microsoft

cobol-80 documentation

Microsoft COBOL-80 and associated software are accompanied by the following documents:

1. COBOL-80 REFERENCE MANUAL
   provides extensive descriptions of COBOL-80's statements, syntax and organization.

2. COBOL-80 USER'S MANUAL
   describes the COBOL-80 compiler commands, file handling and error messages.

3. MICROSOFT UTILITY SOFTWARE MANUAL
   describes the use of the MACRO-80 Assembler, LINK-80 Linking Loader and LIB-80 Library Manager with the COBOL-80 compiler.

COBOL-80

Overview

Microsoft's COBOL-80, which runs on the 8080/Z-80/8085, brings the world's most widely used computer programming language to the microcomputer user. COBOL-80 is comparable to COBOL systems found on minicomputers and large mainframes. Consequently, it greatly enhances the usefulness of microcomputers because it gives users access to the incredibly large number of programs already written in COBOL. Because COBOL-80 is a standard, COBOL programs written on other computers may be run easily on 8080, Z-80 or 8085 systems.

Microsoft's COBOL is based on the 1974 ANSI standard and contains all Level 1 features and the most useful Level 2 options for the "Nucleus" and for Sequential, Relative and Indexed file handling facilities. Additionally, Level 1 Table Handling, Library and Interprogram Communication facilities are provided. Of the advanced Level 2 features, Microsoft has included the verbs STRING, UNSTRING, COMPUTE, SEARCH, and PERFORM (varying/until), along with convenient condition specification by way of condition-names, compound conditions and abbreviated conditions. Furthermore, a data format called COMP-3 allows numeric data to be packed two digits to the byte so that mass storage requirements are reduced. Lastly, a batch-style Debug technique is implemented to get programs running in a minimum of on-line time.

Microsoft's COBOL system consists of two complete packages: a compiler for translating source code into relocatable object code (which, incidentally, is compatible with the object code of our FORTRAN-80 compiler and MACRO-80 assembler), and a runtime system for running the program by interpreting the object code at execution time.

The Compiler

The compiler is written in carefully designed, machine-independent pseudo-code rather than 8080 machine language. There is an interpreter written in 8080 machine language which executes the algorithms specified by the pseudo-code. Because this pseudo-code has been implemented successfully on several minicomputers by simply rewriting the interpreters, the final product is always more reliable, less costly and requires less memory than a purely machine-coded version. This technique simplifies maintenance because the compiler is more compact and the pseudo-code instructions were specifically chosen with the intent of writing a COBOL compiler. Speed degredation is less than 20% because the time spent to interpret each pseudo-code instruction is insignificant compared to the time required for its total function.

The size of Microsoft's compiler plus interpreter is approximately 25K bytes exclusive of operating system and table space. It consists of five overlays, each one executed in turn, in the same memory space. The operating system for compilation needs only the capability to read and write sequential files. Because the compiler is "two-pass," the source code is read while an intermediate file is written; then the intermediate file is read and the generated object code is written. An optional file showing source lines and errors may also be produced. Of course, if the COPY function is used, provision must be made for an alternate source file. Also, an optional table-spill mechanism requires random-access file I/O so that excess table information can be stored temporarily on disk if needed. Lastly, a sequential input file which holds the file overlays is read periodically at the end of each major "phase." An operating system to handle all of this, such as CP/M (along with Microsoft's command scanner), requires about 7K bytes. Table space for a 500 line program is estimated at 12K bytes. Thus the full system requires 25K plus 7K plus 12K or a total of 44K bytes.

Runtime System

The object code generated by the compiler is also interpreted. The runtime system handles all algorithmic functions such as arithmetic, string manipulation and editing in addition to managing sequential, relative and indexed I/O. It also controls program flow as specified at the source level by conditions, GOs and PERFORMs. In short, the runtime system handles, at the machine level, anything that can be written in Microsoft's COBOL.

As in the compiler, the generated pseudo-code specifies the complete algorithmic logic to be executed, and the runtime interpreter manifests the algorithms for a given machine. Again, as with the compiler, the overhead is insignificant when compared to the benefit of memory space reduction. One can simply think of the pseudo-code as subroutine calls to accomplish given tasks with given parameters.

Documentation

Microsoft supplies a COBOL Language Reference Manual and COBOL User's Guide, describing in detail how to write a program, compile it, load it into memory and execute it. Because Microsoft's linking loader is included in the package to load the COBOL object code into memory, information regarding the loader is supplied in the user's guide. The loader format for COBOL is identical to that of Microsoft's FORTRAN compiler and MACRO assembler, so programs of all three languages can be loaded and linked together.

Summary of Features

The following summary specifies the content of Microsoft COBOL with respect to the ANS-74 Standard.

Module	Features Available in Microsoft COBOL
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 VARYING 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 & 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 relationals EQUAL, GREATER, or NOT LESS)
Library	Level 1
Inter-Program Communication	Level 1
Table Handling	All of level 1 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 "WITH DEBUGGING MODE" is given in SOURCE-COMPUTER paragraph.

Contact

COBOL-80 is available to individuals on a single copy, off-the-shelf basis. OEM and dealer agreements are available upon request. For more information, contact:

Ric Weiland
New Products Manager
Microsoft
300 San Mateo, NE, Suite 819
Albuquerque, NM 87108
505-262-1486

Single Copy Pricing

Any purchaser of an off-the-shelf version of COBOL-80 must sign a non-disclosure agreement before COBOL-80 will be shipped by Microsoft. Updates for enhanced versions will be offered for $25 to $100 (depending upon the extent of the enhancements) to single-copy customers. Back-up copies of COBOL may be purchased for $25. COBOL-80 documentation is included with every COBOL-80 system shipped, except back-up copies.

