition is evaluated until one is true or all are found to be false. If one is true, its associated imperative statement is executed and the SEARCH operation terminates. If none is true, the index is incremented by one and step (3) is repeated. Note that incrementation of index applies to whatever item and/or index is selected according to rules 1-4.

If the table is subordinate to another table, an index-name must be associated with each dimension of the entire table via INDEXED BY phrases in all the OCCURS clauses. Only the index-name of the SEARCH table is varied (along with another "VARYING" index-name or data-item). To search an entire two-or three-dimensional table, a SEARCH must be executed several times with the other index-names set appropriately each time, probably with a PERFORM, VARYING statement.

The logic of a Format 1 SEARCH is depicted at the end of this chapter.

6.5 SEARCH STATEMENT --Format 2

Format 2 SEARCH statements deal with tables of ordered data. The general format of such a SEARCH ALL statement is:

```
SEARCH ALL table [AT END imperative-statement-1...]  
    WHEN condition { imperative-statement-2... }  
                   { NEXT SENTENCE }
```

Only one WHEN clause is permitted, and the following rules apply to the condition:

1. Only simple relational conditions or condition-names may be employed, and the subject must be properly indexed by the first index-name associated with table (along with sufficient other indexes if multiple OCCURS clauses apply). Furthermore, each subject data-name (or the data-name associated with condition-name) in the condition must be mentioned in the KEY clause of the table. The KEY clause is an appendage to

the OCCURS clause having the following format:

ASCENDING | DESCENDING KEY IS data-name ...

where data-name is the name defined in this Data Description entry (following level number) or one of the subordinate data-names. If more than one data-name is given, then all of them must be the names of entries subordinate to this group item. The KEY phrase indicates that the repeated data is arranged in ascending or descending order according to the data-names which are listed (in any given KEY phrase) in decreasing order of significance. More than one KEY phrase may be specified.

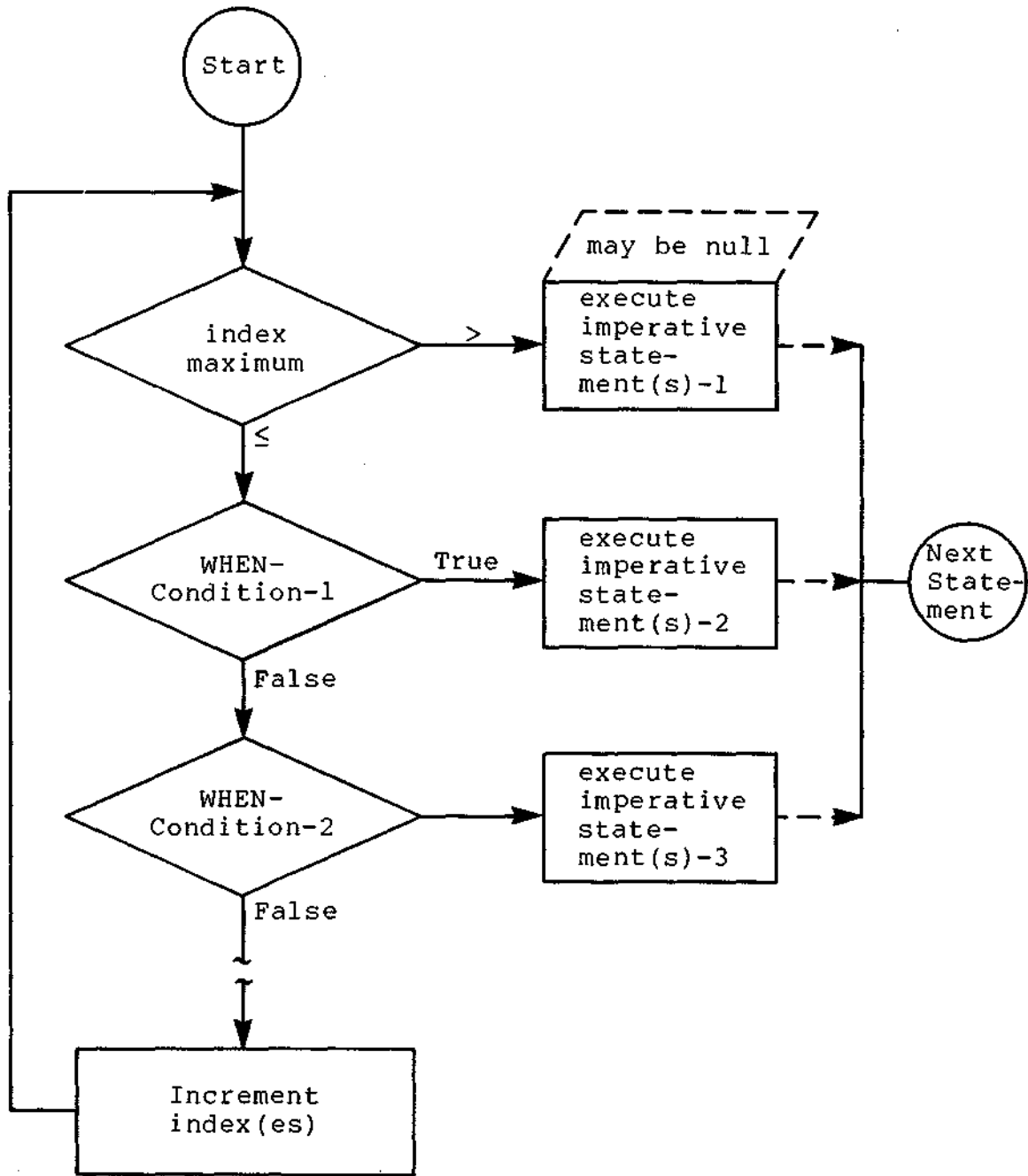
2. In a simple relational condition, only the equality test (using relation = or IS EQUAL TO) is permitted.
3. Any condition-name variable (Level 88 items) must be defined as having only a single value.
4. The condition may be compounded by use of the Logical connector AND, but not OR.
5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or an identifier; the identifier must NOT be referenced in the KEY clause of the table or be indexed by the first index-name associated with the table. (The term identifier means data-name, including any qualifiers and/or subscripts or indexes.)

Failure to conform to these restrictions may yield unpredictable results. Unpredictable results also occur if the table data is not ordered in conformance to the declared KEY clauses, or if the keys referenced in the WHEN-condition are not sufficient to identify a unique table element.

In a Format 2 SEARCH, a nonserial type of search operation may take place, relying upon the declared ordering of data. The initial setting of the index-name for table is ignored and its setting is varied automatically during the searching, always within the bounds of the maximum number of occurrences. If the condition (WHEN) cannot be satisfied for any valid index value, control is passed to imperative-

statement-1, if the AT END clause is present, or to the next executable sentence in the case of no AT END clause.

If all the simple conditions in the single WHEN-condition are satisfied, the resultant index value indicates an occurrence that allows those conditions to be satisfied, and control passes to imperative-statement-2. Otherwise the final setting is not predictable.



Logic Diagram for Format 1 SEARCH

CHAPTER 7
Indexed Files

7.1 DEFINITION OF INDEXED FILE ORGANIZATION

An indexed-file organization provides for recording and accessing records of a "data base" by keeping a directory (called the control index) of pointers that enable direct location of records having particular unique key values. An indexed file must be assigned to DISK in its defining SELECT sentence.

A file whose organization is indexed can be accessed either sequentially, dynamically or randomly.

Sequential access provides access to data records in ascending order of RECORD KEY values.

In the random access mode, the order of access to records is controlled by the programmer. Each record desired is accessed by placing the value of its key in a key data item prior to an access statement.

In the dynamic access mode, the programmer's logic may change from sequential access to random access, and vice versa, at will.

7.2 SYNTAX CONSIDERATIONS

In the Environment Division, the SELECT entry must specify ORGANIZATION IS INDEXED, and the ACCESS clause format is

ACCESS MODE IS SEQUENTIAL | RANDOM | DYNAMIC.

Assign, Reserve, and File Status clause formats are identical to those specified in Section 2.2.1 of this manual.

In the FD entry for an INDEXED file, both LABEL RECORDS STANDARD and a VALUE OF FILE-ID clause must appear. The formats of Section 3.13 apply, except that only the DISK-related forms are applicable.

7.2.1 RECORD KEY CLAUSE

The general format of this clause, which is required, is:

RECORD KEY IS data-name-1

where data-name-1 is an item defined within the record descriptions of the associated file description, and is a group item, an elementary alphanumeric item or a decimal field. A decimal key must have no P characters in its PICTURE, and it may not have a SEPARATE sign. No record key may be subscripted. The maximum KEY length is 60 characters.

If random access mode is specified, the value of data-name-1 designates the record to be accessed by the next DELETE, READ, REWRITE or WRITE statement. Each record must have a unique record key value.

7.2.2 FILE STATUS REPORTING

If a FILE STATUS clause appears in the Environment Division for an Indexed organization file, the designated two-character data item is set after every I-O statement. The following table summarizes the possible settings.