COBOL-80 system (including documentation)	$750.00
COBOL-80 documentation only	$ 20.00

OEM and dealer agreements are available upon request.

Other Products

Microsoft's complete product line includes FOCAL and BASIC for the 6502 and 6800, Altair (8080) BASIC, and FORTRAN for the 8080 and Z-80. In addition, Microsoft has development software that runs on the DEC-10 for all these microprocessors.

MICROSOFT

COBOL-80

reference manual

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 byIBM; 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)

Microsoft
COBOL-80 Reference Manual

CONTENTS PAGE

Introduction

Microsoft 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 Microsoft COBOL with respect to the Standard.

Module	Features Available
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 WITH DEBUGGING MODE is given in SOURCE-COMPUTER 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 his own 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 $\begin{Bmatrix} \text{AREA} \\ \text{AREAS} \end{Bmatrix}$

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.

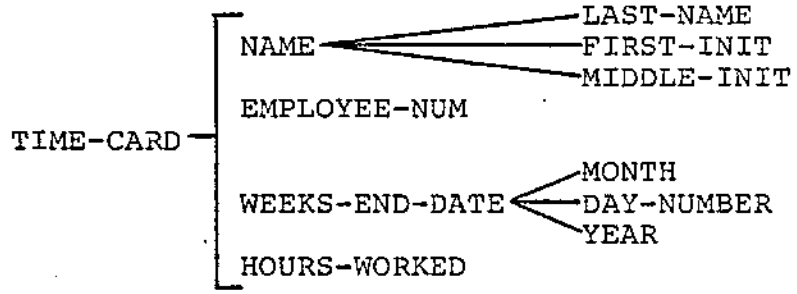
12. Comments, restrictions, and clarification on the use and meaning of every format are contained in the appropriate sections of this manual.

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 heirarchy 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.



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 (01) subordinate to any given file represent implicit redefinitions of the same area whereas in the Working-Storage section, each record (01) 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.

```
01   TIME-CARD.
  02  NAME.
    03 LAST-NAME     PICTURE X(18).
    03 FIRST-INIT    PICTURE X.
    03 MIDDLE-INIT   PICTURE X.
  02  EMPLOYEE-NUM   PICTURE 99999.
  02  WEEKS-END-DATE.
    05  MONTH            PIC 99.
    05  DAY-NUMBER       PIC 99.
    05  YEAR             PIC 99.
  02  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.

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 of a coding sheet, 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 string, which must be non-numeric or a 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 Microsoft 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, such that the punched cards are in ascending order. Blanks are also 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. Frequently, these columns are used to contain the deck 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 card is treated as comments 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-O) 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:

File Status Left	File Status Right	Meaning
'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. 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}  (REDEFINES-clause)  (JUSTIFIED-clause)
             {FILLER   }

  (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}  (REDEFINES-clause)  (USAGE-clause)
             {FILLER   }

   (OCCURS-clause)  (SIGN-clause).
```

Example:

```
01  GROUP-NAME.
    02  FIELD-B PICTURE X.
    02  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)

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

   PICTURE IS an-form  (USAGE IS DISPLAY)  (JUSTIFIED-clause)

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

Examples:

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

REPORT ITEM (also called a numeric-edited item)

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

PICTURE IS report-form  (BLANK WHEN ZERO)  (USAGE IS DISPLAY)

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

Example:

```
     02 XTOTAL PICTURE $999,999.99-.
```

DECIMAL ITEM

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

   PICTURE IS numeric-form  (SIGN-clause)

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

Examples:

```
     02  HOURS-WORKED PICTURE 99V9, USAGE IS DISPLAY.
     02  HOURS-SCHEDULED PIC S99V9, SIGN IS TRAILING.
     - - - - - - - - - - - - - - - - - - -
     11  TAX-RATE PIC S99V999 VALUE 1.375, COMPUTATIONAL-3.
```

BINARY ITEM

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

```
02  SUBSCRIPT COMP, VALUE ZERO.
02  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:

```
                    {COMPUTATIONAL  }
USAGE IS            {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 appreviated 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, 0 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, 0 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 0 - 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.

0 Each appearance of 0 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 examples below 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 (01 group level item), unless s=t=01.

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 01 (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 01 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 01 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

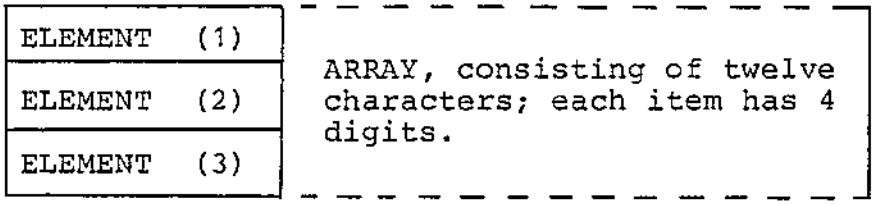
data-name that has an integer value. Whether the subscript is represented by a literal or a data-name, the subscript is enclosed in parentheses and appears after the terminal space of the name of the element. A subscript must be a decimal or binary item. (The latter is strongly recommended, for the sake 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 successivly less inclusive dimensions of the data organization. Multiple subscripts are separated by commas, viz. ITEM (I, J).

Example:

```
01 ARRAY.
03 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 filename

expressed as a COBOL-type "quoted" literal. The general form is:

VALUE OF FILE-ID IS "literal"

Examples:

```
VALUE OF FILE-ID "A:MASTER.ASM"     (CP/M DOS)
VALUE OF FILE-ID IS "D0:X201A.L"    (DTC)
VALUE OF FILE-ID "F0:INVNT.LST"     (Altair)
```

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. See the User's Guide for filename formats for specific disk operating systems.

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

```
VALUE IS  { literal-1 [literal-2...]      }
          { literal-1 THRU literal-2       }
```

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.

```
02 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:

INPECT 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.

	Item to Receive Calculated Result		
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:

```
        {numeric-literal}
ADD     {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:

```
             {data-name-1    }
SUBTRACT     {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      } { BY   } { data-name-2      }
        {numeric-literal-1} {INTO  } {numeric-literal-2}

[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       } [SIZE-ERROR-clause]
{arithmetic-expression}
```

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  } TIMES]
              {data-name}
```

Option 2

```
PERFORM range [VARYING {index-name} FROM
                       {data-name }
    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 / statement(s)-1 } [ELSE statement(s)-2 / 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:

Relation	Meaning
=	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:

```
                                  {NUMERIC   }
data-name IS [NOT]                {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:

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

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 01-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  }           {operand LINE(S)}
{BEFORE } ADVANCING {     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:

Integer	Carriage Control Action
0	No spacing
1	Normal single spacing
2	Double spacing
3	Triple spacing
.	.
.	.
.	.

Single spacing (i.e., "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, as indicated by the FILE STATUS indicator. Otherwise, the FILE STATUS indicator gets a value of '93'. (The FILE STATUS indicator is updated by execution of REWRITE.)

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.

Only if none of the above (INVALID KEY/AT END clause, Declaratives ERROR procedure, or testable FILE STATUS item) exists, then the run-time error handler receives control; the location of the error (source program line number) is noted, and the run is terminated "abnormally."

These remarks apply to processing of any file, whether organization is Sequential, Indexed or Relative.

4.20 ACCEPT DATE/DAY/TIME STATEMENT

The standard date, day or time value may be acquired at execution time by a special form of the

ACCEPT statement:

```
ACCEPT data-name FROM { DATE }
                      { DAY  }
                      { TIME }
```

The formats of standard values DATE, DAY and TIME are:

DATE - a six digit value of the form YYMMDD (year, month, day).
Example: July 4, 1976 is 760704.

DAY - A five digit "Julian date" of the form YYNNN where YY is the two low order digits of year and NNN is the day-in-year number between 1 and 366.

TIME - an eight digit value of the form HHMMSSFF where HH is from 00 to 23, MM is from 00 to 59, 00 is from 0 to 59, and FF is from 00 to 99; HH is the hour, MM is the minutes, SS is the seconds, and FF represents hundredths of a second.

The PICTURE of data-name should be 9(6), 9(5) or 9(8), respectively, for DATE, DAY or TIME acquisition, i.e., all the source values are integers. If not, the standard rules for a move govern storage of the source value in the receiving item (data-name).

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}} ...
                                  {SIZE     }

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 exepected 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

```
                    {literal  }
EXHIBIT NAMED       {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:

```
SET  { index-name-1 }  ...  TO  { index-name-2 }
     { index-item-1 }           { index-item-2 }
     { data-name-1  }           { data-name-2  }
                                { integer-2    }
```

Format 2:

```
SET  { index-name-3 }  ...  { UP BY   }  { index-name-4 }
     { index-item-3 }       { DOWN BY }  { index-item-4 }
     { data-name-3  }                    { data-name-4  }
                                         { integer-4    }
```

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 binary items, except that a decimal item may precede the word TO.

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 on page 84.

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.

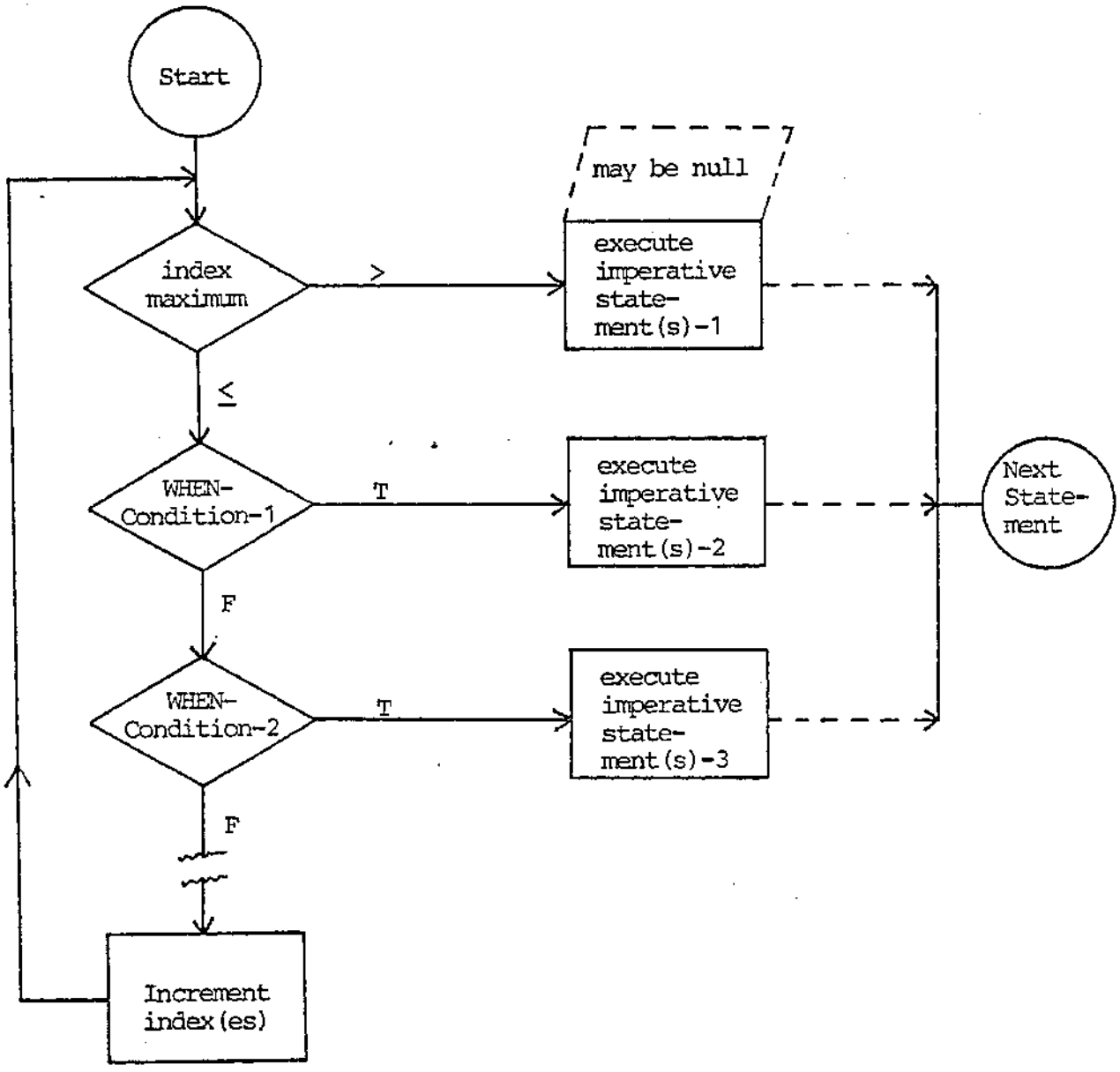
5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or an identifier; the identifier must NOT be referenced in the KEY clause of the table or be indexed by the first index-name associated with the table. (The term identifier means data-name, including any 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.