Status Data Item LEFT Character	Status Data Item RIGHT Character				
	No Further Description (0)	Sequence Error (1)	Duplicate Key (2)	No Record Found (3)	Disk Space Full (4)
Successful Completion (0)	X				
At End (1)	X				
Invalid Key (2)		X	X	X	X
Permanent Error(3)	X				

Sequence error arises if access mode is sequential when WRITES do not occur in ascending sequence for an Indexed file, or the key is altered prior to REWRITE. The other settings are self-explanatory. The left character may also be '9' for implementor-defined errors; see the User's Guide for an explanation of these.

Note that "Disk Space Full" occurs with Invalid Key (2) for Indexed and Relative file handling, whereas it occurred with "Permanent Error" (3) for sequential files.

If an error occurs at execution time and no AT END or INVALID KEY statements are given and no appropriate Declarative ERROR section is supplied and no FILE STATUS is specified, the error will be displayed on the Console and the program will terminate. See Section 4.19.

7.3 PROCEDURE DIVISION STATEMENTS FOR INDEXED FILES

The syntax of the OPEN statement (Section 4.14) also applies to Indexed organized files, except EXTEND is inapplicable.

The following table summarizes the available statement types and their permissibility in terms of ACCESS mode and OPEN option in effect. Where X appears, the statement is permissible, otherwise it is not valid under the associated ACCESS mode and OPEN option.

ACCESS MODE IS	Procedure Statement	OPEN Option in Effect		
		Input	Output	I-O
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
RANDOM	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
DYNAMIC	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

In addition to the above statements, CLOSE is permissible under all conditions; the same format shown in Section 4.17 is used.

7.4 READ STATEMENT

Format 1 (Sequential Access):

READ file-name [NEXT] RECORD [INTO data-name-1]
 [AT END imperative-statement ...]

Format 2 (Random or Dynamic Access):

READ file-name RECORD [INTO data-name-1][KEY IS data-name-2]
 [INVALID KEY imperative-statement...]

Format 1 must be used for all files having sequential-access mode. Format 1 with the NEXT option is used for sequential reads of a DYNAMIC access mode file. The AT END clause is executed when the logical end-of-file condition arises. If this clause is not written in the source statement, an appropriately assigned Declaratives ERROR section is given control at end-of-file time, if available.

Format 2 is used for files in random-access mode or

for files in dynamic-access mode when records are to be retrieved randomly.

In format 2, the INVALID KEY clause specifies action to be taken if the access key value does not refer to an existent key in the file. If the clause is not given, the appropriate Declaratives ERROR section, if supplied, is given control.

The optional "KEY IS" clause must designate the record key item declared in the file's SELECT entry. For non-sequential access, if no "KEY IS" clause is written in a READ statement, then the prime record key is assumed to be the key of record. The user must ensure that a valid key value is in the designated key field prior to execution of a random-access READ.

The rules for sequential files regarding the INTO phrase apply here as well.

7.5 WRITE STATEMENT

The WRITE statement releases a logical record for an output or input-output file; its general format is:

```
WRITE record-name [FROM data-name-1]  
[INVALID KEY imperative-statement...]
```

Just prior to executing the WRITE statement, a valid (unique) value must be in that portion of the record-name (or data-name-1 if FROM appears in the statement) which serves as RECORD KEY.

In the event of an improper key value, the imperative statements are executed if the INVALID KEY clause appears in the statement; otherwise an appropriate Declaratives ERROR section is invoked, if applicable. The INVALID KEY condition arises if:

1. for sequential access, key values are not ascending from one WRITE to the next WRITE;
2. the key value is not unique;
3. the allocated disk space is exceeded.

7.6 REWRITE STATEMENT

The REWRITE statement logically replaces an existing record; the format of the statement is:

```
REWRITE record-name [FROM data-name]  
[INVALID KEY imperative-statement...]
```

For a file in sequential-access mode, the last READ statement must have been successful in order for a REWRITE statement to be valid. If the value of the record key in record-name (or corresponding part of data-name, if FROM appears in the statement) does not equal the key value of the immediately previous read, or if that previous read was unsuccessful, then the invalid key condition exists and the imperative statements are executed, if present; otherwise an applicable Declaratives ERROR section is executed, if available.

For a file in a random or dynamic access mode, the record to be replaced is specified by the record key; no previous READ is necessary. The INVALID KEY condition exists when the record key's value does not equal that of any record stored in the file.

7.7 DELETE STATEMENT

The DELETE statement logically removes a record from the Indexed file. The general format of the statement is:

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

For a file in the sequential access mode, the last input-output statement executed for file-name would have been a successful READ statement. The record that was read is deleted. Consequently, no INVALID KEY phrase should be specified for sequential-access mode files.

For a file having random or dynamic access mode, the record deleted is the one associated with the record key; if there is no such matching record, the invalid key condition exists, and control passes to the imperative statements in the INVALID KEY clause, or to an applicable Declarative ERROR section if no INVALID KEY clause exists.

7.8 START STATEMENT

The START statement enables an Indexed organization file to be positioned for reading at a specified key value. This is permitted for files open in either sequential or dynamic access modes. The format of this statement is:

START file-name [KEY IS { GREATER THAN
NOT LESS THAN } data-name]
EQUAL TO

[INVALID KEY imperative statement...]

Data-name must be the declared record key and the value to be matched by a record in the file must be pre-stored in the data-name. When executing this statement, the file must be open in the input or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no matching record is found, the imperative statements in the INVALID KEY clause are executed, or an appropriate Declaratives ERROR section is executed.

CHAPTER 8

Relative Files

8.1 DEFINITION OF RELATIVE FILE ORGANIZATION

Relative organization is restricted to disk-based files. Records are differentiated on the basis of a RELATIVE RECORD number which ranges from 1 to 32,767, or to a lesser maximum for a smaller file. Unlike the case of an Indexed file, where the identifying key field occupies a part of the data record, relative record numbers are conceptual and are not embedded in the data records.

A relative-organized file may be accessed either sequentially, dynamically or randomly. In sequential access mode, records are accessed in the order of ascending record numbers.

In random access mode, the sequence of record access is controlled by the program, by placing a number in a relative key item. In dynamic access mode, the program may inter-mix random and sequential access verb forms at will.

8.2 SYNTAX CONSIDERATIONS

In the Environment Division, the SELECT entry must specify ORGANIZATION IS RELATIVE, and the ACCESS clause format is

ACCESS MODE IS SEQUENTIAL | RANDOM | DYNAMIC.

Assign, Reserve, and File Status clause formats are identical to those used for sequentially- or indexed-organized files. The values of STATUS Key 2 when STATUS Key 1 equals '2' are:

- '2' for attempt to WRITE a duplicate key
- '3' for nonexistent record
- '4' for disk space full

In the associated FD entry, STANDARD labels must be declared and a VALUE OF FILE-ID clause must be included.

8.2.1 RELATIVE KEY CLAUSE

In addition to the usual clauses in the SELECT entry, a clause of the form

RELATIVE KEY IS data-name-1

is required for random or dynamic access mode. It is also required for sequential-access mode, if a START statement exists for such a file.

Data-name-1 must be described as an unsigned binary integer item not contained within any record description of the file itself. Its value must be positive and nonzero.

8.3 PROCEDURE DIVISION STATEMENT FOR RELATIVE FILES

Within the Procedure Division, the verbs OPEN, CLOSE, READ, WRITE, REWRITE, DELETE and START are available, used as for files whose organization is indexed. (Therefore the charts in Sections 7.2.2 and 7.3 also apply to RELATIVE files.) The statement formats for OPEN and CLOSE (see Sections 4.14 and 4.17) are applicable to Relative files, except for the "EXTEND" phrase.

8.4 READ STATEMENT

Format 1:

READ file-name [NEXT] RECORD [INTO data-name]
[AT END imperative statement...]

Format 2:

READ file-name RECORD [INTO data-name]
[INVALID KEY imperative statement...]

Format 1 must be used for all files in sequential

access mode. The NEXT phrase must be present to achieve sequential access if the file's declared mode of access is Dynamic. The AT END clause, if given, is executed when the logical end-of-file condition exists, or, if not given, the appropriate Declaratives ERROR section is given control, if available.

Format 2 is used to achieve random access with declared mode of access either Random or Dynamic.

If a Relative Key is defined (in the file's SELECT entry), successful execution of a format 1 READ statement updates the contents of the RELATIVE KEY item ("data-name-1") so as to contain the record number of the record retrieved.

For a format 2 READ, the record that is retrieved is the one whose relative record number is pre-stored in the RELATIVE KEY item. If no such record exists, however, the INVALID KEY condition arises, and is handled by (a) the imperative statements given in the INVALID KEY portion of the READ, or (b) an associated Declaratives section.

The rules for sequential files regarding the INTO phrase apply here as well.

8.5 WRITE STATEMENT

The format of the WRITE statement is the same for a Relative file as for an Indexed file:

```
WRITE record-name [FROM data-name] [INVALID  
imperative statement...]
```

If access mode is sequential, then completion of a WRITE statement causes the relative record number of the record just output to be placed in the RELATIVE KEY item.

If access mode is random or dynamic, then the user must pre-set the value of the RELATIVE KEY item in order to assign the record an ordinal (relative) number. The INVALID KEY condition arises if there already exists a record having the specified ordinal number, or if the disk space is exceeded.

8.6 REWRITE STATEMENT

The format of the REWRITE statement is the same for a Relative file as for an Indexed file:

```
REWRITE record-name [FROM data-name]  
      [INVALID KEY imperative statement ...]
```

For a file in sequential access mode, the immediately previous action would have been a successful READ; the record thus previously made available is replaced in the file by executing REWRITE. If the previous READ was unsuccessful, a run-time error will terminate execution. Therefore, no INVALID KEY clause is allowed for sequential access.

For a file with dynamic or random access mode declared, the record that is replaced by executing REWRITE is the one whose ordinal number is pre-set in the RELATIVE KEY item. If no such item exists, the INVALID KEY condition arises.

8.7 DELETE STATEMENT

The format of the DELETE statement is the same for a Relative file as for an Indexed file:

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

For a file in a sequential access mode, the immediately previous action would have been a successful READ statement; the record thus previously made available is logically removed (or made inaccessible). If the previous READ was unsuccessful, a run-time error will terminate execution. Therefore, an INVALID KEY clause may not be specified for sequential-access mode files.

For a file with dynamic or random access mode declared, the removal action pertains to whatever record is designated by the value in the RELATIVE KEY item. If no such numbered record exists, the INVALID KEY condition arises.

8.8 START STATEMENT

The format of the START statement is the same for a Relative file as for an Indexed file:

START file-name [KEY IS { GREATER THAN
NOT LESS THAN } data-name-1]
EQUAL TO

[INVALID KEY imperative statement...]

Execution of this statement specifies the beginning position for reading operations; it is permissible only for a file whose access mode is defined as sequential or dynamic.

Data-name may only be that of the previously declared RELATIVE KEY item, and the number of the relative record must be stored in it before START is executed. When executing this statement, the associated file must be currently open in INPUT or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no such relative record is found, the imperative statements in the INVALID KEY clause are executed, or an appropriate Declaratives ERROR section is executed.

CHAPTER 9

DECLARATIVES and the USE SENTENCE

The Declaratives region provides a method of including procedures that are executed not as part of the sequential coding written by the programmer, but rather when a condition that cannot normally be tested by the programmer occurs.

Although the system automatically handles checking and creation of standard labels and executes error recovery routines in the case of input/output errors, additional procedures may be specified by the COBOL programmer.

Since these procedures are executed only at the time an error in reading or writing occurs, they cannot appear in the regular sequence of procedural statements. They must be written at the beginning of the Procedure Division in a subdivision called DECLARATIVES. Related procedures are preceded by a USE sentence that specifies their function. A declarative section ends with the occurrence of another section-name with a USE sentence or with the key words END DECLARATIVES.

The key words DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a period. No other text may appear on the Declaratives at the front of the Procedure Division.

PROCEDURE DIVISION.

DECLARATIVES.

{section-name SECTION. USE sentence.

{paragraph-name. {sentence}...} ...} ...

END DECLARATIVES.

The USE sentence defines the applicability of the associated section of coding.

A USE sentence, when present, must immediately follow a section header in the Declarative portion of the Procedure Division and must be followed by a period followed by a space. The remainder of the section must consist of zero, one or more procedural paragraphs that define the

procedures to be used. The USE sentence itself is never executed; rather, it defines the conditions for the execution of the USE procedure. The general format of the USE sentence is

USE AFTER STANDARD EXCEPTION | ERROR PROCEDURE

ON {file-name... | INPUT | OUTPUT | I-O | EXTEND}.

The words EXCEPTION and ERROR may be used interchangeably. The associated declarative section is executed (by the PERFORM mechanism) after the standard I-O recovery procedures for the files designated, or after the INVALID KEY or AT END condition arises on a statement lacking the INVALID KEY or AT END clause. A given file-name may not be associated with more than one declarative section.

Within a declarative section there must be no reference to any nondeclarative procedure. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declaratives section, except that PERFORM statements may refer to a USE statement and its procedures; but in a range specification (see PERFORM, Section 4.10) if one procedure-name is in a Declarative Section, then the other must be in the same Declarative Section.

An exit from a Declarative Section is inserted by the compiler following the last statement in the section. All logical program paths within the section must lead to the exit point.

APPENDIX I

Advanced Forms of Conditions

Evaluation Rules for Compound Conditions

1. Evaluation of individual simple conditions (relation, class, condition-name, and sign test) is done first.
2. AND-connected simple conditions are evaluated next as a single result.
3. OR and its adjacent conditions (or previously evaluated results) are then evaluated.

EXAMPLES:

1. A < B OR C = D OR E NOT > F

The evaluation is equivalent to (A<B) OR (C=D) OR (E<F) and is true if any of the three individual parenthesized simple conditions is true.

2. WEEKLY AND HOURS NOT = 0

The evaluation is equivalent, after expanding level 88 condition-name WEEKLY, to

(PAY-CODE = 'W') AND (HOURS ≠ 0)

and is true only if both the simple conditions are true.

3. A = 1 AND B = Z AND G > -3

OR P NOT EQUAL TO "SPAIN"

is evaluated as

[(A = 1) AND (B = Z) AND (G > -3)]

OR (P ≠ "SPAIN")

If P = "SPAIN", the compound condition can only be true if all three of the following are true:

(c.1) A = 1
(c.2) B = 2
(c.3) G > -3

However, if P is not equal to "SPAIN", the compound condition is true regardless of the values of A, B and G.

Parenthesized Conditions

Parentheses may be written within a compound condition or parts thereof in order to take precedence in the evaluation order.

Example:

```
IF A = B AND (A = 5 OR A = 1)
   PERFORM PROCEDURE-44.
```

In this case, PROCEDURE-44 is executed if A = 5 OR A = 1 while at the same time A = B. In this manner, compound conditions may be formed containing other compound conditions, not just simple conditions, via the use of parentheses.

Abbreviated Conditions

For the sake of brevity, the user may omit the "subject" when it is common to several successive relational tests. For example, the condition A = 5 OR A = 1 may be written A = 5 OR = 1. This may also be written A = 5 OR 1, where both subject and relation being implied are the same.

Another example:

```
IF A = B OR < C OR Y
```

is a shortened form of

```
IF A = B OR A < C OR A < Y
```

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

1. If the item immediately following 'NOT' is a

- relational operator, then the 'NOT' participates as part of the relational operator;
2. otherwise, the beginning of a new, completely separate condition must follow 'NOT', not to be considered part of the abbreviated condition.

Caution: Abbreviations in which the subject and relation are implied are permissible only in relation tests; the subject of a sign test or class test cannot be omitted.

NOT, the Logical Negation Operator

In addition to its use as a part of a relation (e.g., IF A IS NOT = B), "NOT" may precede a condition. For example, the condition NOT (A = B OR C) is true when (A = B OR A = C) is false. The word NOT may precede a level 88 condition name, also.

APPENDIX II

Table of Permissible MOVE Operands

Source Operand	Receiving Operand in MOVE Statement					
	Numeric Integer	Numeric Non-integer	Numeric Edited	Alphanumeric Edited	Alphanumeric	Group
Numeric Integer	OK	OK	OK	OK (A)	OK (A)	OK (B)
Numeric Non-integer	OK	OK	OK			OK (B)
Numeric Edited				OK	OK	OK (B)
Alphanumeric Edited				OK	OK	OK (B)
Alphanumeric	OK (C)	OK (C)	OK (C)	OK	OK	OK (B)
Group	OK (B)	OK (B)	OK (B)	OK (B)	OK (B)	OK (B)

- KEY: (A) Source sign, if any, is ignored.
- (B) If the source operand or the receiving operand is a Group Item, the move is considered to be a Group Move. See Section 4.3 for a discussion of the effect of a Group Move.
- (C) Source is treated as an unsigned integer; source length may not exceed 31.

NOTE: No distinction is made in the compiler between alphabetic and alphanumeric; one should not move numeric items to alphabetic items and vice versa.