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 or an unsuccessful READ preceded a DELETE or 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 FILE

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          } data-name]
                        {NOT LESS THAN         }
                        {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:

'1' for a sequential REWRITE or DELETE with no previous successful READ

'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, just 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, REWRITE will return a FILE STATUS code of '21'.

However, 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, DELETE will return a value of '21'. However, an INVALID KEY phrase 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       } data-name-1]
                        {NOT LESS THAN      }
                        {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 = 2) 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

	Receiving Operand in MOVE Statement					
Source Operand	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

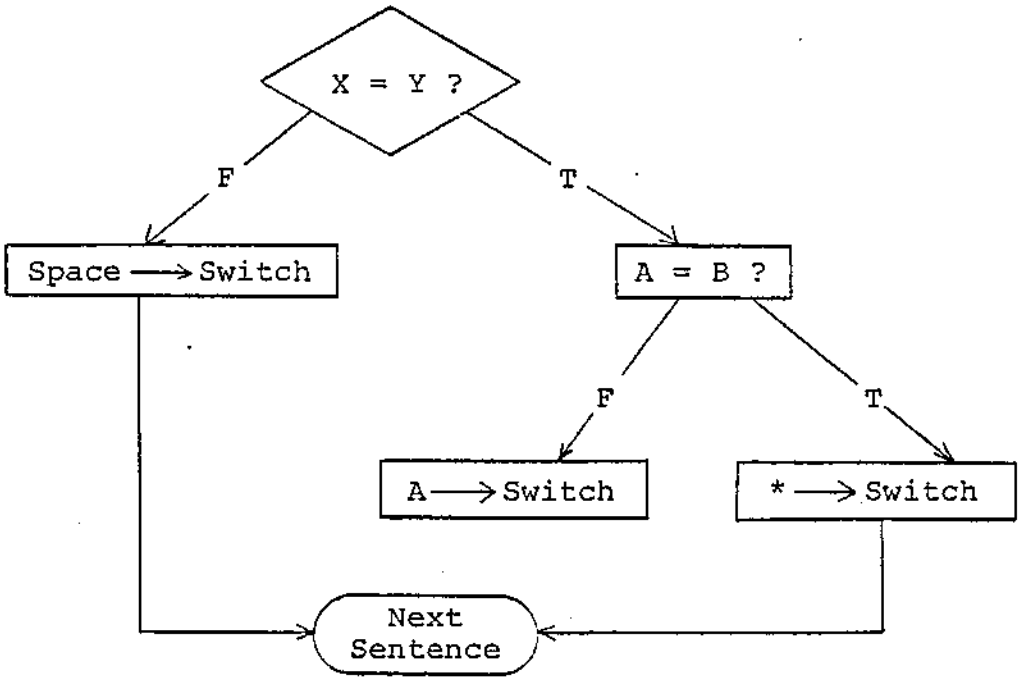
Nesting of IF Statements

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
+-----------+-------------------------------------------+
|           | IF2       A = B                           |
| true      |           true-action : MOVE "*" TO SWITCH |
| action1:  | 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 M1KO 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

Character	Octal Value	Character	Octal Value
A	101	0	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
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
*ALSO
ALTER
*ALTERNATE
AND
ARE
AREA(S)
ASCENDING
**ASCII
ASSIGN
AT
AUTHOR