APPENDIX III

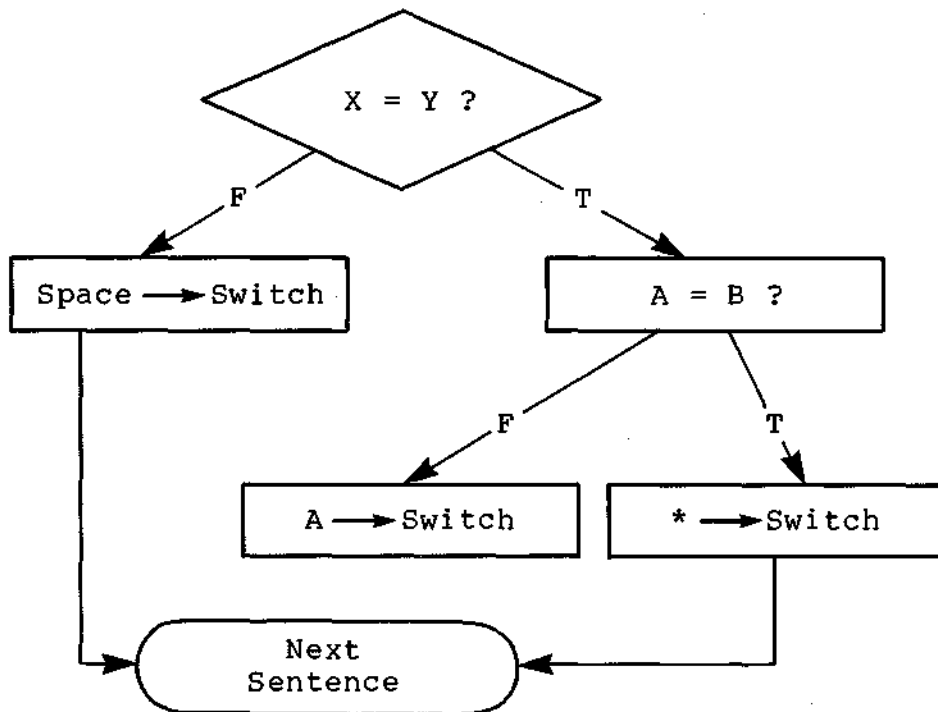
Nesting of IF Statements

A "nested IF" exists when, in a single sentence, more than one IF precedes the first ELSE.

Example:

```
IF X = Y IF A = B  
  MOVE "*" TO SWITCH  
ELSE MOVE "A" TO SWITCH  
ELSE MOVE SPACE TO SWITCH
```

The flow of the above sentence may be represented by a tree structure:



Another useful way of viewing nested IF structures is based on numbering IF and ELSE verbs to show their priority.

```

                IF1      X = Y
  true
  action1:      IF2      A = B
                true-action : MOVE "*" TO SWITCH
                ELSE2 false-action2 : MOVE "A" TO SWITCH
  ELSE1
                false-action1 : MOVE SPACE TO SWITCH.
  
```

The above illustration shows clearly the fact that IF2 is wholly nested within the true-action side of IF1.

The number of ELSEs in a sentence need not be the same as the number of IFs; there may be fewer ELSE branches.

Examples:

```

IF  M = 1      IF  K = 0
GO TO MIKO ELSE GO TO MNOT1.

IF  AMOUNT IS NUMERIC IF AMOUNT
IS ZERO GO TO CLOSE-OUT.
  
```

In the latter case, IF2 could equally well have been written as AND.

APPENDIX IV

ASCII Character Set
 For ANS-74 COBOL

<u>Character</u>	<u>Octal Value</u>	<u>Character</u>	<u>Octal Value</u>
A	101	Ø	60
B	102	1	61
C	103	2	62
D	104	3	63
E	105	4	64
F	106	5	65
G	107	6	66
H	110	7	67
I	111	8	70
J	112	9	71
K	113	(SPACE)	40
L	114	"	42
M	115	\$	44
N	116	' (non-ANSI)	47
O	117	(50
P	120)	51
Q	121	*	52
R	122	+	53
S	123	,	54
T	124	-	55
U	125	.	56
V	126	/	57
W	127	;	73
X	130	<	74
Y	131	=	75
Z	132	>	76

Plus-zero (zero with embedded positive sign); 173
 Minus-zero (zero with embedded negative sign); 175

APPENDIX V

Reserved Words

* words not used by COBOL-80

** additional words required by COBOL-80

ACCEPT	COPY
ACCESS	*CORR (ESPONDING)
ADD	COUNT
ADVANCING	CURRENCY
AFTER	DATA
ALL	DATE
ALPHABETIC	DATE-COMPILED
*ALSO	DATE-WRITTEN
ALTER	DAY
*ALTERNATE	DEBUGGING
AND	*DEBUG-CONTENTS
ARE	*DEBUG-ITEM
AREA (S)	*DEBUG-LINE
ASCENDING	*DEBUG-NAME
**ASCII	*DEBUG-SUB-1
ASSIGN	*DEBUG-SUB-2
AT	*DEBUG-SUB-3
AUTHOR	DECIMAL-POINT
	DECLARATIVES
**BEEP	DELETE
BEFORE	DELIMITED
BLANK	DELIMITER
BLOCK	DEPENDING
*BOTTOM	DESCENDING
BY	*DESTINATION
	*DE (TAIL)
CALL	*DISABLE
*CANCEL	**DISK
*CD	DISPLAY
*CF	DIVIDE
*CH	DIVISION
CHARACTER (S)	DOWN
*CLOCK-UNITS	*DUPLICATES
CLOSE	DYNAMIC
*CLOSE	
*CODE	*EGI
CODE-SET	ELSE
COLLATING	*EMI
*COLUMN	*ENABLE
COMMA	END
*COMMUNICATION	*END-OF-PAGE
COMP	*ENTER
COMPUTATIONAL	ENVIRONMENT
**COMPUTATIONAL-3	*EOP
**COMP-3	EQUAL
COMPUTE	ERROR
CONFIGURATION	*ESI
**CONSOLE	*EVERY
CONTAINS	EXCEPTION
*CONTROL (S)	

CROMEMCO COBOL REFERENCE MANUAL
Appendix V

**EXHIBIT	*LIMIT(S)
EXIT	*LINAGE
EXTEND	*LINAGE-COUNTER
	LINE(S)
FD	*LINE-COUNTER
FILE	LINKAGE
FILE-CONTROL	LOCK
**FILE-ID	LOW-VALUE(S)
FILLER	
*FINAL	MEMORY
FIRST	*MERGE
*FOOTING	*MESSAGE
FOR	MODE
FROM	MODULES
	MOVE
*GENERATE	*MULTIPLE
GIVING	MULTIPLY
GO	
GREATER	**NAMED
*GROUP	NATIVE
	NEGATIVE
*HEADING	NEXT
HIGH-VALUE(S)	*NO
	NOT
IDENTIFICATION	*NUMBER
IF	NUMERIC
IN	
INDEX	OBJECT-COMPUTER
INDEXED	OCCURS
INITIAL	OF
*INITIATE	OFF
INPUT	OMITTED
INPUT-OUTPUT	ON
INSPECT	OPEN
INSTALLATION	*OPTIONAL
INTO	OR
INVALID	ORGANIZATION
IS	OUTPUT
I-O	OVERFLOW
I-O-CONTROL	
	PAGE
JUST(IFIED)	*PAGE-COUNTER
	PERFORM
KEY	*PF
	*PH
	PIC(TURE)
LABEL	*PLUS
*LAST	POINTER
LEADING	POSITION
LEFT	POSITIVE
*LENGTH	**PRINTER
LESS	*PRINTING

PROCEDURE (S)	SEQUENCE
PROCEED	SEQUENTIAL
PROGRAM	SET
PROGRAM-ID	SIGN
**PROMPT	SIZE
	*SORT
*QUEUE	*SORT-MERGE
QUOTE (S)	*SOURCE
	SOURCE-COMPUTER
RANDOM	SPACES (S)
*RD	SPECIAL-NAMES
READ	STANDARD
**READY	STANDARD-1
*RECEIVE	START
RECORD (S)	STATUS
REDEFINES	STOP
*REEL	STRING
*REFERENCES	*SUB-QUEUE-1, 2, 3
RELATIVE	SUBTRACT
*RELEASE	*SUM
*REMAINDER	*SUPPRESS
*REMOVAL	*SYMBOLIC
*RENAMES	SYNC (HRONIZED)
REPLACING	
*REPORT (S)	*TABLE
*REPORTING	TALLYING
*RERUN	*TAPE
RESERVE	*TERMINAL
RESET	*TERMINATE
*RETURN	*TEXT
*REVERSED	THAN
*REWIND	THROUGH
REWRITE	THRU
*RF	TIME
*RH	TIMES
RIGHT	TO
ROUNDED	*TOP
RUN	**TRACE
	TRAILING
SAME	*TYPE
*SD	
SEARCH	*UNIT
SECTION	UNSTRING
SECURITY	UNTIL
*SEGMENT	UP
*SEGMENT-LIMIT	UPON
SELECT	USAGE
*SEND	USE
SENTENCE	USING
SEPARATE	

CROMEMCO COBOL REFERENCE MANUAL
Appendix V

VALUE(S)
VARYING
WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO((E)S)

+

-

*

/

**

>

<

=

APPENDIX VI

PERFORM with VARYING and AFTER Clauses

PERFORM range

VARYING identifier-1 FROM amount-1 BY amount-2
UNTIL condition-1

```
[ AFTER identifier-2 FROM amount-3 BY amount-4  
  UNTIL condition-2  
[ [AFTER identifier-3 FROM amount-5 BY amount-6]  
  UNTIL condition-3 ] ]
```

Identifier here means a data-name or index-name. Amount-1, -3, and -5 may be a data-name, index-name, or literal. Amount-2, -4, and -6 may be a data-name or literal only.