**BEEP
BEFORE
BLANK
BLOCK
*BOTTOM
BY

CALL
*CANCEL
*CD
*CF
*CH
CHARACTER(S)
*CLOCK-UNITS
CLOSE
*CLOSE
*CODE
CODE-SET
COLLATING
*COLUMN
COMMA
*COMMUNICATION
COMP
COMPUTATIONAL
**COMPUTATIONAL-3
**COMP-3
COMPUTE
CONFIGURATION
**CONSOLE
CONTAINS
*CONTROL(S)
COPY
*CORR(ESPONDING)
COUNT
CURRENCY
DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DEBUGGING
*DEBUG-CONTENTS
*DEBUG-ITEM
*DEBUG-LINE
*DEBUG-NAME
*DEBUG-SUB-1
*DEBUG-SUB-2
*DEBUG-SUB-3
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
*DESTINATION
*DE(TAIL)
*DISABLE
**DISK
DISPLAY
DIVIDE
DIVISION
DOWN
*DUPLICATES
DYNAMIC

*EGI
ELSE
*EMI
*ENABLE
END
*END-OF-PAGE
*ENTER
ENVIRONMENT
*EOP
EQUAL
ERROR
*ESI
*EVERY
EXCEPTION

**EXHIBIT
EXIT
EXTEND

FD
FILE
FILE-CONTROL
**FILE-ID
FILLER
*FINAL
FIRST
*FOOTING
FOR
FROM

*GENERATE
GIVING
GO
GREATER
*GROUP

*HEADING
HIGH-VALUE(S)

IDENTIFICATION
IF
IN
INDEX
INDEXED
INITIAL
*INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS
I-O
I-O-CONTROL

JUST(IFIED)

KEY

LABEL
*LAST
LEADING
LEFT
*LENGTH
LESS

*LIMIT(S)
*LINAGE
*LINAGE-COUNTER
LINE(S)
*LINE-COUNTER
LINKAGE
LOCK
LOW-VALUE(S)

MEMORY
*MERGE
*MESSAGE
MODE
MODULES
MOVE
*MULTIPLE
MULTIPLY

**NAMED
NATIVE
NEGATIVE
NEXT
*NO
NOT
*NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
*OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW

PAGE
*PAGE-COUNTER
PERFORM
*PF
*PH
PIC(TURE)
*PLUS
POINTER
POSITION
POSITIVE
**PRINTER
*PRINTING

```
  PROCEDURE(S)
  PROCEED
  PROGRAM
  PROGRAM-ID
**PROMPT

 *QUEUE
  QUOTE(S)

  RANDOM
 *RD
  READ
**READY
 *RECEIVE
  RECORD(S)
  REDEFINES
 *REEL
 *REFERENCES
  RELATIVE
 *RELEASE
 *REMAINDER
 *REMOVAL
 *RENAMES
  REPLACING
 *REPORT(S)
 *REPORTING
 *RERUN
  RESERVE
  RESET
 *RETURN
 *REVERSED
 *REWIND
 *REWRITE
 *RF
 *RH
  RIGHT
  ROUNDED
  RUN

  SAME
 *SD
  SEARCH
  SECTION
  SECURITY
 *SEGMENT
 *SEGMENT-LIMIT
  SELECT
 *SEND
  SENTENCE
  SEPARATE
  SEQUENCE
  SEQUENTIAL
  SET
  SIGN
  SIZE
*SORT
*SORT-MERGE
*SOURCE
  SOURCE-COMPUTER
  SPACES(S)
  SPECIAL-NAMES
  STANDARD
  STANDARD-1
  START
  STATUS
  STOP
  STRING
*SUB-QUEUE-1,2,3
  SUBTRACT
*SUM
*SUPPRESS
*SYMBOLIC
  SYNC(HRONIZED)

*TABLE
  TALLYING
*TAPE
*TERMINAL
*TERMINATE
*TEXT
  THAN
  THROUGH
  THRU
  TIME
  TIMES
  TO
*TOP
**TRACE
  TRAILING
*TYPE

*UNIT
  UNSTRING
  UNTIL
  UP
  UPON
  USAGE
  USE
  USING
```

```
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).

Index

MICROSOFT

COBOL-80

user's manual

Foreword

The current release of COBOL-80 (Version 1.0) runs under the CP/M operating system as described in Section 4 of the Microsoft Utility Software Manual. Future releases of COBOL-80 will run under ISIS-II and other operating systems, as dictated by user demand.

Microsoft
COBOL-80 User's Manual

CONTENTS

SECTION 1

Compiling COBOL Programs

1.1 COBOL-80 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-80 command scanner. Those familiar with Microsoft's FORTRAN-80 and MACRO-80 will find the command format is identical for COBOL-80. However, different switches (options) are used with COBOL-80.

1.1.1 Format of Commands

COBOL-80 is invoked by typing COBOL followed by a space, followed by an appropriate command string, as described below. COBOL-80 is read from the disk and then examines the command string. If it is ok, compilation commences. If not, COBOL-80 responds with "?COMMAND ERROR" followed by an asterisk so the user can try again. When finished, COBOL-80 always exits to the operating system.

The general format of a COBOL-80 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-80 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. See Section 4 of the Microsoft Utility Software Manual for the defaults supplied by CP/M and other operating systems.

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. 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 but with the default extension for object files. Similarly "/L" may be used to place the listing file on the same disk with the same name as the source with the default extension for listing files.

Examples (using CP/M default extensions):

=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 is generated.
PAYOBJ=PAYROLL.COB	Compile PAYROLL.COB putting the object into PAYOBJ.REL.
PAYROLL,PAYROLL=PAYROLL	Compile 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-80 Compilation Switches

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

Switch	Action
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 not needed.

1.2 Output Listings and Error Messages

The listing file output by COBOL-80 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 below, and compilation continues.

Flag	Reason for Flag	Continuation Action
"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.	Excessive 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, based upon the assumption that a COBOL-80 Reference Manual is available.

1.3 Files Used by COBOL-80

In addition to the Source, Listing and Object files used by COBOL-80, 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, the user should never run into it 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 simply gives the name of the source file to be read in 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

Runtime Execution

2.1 Printer File Handling

Printer files should be viewed simply as a stream of characters going to the printer. Records should be defined simply 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 however, 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.

2.2 Disk File Handling

Disk files must have "LABEL RECORD IS STANDARD" declared and have a "VALUE OF" clause that includes a File Specification. 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 'FF'.

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.

2.3 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 a sequential access file when the last operation was not a successful READ.
REDUNDANT CLOSE	Attempt to close file that is not open.
GO TO. (NOT SET)	Attempt to execute an uninitialized alterable paragraph containing only a null GO statement.
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.

MICROSOFT

utility software manual

Microsoft
Utility Software Manual

CONTENTS

SECTION 1

MACRO-80 Assembler

1.1 Format of MACRO-80 Commands

1.1.1 MACRO-80 Command Strings

To run MACRO-80, type M80 followed by a carriage return. MACRO-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. The format of a MACRO-80 command string is:

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

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

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

source-dev:
The device from which the source-program input to MACRO-80 is obtained. If a device name is omitted, it defaults to the currently selected drive.

filename.ext
The filename and filename extension of the object program file, the listing file, and the source file. Filename extensions may be omitted. See Section 4 for the default extension supplied by your operating system.

Either the object file or the listing file 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. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples:

*=SOURCE.MAC	Assemble the program SOURCE.MAC and place the object in SOURCE.REL
*,LST:=TEST	Assemble the program TEST.MAC and list on device LST

*SMALL,TTY:=TEST	Assemble the program TEST.MAC, place the object in SMALL.REL and list on TTY

1.1.2 MACRO-80 Switches

A number of different switches may be given in the MACRO-80 command string that will affect the format of the listing file. Each switch must be preceded by a slash (/):

Switch	Action
O	Print all listing addresses, etc. in octal. (Default for Altair DOS)
H	Print all listing addresses, etc. in hexadecimal. (Default for non-Altair versions)
R	Force generation of an object file.
L	Force generation of a listing file.
C	Force generation of a cross reference file.

Examples:

*=TEST/L	Compile TEST.MAC with object file TEST.REL and listing file TEST.LST
*LAST,LAST/C=MOD1	Compile MOD1.MAC with object file LAST.REL and cross reference file LAST.CRF for use with CREF-80 (See Section 1.8)

1.2 Format of MACRO-80 Source Files

In general, MACRO-80 accepts a source file that is almost identical to source files for INTEL compatible assemblers. Input source lines of up to 132 characters in length are acceptable.

The assembler outputs a module name to the loader. This module name consists of the first six characters of the title if a TITLE statement is included. If no TITLE statement is included, the module name is created from the source file name.

1.3 Assembler Features

The features of the MACRO-80 assembler are described briefly below.

1.3.1 Names

All names are 1-6 characters. The first character is an alpha character (A-Z) or $. The remaining characters may be alphanumeric (A-Z, 0-9) or . or $ or ? or @. Names followed immediately by two number signs with no intervening blanks (e.g. NAME##) are classified as external. This type of name is an alternative to the program statement