The operation of this complex PERFORM statement is equivalent to the following COBOL statements (example varying three items):

```
START-PERFORM.  
  MOVE amount-1 TO identifier-1  
  MOVE amount-3 TO identifier-2  
  MOVE amount-5 TO identifier-3.  
  
TEST-CONDITION-1.  
  IF condition-1 GO TO END-PERFORM.  
  
TEST-CONDITION-2.  
  IF condition-2  
    MOVE amount-3 TO identifier-2  
    ADD amount-2 TO identifier-1  
    GO TO TEST-CONDITION-1.  
  
TEST-CONDITION-3.  
  IF condition-3  
    MOVE amount-5 TO identifier-3  
    ADD amount-4 TO identifier-2  
    GO TO TEST-CONDITION-2.
```

```
PERFORM range  
ADD amount-6 TO identifier-3
```

GO TO TEST-CONDITION-3.

END-PERFORM. Next statement.

NOTE

If any identifier above were an index-name, the associated MOVE would instead be a SET (TO form), and the associated ADD would be a SET (UP form).

SECTION 1

Compiling COBOL Programs

1.1 COBOL Command Scanner

To tell the COBOL compiler what to compile and with which options, it is necessary to input a command string, which is read by the COBOL command scanner. Those familiar with Cromemco's FORTRAN will find the command format is identical for COBOL. However, different switches (options) are used with COBOL.

It should be noted that COBOL consists of a main segment and four overlays which are read in consecutively during compilation. These overlays are always read from the currently logged in disk. Consequently one should always run COBOL from the currently logged in disk.

1.1.1 Format of Commands

COBOL is invoked by typing COBOL followed by a space, followed by an appropriate command string, as described below. COBOL is read from the disk and then examines the command string. If it is valid, compilation commences. If not, COBOL responds with "?COMMAND ERROR" followed by an asterisk so the user can try again. When finished, COBOL always exits to CDOS.

The general format of a COBOL compiler command is:

```
objprog-dev:filename.ext, list-dev:filename.ext=  
source-dev:filename.ext
```

where the various terms mean:

objprog-dev: The device on which the object program is to be written

list-dev: The device on which the program listing is to be written

source-dev: The device from which the source program input to COBOL is taken

NOTE

Whenever a device name is omitted, it defaults to the currently selected disk.

filename.ext

The filename and filename extension of the object program file must be supplied if the device is a directory device. Filename extensions may be omitted, in which case default values are supplied. These values are REL for the object file and LST for the listing file. The source file must have an extension of COB.

Either the object file or the listing file specification or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. The purpose then is only to syntax check for errors which are displayed on the console. If nothing is typed to the left of the equal sign, the object file is written on the same device with the same name as the source file, but with the default extension REL. If only a listing file specification is given, the user may still write out the object file by typing "/R" after the source name. This too writes the object file on the same disk with the same name as the source and the default extension. Similarly "/L" may be used to place the listing file on the same disk with same name as the source and the default extension LST.

Examples

=PAYROLL	Compile the source from PAYROLL.COB placing the object into PAYROLL.REL.
,TTY:=PAYROLL	Compile the source from PAYROLL.COB placing the listing output on the terminal. No object code is generated.
PAYOBJ=PAYROLL.COB	Compile PAYROLL.COB putting the object into PAYROLL.REL.
PAYROLL,PAYROLL=PAYROLL	Compile the PAYROLL.COB putting the object into PAYROLL.REL and listing into PAYROLL.LST.
,=PAYROLL	Compile PAYROLL but produce no object or listing file. Useful for error checking.

1.1.2 COBOL Compilation Switches

A variety of switches may be used in the command string that will affect compilation. Each switch must be preceded by a slash (/).

<u>Switch</u>	<u>Action</u>
R	Force Generation of an object file as described above.
L	Force generation of a listing file as described above.
P	Each/P allocates an extra 100 bytes of stack space for use during compilation. Use /P if stack overflow errors occur during compilation. Otherwise this switch is not needed.

1.2 Output Listings and Error Messages

The listing file output by COBOL is a line-by-line account of the source file with error messages, some interspersed throughout the listing, some generated only at the end. Each source line listed is preceded by a consecutive 4-digit decimal number. This is used by the error messages at the end to refer back to lines in error, and also by the Runtime System to indicate what statement has caused a Runtime Error after it occurs.

Two classes of diagnostic error messages may be produced during compilation.

Low level flags are displayed directly below source lines on the listing when simple syntax violations occur. Remedial action is assumed in each case, as documented in the following table, and compilation continues.

CROMEMCO COBOL USER'S GUIDE
 Compiling COBOL Programs

<u>Flag</u>	<u>Reason for Flag</u>	<u>Continuation Action</u>
"QLIT"?	Faulty quoted literal	
	1. Zero length	Ignore and continue.
	2. Improper continuation	Assume acceptable.
	3. Premature end-of-file (before ending delimiter)	Assume program end.
LENGTH?	Quoted literal length over 120 characters, or numeric literal over 18 digits, or 'word' (identifier, or name) over 30 characters.	Excess characters are ignored.
CHRCTR?	Illegal character	Ignore and continue.
PUNCT?	Improper punctuation (e.g. comma not followed by a space).	Assumes acceptable.
BADWORD	Current word is malformed such as ending in hyphen, or multiple decimal points in a numeric literal.	Ignore and continue.
SEQ #	Improper sequence number (includes case of out-of-order sequence number).	Accept and continue.
NAME?	Name does not begin with a letter (A -Z).	Accept and continue.
PIC = X	An improper Picture.	PIC X is assumed.
COL.7?	An improper character appears in source line character 'column' 7, where only * -/ D are permissible.	Assumes a blank in column 7.
AREA A?	Area A, columns 8-12, is not blank in a continuation line.	Ignore contents of Area A (assumes blank).

High level diagnostic messages consist of two or three parts:

1. The associated source line number -- four digits, followed by a colon (:).
2. An english explanation of the error detected by the compiler. If this text begins with /W/, then it is only a warning; if not, it is an error sufficiently severe to inhibit assembly, linkage, and execution of an object program.
3. (Optional) The program element cited at the point of error is listed.

Design of the high level diagnostic message text is such that no list of 'messages and error codes' is necessary. The messages are designed to be self-explanatory.

1.3 Files Used by COBOL

In addition to the Source, Listing and Object files used by COBOL, two other files should be noted.

First, there is a file called STEXT.INT which the compiler always places on the primary disk. It is used to hold intermediate symbolic text between pass one and pass two of the compiler. It is created, written, then closed, read, and then deleted before the compiler exits. Consequently, it should be transparent to the user unless the compilation is aborted.

Another file of concern to the user is the file to be copied due to a COPY verb in the COBOL program. The user gives the name of the source file to be read and compiled in place of the COPY statement. Remember that copied files cannot have COPY statements within them and the rest of the line after a COPY statement is ignored.

SECTION 2

Linking and Loading COBOL programs

2.1 Command Format

The CROMEMCO Linker/Loader is used to link assembled program modules together, load them into memory, and begin execution there if desired. The Linker is supplied to the user on diskette (large or small) under the directory entry "LINK.COM". The command line to call LINK consists of a number of filenames and switches according to the following format:

```
LINK <d:filenam1.ext/s,d:filenam2.ext/s,...>
```

where d stands for the disk drive letter (A through D), s stands for one of the legal switches of the Linker (see list in this chapter), and filename.ext stands for a user filename plus its 3-letter extension. The only argument required after the word LINK is filenam1. LINK defaults to the current drive if the disk drive letter is omitted, and it defaults to the extension .REL if the 3-letter extension is omitted. The switches are not required. If used, they give LINK instructions regarding the files. The Linker will accept commands in the order received, but does not require a single command line. The prompt for LINK is an asterisk (*). Any time the asterisk appears, a command may be entered. Thus, the names of files to be linked may be given one at a time rather than on one command line. After each command line is entered, LINK will load or search the named file(s). When LINK finishes this process, it will list all symbols that remain undefined followed by an asterisk.

The switches LINK accepts give the user a variety of ways to control the linking process. For example the user may cause the Linker to search special library files to satisfy undefined globals by linking the filename to be searched followed by /S. The /M switch can be used to map a list of all defined and undefined symbols. These switches are described in the next section.

2.2 LINK Switches

The Linker allows a number of switches which specify actions affecting the loading process. Each switch must be preceded by a slash (/). These switches and a description of each follow.

2.2.1 E (Exit to CDOS)

Exit to CDOS upon completion of link and load. Prior to exiting, LINK prints on the console the start and stop execution addresses along with the number of 256-byte pages of memory the program occupies (in decimal), according to the following format:

[xxxx yyyy zz]

where xxxx is the address at which execution will start, yyyy is one more than the highest location used by the loaded object code, and zz is the decimal number of pages required.