```
    EXT    NAME
or
    EXTRN  NAME
```

1.3.2 Constants

a. Decimal: Numbers consisting of decimal digits and having no leading zero. The allowable range is 65535 to -65535.

b. Octal: Numbers consisting of octal digits and having a leading zero or a trailing Q or O. The allowable range is 0177777 to -0177777.

c. Hex: Numbers consisting of one to four hexadecimal digits and having the form x'hhhh'. One-digit or three-digit values are treated as though zero were to the left (i.e., X'A' and X'0A' are the same). The allowable range is X'FFFF' to -X'FFFF'. Numbers consisting of from one to four hexadecimal digits immediately followed by the suffix H (e.g., hhhhH) are also allowed.

d. Binary: Numbers consisting of a string of binary digits (0's and 1's) followed by a B. (e.g., 101011B)

e. Character: One or two ASCII characters preceded and followed by quotation marks (i.e., "a" or "BC" or 'BC'). The delimiters may be either single quotes (') or double quotes ("), but the starting and end delimiters

must be identical. Whenever one type of quote is used as a delimiter, the other type of quote is allowed as a character. Two-character strings are stored in low order byte/high order byte sequence. See Section 1.4.4.

1.3.3 Labels

A label is a name that does not contain an imbedded space and is terminated by a colon (:). Labels alone on a line with no further opcode or pseudo-op are allowed.

1.3.4 Operators

An operator consists of an 8080 mnemonic or one of the pseudo-operations described in Section 1.4.

1.3.5 Address Expressions

An address expression uses the current assigned address of a name or the 16-bit value of a constant to form a 16-bit value which, after the expression is evaluated, is truncated to the field size required by the operator.

1.3.6 Remarks

A remark always begins with a semicolon (;) and ends with a carriage return. A remark may be a line by itself or it may be appended to a line that contains a statement.

1.3.7 Statement Form

A statement consists of an optional label followed by an operator, followed by as many address expressioks as the operator requires, followed by an optional remark, and terminated by a carriage return. It is not necessary that statements begin in column 1. Multiple blanks or tabs may be used to improve readability (except inside character constants or character strings).

1.3.8 Expression Evaluation

Operator precedence during expression evaluation is

as follows:

```
Parenthesized expressions
HIGH, LOW
*, /, MOD, SHL, SHR
+, - (unary and binary)
Relational Operators EQ, LT, LE, GT, GE, NE
Logical NOT
Logical AND
Logical OR, XOR
```

The Relational, Logical and HIGH/LOW operators must be separated from their operands by at least one space.

Byte Isolation Operators

The byte isolation operators are as follows:

HIGH	Isolate the high order 8 bits of a 16-bit value
LOW	Isolate the low order 8 bits of a 16-bit value

Example:

```
IF HIGH VALUE EQ 0
```

The above IF pseudo-op determines whether the high order byte of VALUE is zero.

Relational Operators

The relational operators are as follows:

EQ	Equal
NE	Not equal
LT	Less than
LE	Less than or equal
GT	Greater than
GE	Greater than or equal

These operators yeild a true or false result. They are commonly used in conditional IF pseudo-ops. They must be separated from their operands by spaces. Example:

```
IF LABEL1 EQ LABEL2
```

The above pseudo-op tests the values of LABEL1 and LABEL2 for equality. If the result of the comparison is true, the assembly language code following the IF pseudo-op is assembled, otherwise the code is ignored.

1.3.9 Opcodes as Operands

8080 opcodes are valid one-byte operands. Note that only the first byte is a valid operand. For example:

```
MVI     A,(JMP)
ADI     (CPI)
MVI     B,(RNZ)
CPI     (INX H)
ACI     (LXI B)
MVI     C,(MOV A,B)
```

Errors will be generated if more than one byte is included in the operand -- such as (CPI 5), (LXI B,LABEL1) or (JMP LABEL2).

Opcodes used as one-byte operands need not be enclosed in parentheses.

1.4 Pseudo Operations

1.4.1 Define Byte

```
        DB      E1,E2,...,En
or
        DB      "Character String"
or
        DB      'Character String'
```

Each of the address expressions E1, E2,...En is evaluated and stored in n successive bytes. One- and two-character strings can be used in any expression. A string that is longer than two characters may only be used as a string.

Either single or double quotes may be used as character string delimiters, but the starting and end delimiters must be identical. It is permissible to use the delimiter quotes as characters, but the quote marks must appear twice for every character occurrence desired. For example:

```
        DB      "I am ""great"" today"
will store
                I am "great" today
```

Each character in the character string is stored as one byte with its high-order bit set to zero.

1.4.2 Define Character

```
DC    "Character String"
```

Only double quotes may be used as character string delimiters, and double quotes may not be used as characters.

Each character in the character string is stored as one byte with its high-order bit set to zero except for the last byte which has its high-order bit set to one.

1.4.3 Define Space

```
DS    E
```

The address expression E is evaluated and that many bytes of space are allocated. All names used in E must be defined prior to the DS statement.

1.4.4. Define Word

```
DW    E1, E2, ..., En
```

Each address expression is evaluated and stored as n successive words. Example:

```
DW    'AB'
```

Two-byte values are stored in memory in low order byte/high order byte sequence. The ASCII code representation for character B is stored, then the character A is stored.

On the object code listing however, the printout for all two-byte values is in high order byte/low order byte sequence, for easier reading.

1.4.5 Program Termination

```
END    E
```

This statement is the last statement of each program. The optional address expression E gives the program execution address. If E evaluates to absolute zero, it is equivalent to no execution address.

1.4.6 Terminated Conditional Assembly

```
ENDIF
```

Terminates conditional assembly initiated by a previous IFF or IFT.

1.4.7 Define Entry Points

```
ENTRY     N1, N2, ..., Nn
```
or
```
PUBLIC    N1, N2, ..., Nn
```

The names N1, N2, ..., Nn are entry points from external programs and act as names for the program being assembled. The names must appear in an ENTRY or PUBLIC statement prior to their appearance as a label.

1.4.8 Define Equivalence

```
Label   EQU    E
```

The label of the EQU statement is assigned the address given by address expression E. The label is required and must not have previously appeared as a label. All names used in E must be defined prior to the EQU statement.

1.4.9 Define External

```
EXT     N1, N2, ..., Nn
```
or
```
EXTRN   N1, N2, ..., Nn
```

The names N1, N2, ..., Nn are defined to be external references and may not have been used as a label. Names may also be defined as external by using NAME##. See Section 1.3.1.

1.4.10 False Conditional Assembly

```
IFF    E
```

The address expression E is evaluated and if it is False (=0), all statements down to the next ENDIF are assembled. If E is True (not =0), the statements are not assembled.

1.4.11 True Conditional Assembly

```
        IFT     E
or
        IF      E
```

The address expression E is evaluated and if it is True (not =0), all statements down to the next ENDIF are assembled. If E is False (=0), the statements are not assembled. Unlimited nesting of conditionals is allowed.

1.4.12 Define Origin

```
        ORG     E
```

The address expression E is evaluated and the assembler assigns generated code starting with that value. All names used in E must be defined prior to the ORG statement, and the mode of E must not be external.

1.4.13 Page Break

```
        PAGE
```

A page break will occur on the listing. The PAGE statement will not list and code is not generated. If a TITLE statement has been included, the title (up to 125 characters) will be printed at the top of the page.

1.4.14 Set

```
        Label   SET     E
```

The label of the SET statement is assigned the address given by expression E. The label is required and must not have previously appeared as a label. All names used in E must be defined prior to the SET statement.

The difference between the SET and EQU statements is that SET allows redefinition of label values. Redefinition of a label by an EQU statement will result in an error.

1.4.15 Title