The file which has been linked and loaded into memory may be saved after exiting to CDOS. This can be done using the SAVE command, which is one of the CDOS intrinsic commands (see the Cromemco CDOS manual). The user would type:

SAVE filename.ext zz

where zz is the same number printed out by LINK (following the issue of /E). The filename can be any legal name. If the name used already resides on the disk, the saved file will be written over this existing file. The 3-letter extension COM is frequently used; this will create a COMMAND file. Note that other CDOS INTRINSIC commands may be given before the SAVE command; for example, DIR may be typed to see how much directory space is available. Executing any utility program (XFER, EDIT, etc.) will change the contents of the user-area.

2.2.2 G (Go - start execution)

Start execution of the program as soon as the current command line has been interpreted. Prior to execution, LINK prints on the console the start and stop addresses and the number of 256-byte pages occupied by the object code, according to the format shown above (see /E). Following this, the message "[BEGIN EXECUTION]" is displayed on the console at which point execution is started by LINK. The Linker initializes the stack pointer at the highest address of the user-area.

2.2.3 M (Map all symbols)

List both all the defined globals and their values and all undefined globals followed by an asterisk. The map may be sent to the printer by typing Control-P (^P) following the LINK command line. This printed map of symbols is very useful for debugging the user-program. Once the object code has been loaded into memory by LINK, /E can be issued and the correct portion of the user-area saved in a file. Then the program DEBUG can be called and used to load and debug the file just created. The global map printed previously can be used to reference addresses.

2.2.4 R (Reset Linker)

Put Loader back in its initial state. /R is used to restart LINK if the wrong file was loaded by mistake. /R will take effect as soon as it is encountered in a command string.

2.2.5 S (Search file)

Search the disk file having the filename immediately preceding the /S to satisfy any undefined globals. This is convenient for having the Linker search a library file of much-used routines.

2.2.6 U (list all Undefined globals)

List all undefined globals as soon as the current command line has been interpreted and executed. LINK defaults to this switch; therefore, it is generally not needed unless it is desired to reproduce this list more than once. For example say that during link the list of undefined globals is printed to the console. The user could then type Control-P followed by "/U" to cause the undefined globals to be listed a second time, this time to the printer as well as the console.

2.3 Format of Link-Compatible Object Files

The following is a description of the format of .REL files which are to be compatible with the CROMEMCO Linker. This information is provided for the interested programmer, but is not in any way required reading for the person learning how to use the Linker.

LINK compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries except as noted below. The use of a bit stream for relocatable object files keeps the size of the files to a minimum, thereby decreasing the number of disk reads and writes. The first bit of a field is either a one or a zero, and this is followed either by an 8-bit byte or a 2-bit field having the following meanings:

Bit	Meaning
0	load the following eight-bit byte as absolute code
1	read in the following two bit field:
11	Add sixteen-bit offset to common base
10	Add sixteen-bit offset to data base
01	Add sixteen-bit offset to program base
00	Special LINK item

Special LINK item fields begin with the bit stream 100 as just explained. This is followed by a four-bit control field, an optional A-field which consists of a two-bit code specifying address type,

and an optional B-field which consists of 3 bits giving a symbol length. The 2-bit address type has the same meanings as the 2-bit field above except 00 specifies absolute addressing. The 3-bit symbol length is followed by eight bits for each character of the symbol. We can represent this bit stream by the following:

	A-field	B-field
1	00 xxxx <yy two-byte-value>	<zzz characters-of-symbol-name>

where the spaces in the above show where the various fields end, the angular brackets denote optional quantities, and where:

xxxx	is the four-bit control field,
yy	is the two-bit address type field, and
zzz	is the three-bit symbol length field.

The two-byte-value following yy will be either the 16-bit offset specified or the absolute address, and the characters-of-symbol-name following zzz will be in ASCII, each character occupying eight bits.

The four-bit control field will specify the operation or function of the bit stream. It can have the following values, where the four-bit value is given in the left-hand column in decimal:

(The following LINK items have a B-field only)

0	Entry Symbol (name for search).
1	Select COMMON Block.
2	Program Name.
3	Reserved for Future Expansion.
4	Reserved for Future Expansion.

(The following LINK items have both an A-field and a B-field)

5	Define COMMON Size.
6	Chain External (A is head of address chain). B is name of external symbol.
7	Define Entry Point.
8	Reserved for Future Expansion.
9	Reserved for Future Expansion.

(The following LINK items have an A-field only)

10	Define Size of Program Data Area.
----	-----------------------------------

11 Set Loading Location.
12 Chain Address.
 A is head of chain; replace all entries in
 current location counter. The last entry
 has an address field of absolute zero.
13 Define Program Size.
14 End Program (forces to byte boundary).

(The following LINK item has neither an A- nor a B-field)

15 End of File.

2.4 Link Error Messages

The Linker gives several error messages in case of an illegal operation. These are listed below in the summary along with an explanation of each one. Note that there are two types of error messages: fatal errors and warnings. Fatal error messages are preceded by question marks (?) and warning messages are preceded by percent signs (%). A program will run in some cases when a warning has been issued; however, it is better practice to correct the error and link again.

2.4.1 Fatal Errors

?No Start Address	A /G switch is issued, but no main program module has been loaded. Remember when creating and linking machine language programs that the main module must have an address or label in its END statement. This then becomes part of the .REL file which informs LINK where to begin execution (see the END pseudo-op in the Cromemco Assembler Manual).
?Loading Error	The last file given to be linked and loaded is not a properly formatted LINK object file.

?Fatal Table Collision	There is not enough memory to load the given program(s).
?Command Error	An unrecognizable LINK command has been given. Type the correct command or re-link.
?File Not Found	A file in the command string does not exist as spelled or specified. Check to see if the file resides on the specified drive. Often this message results if the user forgets to specify the drive letter, and LINK searches the current drive.

2.4.2 Warnings

%2nd COMMON Larger /XXXXXX/	The first definition of COMMON block XXXXXX is not the largest. COMMONs do not have to be the same size provided the module containing the larger COMMON specification is linked first so that LINK allocates an appropriate number of bytes for data storage. To prevent this error re-order the module loading sequence or change the COMMON block definitions.
%Mult. Def. Global YYYYYY	More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

2.5 Examples of Linking Modules

Following are several examples of the process of linking, loading, saving, and executing files. The asterisk (*) in the following command lines is NOT user-typed; it is the prompt for LINK.

The following command will load a 32-byte program called MYPROG into memory and begin execution:

LINK MYPROG/G

If the load is successful (no errors), the Linker will respond with the message:

```
[1000 1020 16]
[BEGIN EXECUTION]
```

This program will begin execution at 1000H. If desired, the program can be saved prior to execution by typing:

LINK MYPROG/E

to which the Linker would respond:

```
[1000 1020 16]
```

followed by a return to CDOS and the issue of the CDOS prompt. This return to CDOS does not change the user area. Therefore, the program could be saved by issuing the command:

SAVE MYPROG.COM 16

Since the extension COM was chosen, the program can be executed directly from CDOS by typing the command "MYPROG" in response to the CDOS prompt.

Another example would be to link several modules together as they are loaded into memory. Suppose we have the three relocatable modules GRAPHX, MAIN, and SUBPLOT. The Linker is called first by issuing the command:

LINK <CR>

to which LINK responds with the asterisk. We could then type:

MAIN

The Linker would look on the current drive for MAIN and then return the still-undefined symbols (each one followed by an asterisk) and the address at which they are referenced:

```
INITG* 122E
LINE* 164D
CURSR* 163E
STRIN* 131B
```

CROMEMCO COBOL USER'S GUIDE
Linking and Loading COBOL programs

```
SUBROT*147D  
*
```

We then link the next module:

```
GRAPHX
```

and LINK again responds with the undefined symbols and the prompt:

```
  SUBROT* 147D  
*
```

Finally, we LINK the last module:

```
SUBROT
```

to which LINK responds with the prompt. We can now type /G or /E to run or exit from the program as we did in the first example. However, let's first generate a map of all the symbols using the /M LINK switch:

```
*/M
```

to which the Linker will respond:

```
INITG    122E  
LINE     164D  
CURSR    163E  
STRIN    131B  
SUBROT   147D  
PAGE     17DF  
DOT      180E  
ANIMT    1558
```

Note that this is similar to the map of undefined symbols; however, in this case symbols which are not used, but have been defined in one of the linked programs, are also listed.

The above example could also have been linked directly, and without producing the maps of undefined symbols, by typing the command line:

```
LINK GRAPHX,SUBPLOT,MAIN/M
```

Note also that this command line links them in a different order than the first case since all of the modules are relocatable. Thus, the map printed to the console this time would have a different address

CROMEMCO COBOL USER'S GUIDE
Linking and Loading COBOL programs

after each symbol.

The user may return to CDOS at any time while using LINK (to abort the linking or loading process, for example) by typing Control-C (^C).

SECTION 3

Runtime Execution

3.1 Printer File Handling

Printer files should be viewed as a stream of characters going to the printer. Records should be defined as the fields to appear on the printer. No extra characters are needed in the record for carriage control characters. Carriage return, line feed and form feed are sent to the printer as needed between lines. Note that blank characters (spaces) on the end of a print line are truncated to make printing faster.

No "VALUE OF" clause should be given for a PRINTER file in the FD, but "LABEL RECORD IS OMITTED" must be specified. The BLOCK clause must not be used for printer files.

3.2 Disk File Handling

Disk files must have "LABEL RECORD IS STANDARD" declared and have a "VALUE OF" clause that includes a File ID. Block clauses are checked for syntax but have no effect on any type file at this time.

The format of sequential files is always that of variable length strings delimited by a carriage return/line feed. Records are packed together as much as possible to make maximum use of floppy disks.

The format of relative files is always that of fixed length records of the size of the largest record defined for the file. No delimiter is needed, and therefore none is provided. Deleted records are filled with hex value '00'. Also note that 6 bytes are reserved at the beginning of the file to contain bookkeeping information for COBOL.

The format of indexed files is too complicated to include in this document. It is a complex mixture of keys, data, linear pointers, deletion pointers, and scramble+function pointers. It is doubtful that

the COBOL programmer would require access to such information.

3.3 Runtime Errors

Runtime terminal errors result in a four-line synopsis, printed on the console.

```
** RUN-TIME ERR:  
reason (see list below)  
line number  
program-id
```

The possible reasons for termination, with additional explanation, are listed below.

REDUNDANT OPEN	Attempt to open a file that is already open.
DATA UNAVAILABLE	A file's base register contains a non-zero address if, and only if, the file is open and available record areas exist. Reference to data in a record of a non-open file, or one that has already reached the "AT END" condition, is invalid, and is detected by recognizing zero in the associated base register.
SUBSCRIPT FAULT	A subscript has an illegal value (usually, less than 1). This applies to an index reference such as I + 2, the value of which must not be less than 1.
INPUT/OUTPUT	Unrecoverable I-O error, with no provision in the user's COBOL program for acting upon the situation by way of an AT END clause, INVALID KEY clause, DECLARATIVE procedure, etc.

NON-NUMERIC DATA	Whenever the contents of a numeric item does not conform to the given PICTURE, this condition may arise. Corresponds to the hardware 'data exception' interrupt in some computers. The user should always check input data, if it is subject to error (because "input editing" has not yet been done) by use of the NUMERIC test.
PERFORM OVERLAP	An illegal sequence of PERFORM's as, for example, when paragraph A is performed, and prior to exiting from it another PERFORM A is initiated.
CALL PARAMETERS	There is a disparity between the number of parameters in calling program and called subprogram.
ILLEGAL READ	Attempt to READ a file that is not open in the input or I-O mode.
ILLEGAL WRITE	Attempt to WRITE to a file that is not open in the output mode for sequential access files, or in the output or I-O mode for random or dynamic access files.
ILLEGAL REWRITE	Attempt to REWRITE a record in a file not open in the I-O mode.
REWRITE; NO READ	Attempt to REWRITE a record of sequential access file when the last operation was not a successful READ.
REDUNDANT CLOSE	Attempt to close a file that is not open.
GO TO. (NOT SET)	Attempt to execute an uninitialized alterable paragraph containing only a null GO statement.

CROMEMCO COBOL USER'S GUIDE
Runtime Execution

FILE LOCKED	Attempt to open after earlier CLOSE WITH LOCK.
READ BEYOND EOF	Attempt to read (next) after already encountering end-of-file.
DELETE; NO READ	Attempt to DELETE a record of a sequential access file when the last operation was not a successful READ.
ILLEGAL DELETE	Relative file not opened for I-O.
ILLEGAL START	File not opened for input or I-O.

I N D E X

A

ACCEPT statement, 15, 66, 79
ACCESS clause, 26, 94, 101
ADD statement, 60
ADVANCING option, 76
ALL phrase, 81
Alphanumeric item, 28, 31, 34
Alphanumeric-edited item, 34
ALTER statement, 69
ANSI level 1, 6
ANSI level 2, 6
Arithmetic expression, 63
Arithmetic statements, 58
ASCII-ENTRY, 25
AT END clause, 51, 74, 96, 97
AUTHOR, 23

B

Binary item, 29, 33
BLANK WHEN ZERO clause, 44
BLOCK clause, 47

C

CALL statement, 86
Character comparisons, 72
Character set, 8
Class test condition, 72
CLOSE statement, 77
CODE-SET clause, 48
Comments, 21
Compound condition, 70
COMPUTATIONAL, 30, 33
COMPUTATIONAL-3, 29, 33
COMPUTE statement, 63
Conditions, 9, 11, 15, 49, 51, 55, 70, 73
CONFIGURATION SECTION, 24
Continuation line, 16, 21
Control index, 94
COPY statement, 22
COUNT IN phrase, 82
CURRENCY SIGN, 24

D

Data description entry, 30, 48
Data Division, 14, 18
Data Item, 13, 28
DATA RECORDS clause, 47

Data-name, 10, 12, 14, 30
DATE-COMPILED, 23
DATE-WRITTEN, 23
Debugging, 7, 24, 83
Decimal item, 32, 45
Decimal point, 36
DECIMAL-POINT IS COMMA, 17, 24
DECLARATIVES, 53, 106
DELETE statement, 99, 104
DELIMITED BY phrase, 80
DISPLAY statement, 15, 66
DIVIDE statement, 62

E

Elementary item, 12, 28, 31, 33
Ellipsis, 12
Environment Division, 15, 18, 24
EXHIBIT statement, 84
EXIT PROGRAM statement, 86
EXIT statement, 68
EXTEND phrase, 74
External decimal item, 29

F

FD entry, 14, 21, 46
Figurative constants, 17
File, 12
File name, 14
File Section, 14, 46
FILE STATUS clause, 26, 95
FILE STATUS data item, 74
FILE-CONTROL, 26
File-name, 10
FILLER, 30
Floating string, 37
Format notation, 10

G

General Formats, 10
GIVING option, 60
GO TO statement, 65
Group, 33
Group item, 13, 28, 31, 43, 53

H

HIGH-VALUE, 17

I

I-O, 74

I-O error handling, 78
I-O-CONTROL paragraph, 26, 27
Identification Division, 18, 23
IF statement, 69
Imperative statements, 51, 58
Index data-item, 30, 33, 87
Index-name, 87
Indexed I-O, 7
Indexed-file organization, 94
INPUT file, 73
INPUT-OUTPUT SECTION, 25
INSPECT statement, 55
INSTALLATION, 23
Inter-Program Communication, 7
Internal decimal item, 29
INTO option, 75
INVALID KEY clause, 51, 96, 98, 99, 100

J
JUSTIFIED RIGHT clause, 44

K
KEY clause, 90
KEY IS clause, 98

L
LABEL clause, 46
Level 88, 49
Level number, 12, 21, 28, 30, 49
Library, 7
Linkage section, 49
Literals, 15
LOCK suffix, 77
LOW-VALUE, 17

M
Mnemonic-name, 10, 15
Modules, 6
MOVE statement, 53
MULTIPLY statement, 62

N
Non-numeric literals, 15
Nucleus, 6
Numeric Comparisons, 72
Numeric item, 28, 34
Numeric literals, 16

O

OBJECT-COMPUTER, 24
OCCURS clause, 42
OMITTED, 46
ON OVERFLOW clause, 82
OPEN statement, 73
ORGANIZATION clause, 26
OUTPUT file, 73
OVERFLOW, 51

P

Packed decimal, 29
Paragraphs, 52
Parentheses, 12
PERFORM statement, 67
PICTURE, 29
PICTURE clause, 34
POINTER phrase, 80
PRINTER, 15, 46, 47
Procedure Division, 18, 51
Procedure-name, 10, 21, 52
PROGRAM-ID, 23
Punctuation, 8, 9, 10

Q

Qualification, 22
QUOTE, 17

R

Range (PERFORM), 68
READ statement, 74, 97, 103
READY TRACE statement, 84
Records 12, 48, 95
REDEFINES clause, 41
Relative I-O, 7
Relative indexing, 88
RELATIVE KEY clause, 102
RELATIVE KEY item, 103
Relative organization, 101
REPLACING clause, 56
Report item, 28, 32, 35
RESERVE clause, 26
Reserved words, 10, 21
RESET TRACE statement, 83
REWRITE statement, 77, 99, 104
ROUNDED option, 59

S

SAME AREA, 27

SEARCH ALL statement, 89
SEARCH statement, 90
Section-name, 52
Sections, 52
SECURITY, 23
SELECT entry, 26, 94, 101, 102
Sentences, 51, 52
Separator, 9
Sequence number, 21
Sequential I-O, 7
SET statement, 87
SIGN clause, 29, 45
Sign test, 73
Simple condition, 70
SIZE ERROR option, 51, 59
SOURCE-COMPUTER, 24
SPACE, 17
SPECIAL-NAMES, 24
STANDARD, 46
START statement, 100, 105
Statements, 51
STOP statement, 65
STRING statement, 80
Subscripts, 43, 50
SUBTRACT statement, 61
SYNCHRONIZED clause, 44

T

Table Handling, 7
TALLYING clause, 56
TRACE mode, 83

U

UNSTRING statement, 81
USAGE clause, 33
USE sentence, 106
USING list, 49, 85

V

VALUE IS clause, 40, 49
VALUE OF clause, 46
VARYING, 89
Verbs, 51

W

WHEN clause, 90
Word, 8, 10
Working-storage section, 49
WRITE statement, 75, 98, 103

ERRATA

Page : 11, item "6", line 2

Incorrect : ...in brackets is...

Correct : ...in brackets ([]) is...

Page : 16, item "2.", line 1

Incorrect : ...Area B preceding...

Correct : ...Area B (see 1.12, Coding Rules) preceding...

Page : 19, under ENVIRONMENT DIVISION

Incorrect : [CONFIGURATION SECTION.]

Correct : [CONFIGURATION SECTION.]

Page : 21, item "3.", line 2

Incorrect : ...columns 12-72...

Correct : ...Area B (columns 12-72)...

Page : 67, PERFORM, option 2

Incorrect : PERFORM range VARYING {index name} {data name} FROM
amount-1 BY amount-2 UNTIL condition.

Correct : PERFORM range [VARYING {index name} {data name}]
FROM amount-1 BY amount-2] UNTIL
condition.

Page : 68, EXIT statement, add last paragraph

Add : The paragraph containing the EXIT statement must be the final paragraph in a PERFORM'ed section in order to function as an end-point from which control returns to the statement following the PERFORM.

Page : 73, OPEN STATEMENT

Incorrect : { OPEN { INPUT }
{ I-O } file-name ... } ...
{ OUTPUT }
{ EXTEND }

Correct : OPEN { { INPUT }
{ I-O } file-name ... } ...
{ OUTPUT }
{ EXTEND }

Page : 75, WRITE STATEMENT

Incorrect : WRITE record-name FROM data-name-1
{AFTER} ADVANCING { operand LINE (S) }
{BEFORE} { PAGE }

Correct : WRITE record-name [FROM data-name-1]
[{ AFTER } ADVANCING { operand LINE (S) }]
[{ BEFORE } { PAGE }]

CROMEMCO COBOL MANUAL SUPPLEMENT

Page : 83, first paragraph, add last sentence
Add : Then the next receiving field (if any) in the list becomes the current receiving field and scanning continues.

Page : 86, CALL STATEMENT
Incorrect : CALL literal USING data-name ...
Correct : CALL literal [USING data-name ...]

Page : 95, RECORD KEY CLAUSE
Incorrect : ...required, is...
Correct : ...required in the SELECT entry for the file, is...

Page : 95, RECORD KEY CLAUSE
Incorrect : ...group item, an elementary alphanumeric item or a decimal field. A decimal key...sign. No record key...
Correct : ...group item or an elementary alphanumeric item. No record key...

Page : 108, example 1
Incorrect : ...OR (E < F) and...
Correct : ...OR (E <= F) and...

SUPPLEMENTAL INFORMATION - REFERENCE MANUAL

COPY STATEMENT

This statement must begin in column 8, and the name of the file containing the source module is not enclosed in either single or double quotes.

The file referenced in the COPY statement must contain a single module of source statements. These statements will be inserted without change in place of the COPY statement.

There is no COPY library as such--each module of library code must reside in a uniquely named file on disk.

Calling assembler and FORTRAN subprograms

An assembler or FORTRAN subprogram may be called from a COBOL program by using its name in the CALL statement, then linking the relocatable object code modules together using LINK.

For example, suppose the assembler routine SCAN is called from the COBOL program FINDTEXT. The COBOL program is compiled and the assembler routine is assembled, then the resulting relocatable object code modules are linked by LINK:

A. LINK FINDTEXT, SCAN, FINDTEXT/N/E

Parameters are always passed by reference (i.e., the value passed is the two-byte address of the first byte of the actual argument).

Three or fewer parameters are passed this way:

Parameter 1 address in HL
Parameter 2 address in DE
Parameter 3 address in BC

Four or more parameters are passed this way:

Parameter 1 address in HL
Parameter 2 address in DE
Parameter 3 - n addresses are in one block of memory BC points to the first byte of the block

CROMEMCO COBOL MANUAL SUPPLEMENT

To return from a FORTRAN subprogram to a calling COBOL program, use the RETURN statement.

To return from an assembler subprogram to a calling COBOL program, use the RET command.

Note that the only type of data that is stored in the same format in COBOL and FORTRAN is alphanumeric data. None of the numeric types of data are interchangeable.

SUPPLEMENTAL INFORMATION - USER'S GUIDE

Compiling COBOL Programs

COBOL Overlay File Usage

OVRLY1 These files are overlay files brought into
OVRLY2 memory by COBOL during the compilation. They
OVRLY3 must be on the currently logged-in disk.
OVRLY4

"Dictionary Full" compile time error message

This message indicates that the symbol table used by the compiler is full and that the compiler cannot proceed with the compile. Restart the compilation, using the /P compiler switch to allocate more memory to the compiler stack.

COBOL Memory Usage Approximation

8K CDOS (version 1.07)
31K COBOL compiler
7K minimum table space

20 bytes for each line of COBOL source code

The sum of these quantities is the approximate memory required to compile a COBOL program.

Linking and Loading COBOL Programs

LINK Switches

N Specifying <filename>/N will cause the linked program to be saved on disk when a /E or a /G is entered. <filename> will be the name assigned to the program, with a default extension of .COM when no extension is specified. A jump to the start of the program is inserted if the program's program area is loaded starting at a location other than 100 hex.

P and D /P and /D allow the program and data origins to be set for the next program loaded. /P and /D take effect when entered and they have no effect on program modules already loaded. The form is /P:<address> and /D:<address>, where <address> is the desired origin in the current typeout radix. (The default radix is hex. /O sets the radix to octal; /H sets the radix to hex.) LINK defaults to /P:<link origin> + 3 (103 hex) to leave room for the jump to the start of the program's program area.

Do not use /P or /D to load programs or data into locations 100 hex to 102 hex, which is where LINK inserts the jump to the start of the program's program area, unless the start of the program area is loaded there. LINK will not generate and insert the jump instruction if these locations are used.

If no /D is entered, data areas are loaded before program areas for each module. If a /D is entered, all data and Common areas are loaded starting at the data origin and the program area at the program origin.

Example:

```

*/P:200,EXAMPL
Data      200      300
*/R
*/P:200 /D:400,EXAMPL
Data      400      480
Program  200      280

```

Format of LINK Compatible Object Files

Loader type 9 is now in use; it is external + offset. Type 9 has only an A field, there is no B field. The value for type 9 will be added to the two bytes starting at the current location counter. This addition is done after a /E or /G is entered, so unless undefined symbols remain, the effect is external + offset.

This type can also be used to add program and data relatives or almost any other combination of relocation types.

LINK Error Messages

?Out of Memory has replaced ?Fatal Table Collision

?<file> Not Found has replaced ?File Not Found. The name of the file not found is printed.

%Overlaying [Program] Area
[Data]

A /D or /P will cause previously loaded data or program to be destroyed.

?Intersecting [Program] Area
[Data]

The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

?Start Symbol -<name> -Undefined

After a /E: or /G: is entered, the symbol specified was not defined.

Origin [Above] Loader Memory, Move Anyway (Y or N)?
[Below]

After a /E or /G was entered, either the data or program area has an origin or top which lies outside loader memory (i.e., loader origin to top of memory). If a Y <cr> is entered, LINK will move the area and continue. If anything else is entered, LINK will exit. In either case, if a /N had already been entered, the program image will have already been saved on disk.

?Can't Save Object File

Available disk was exhausted, or a disk error occurred when LINK was attempting to save the program.

Separating Data and Program Areas

By default, LINK loads the data area first in memory, followed immediately by the program area. The user may use switches /D and /P to set the beginning addresses of the data and program areas, with the following considerations.

When linking FORTRAN subprograms, one should be aware that the disk driver module in the FORLIB relocatable library file has declared an ENTRY global symbol called \$MEMORY in order to obtain from LINK a pointer to the first available byte following a program's data area. When LINK finds this symbol in the module being loaded, it loads into it the address of the top of the data area + 1.

In the default case, where neither /D nor /P has been used, the program area is considered part of the data area, and LINK loads the two-part data area (data and program), then sets \$MEMORY to point to the top of the two-part data area + 1.

When either /D or /P has been used to separate the two areas, \$MEMORY will be set to point to the top of the data area + 1.

Thus, when /D is used to load the data area below the program area, the user must be certain that there is enough memory between the top of the data and the bottom of the program area. Otherwise the program area will be overwritten and the program destroyed at runtime.

The FORTRAN disk driver module uses \$MEMORY to allocate disk buffers (128 decimal bytes each) and FCB's (33 decimal bytes each). These follow the address pointed to by \$MEMORY (top of data area + 1), so the user must have an idea of their total length in order to start the program area above their ending address.

Part No. 023-0049



Cromemco
i n c o r p o r a t e d
Tomorrow's Computers Today
260 BERNARDO AVE. MOUNTAIN VIEW, CA 94043