```
TITLE ICOMP INTEGER COMPARE ROUTINE
```

TITLE followed by a title of up to 125 characters is allowed. This title will appear at the top of each page. The title must be terminated by a carriage return. The module name that the assembler outputs to the loader is taken from the first six characters that follow the TITLE statement. If no TITLE statement is included, the assembler outputs to the loader a module name that is taken from the file name.

1.4.16 Memory Segment Specification

It is possible to specify that sections of a program be loaded in absolute, code relative or data relative segments of memory. The pseudo-ops are:

ASEG	For loading in an absolute segment of memory
DSEG	For loading in a data relative segment of memory
CSEG	For loading in a code relative segment of memory

One of the possible uses of these pseudo-ops is to specify RAM and ROM segments of memory. The data relative segment would be RAM, and the code relative segment would be ROM.

After an ASEG, CSEG, or DSEG pseudo-op is encountered, all following code is loaded in that area until a subsequent ASEG, CSEG or DSEG pseudo-op is encountered.

If none of these three pseudo-ops is specified, the default condition is to load everything code relative.

Additional flexibility in relocating code is possible through use of the ORG pseudo-op, which sets the value of the appropriate program counter. For example:

```
DSEG        Sets the data relative program
ORG 50      counter to a value of 50
```

NOTE

1. The Intel operands PAGE and INPAGE will generate expression errors when used with CSEG or DSEG pseudo-ops. These errors are warnings; the assembler ignores the operands.

2. In version 3.0 of the MACRO-80 Assembler, references to a particular external symbol may not be made in more than one memory segment. For example, an external symbol EXT1 might be referenced in the code relative segment, external symbols EXT3, EXT4 might be referenced in the data relative segment, but none could be referenced in more than one memory segment. (This restriction will be removed in a later release of the MACRO-80 Assembler.)

Refer to Section 2, LINK-80 Linking Loader, to determine how these segments are placed in specific areas of memory.

1.5 Notes

1. A dollar sign ($) indicates the value of the location counter at the start of the statement.

2. When the assembler is entered, the origin is assumed to be Relative-0.

3. Address expressions used in the conditional assembly pseudo-operations IFF and IFT must have all names defined prior to the use in the expression, and the expression must be Absolute.

4. Address expressions whose final mode is other than Absolute must generate assembly data that is stored as two bytes.

5. The following names are defined by the assembler to have the indicated Absolute values.

A=7	B=0	C=1	D=2	E=3
H=4	L=5	M=6	SP=6	PSW=6

1.6 Sample Assembly

```
A>M80

*EXMPL1,TTY:=EXMPL1

        MAC80 3.0       PAGE    1

                        00100   ;CSL3(P1,P2)
                        00200   ;SHIFT P1 LEFT CIRCULARLY 3 BITS
                        00300   ;RETURN RESULT IN P2
  0000'                 00400           ENTRY   CSL3
                        00450   ;GET VALUE OF FIRST PARAMETER
                        00500   CSL3:
  0000'  7E             00600           MOV     A,M
  0001'  23             00700           INX     H
  0002'  66             00800           MOV     H,M
  0003'  6F             00900           MOV     L,A
                        01000   ;SHIFT COUNT
  0004'  06 03          01100           MVI     B,3
  0006'  AF             01200   LOOP:   XRA     A
                        01300   ;SHIFT LEFT
  0007'  29             01400           DAD     H
                        01500   ;ROTATE IN CY BIT
  0008'  17             01600           RAL
  0009'  85             01700           ADD     L
  000A'  6F             01800           MOV     L,A
                        01900   ;DECREMENT COUNT
  000B'  05             02000           DCR     B
                        02100   ;ONE MORE TIME
  000C'  C2 0006 '      02200           JNZ     LOOP
  000F'  EB             02300           XCHG
                        02400   ;SAVE RESULT IN SECOND PARAMETER
  0010'  73             02500           MOV     M,E
  0011'  23             02600           INX     H
  0012'  72             02700           MOV     M,D
  0013'  C9             02800           RET
  0014'                 02900           END

        MAC80 3.0       PAGE    2


CSL3    0000'   LOOP    0006'
```

1.7 MACRO-80 Errors

MACRO-80 errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Below is a list of the MACRO-80 Error Codes:

Code	Meaning
C	Too many ENDIFs
D	Bad octal or hex or binary digit
E	Expression error
L	No label in EQU
M	Label or symbol defined more than once
N	Name too long
O	Bad operator (opcode)
T	Illegal field termination
U	Undefined symbol
2	Missing second field for opcode
P	Phase error
Q	Missing or incorrect character string delimiter

1.8 Cross Reference Facility

The Cross Reference Facility is invoked by typing CREF80. In order to generate a cross reference listing, the assembler must output a special listing file with embedded control characters. The MACRO-80 command string tells the assembler to output this special listing file. /C is the cross reference switch. When the /C switch is encountered in a MACRO-80 command string, the assembler opens a .CRF file instead of a .LST file.

Examples:

*=TEST/C	Assemble file TEST.MAC and create object file TEST.REL and cross reference file TEST.CRF
*T,U=TEST/C	Assemble file TEST.MAC and create object file T.REL and cross reference file U.CRF.

When the assembler is finished, it is necessary to call the cross reference facility by typing CREF80.

The command string is:

```
*listing file=source file
```

The default extension for the source file is .CRF. The /L switch is ignored, and any other switch will cause an error message to be sent to the terminal. Possible command strings are:

*=TEST	Examine file TEST.CRF and generate a cross reference listing file TEST.LST.
*T=TEST	Examine file TEST.CRF and generate a cross reference listing file T.LST.

Cross reference listing files differ from ordinary listing files in that:

1. Each source statement is numbered.

2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined.

SECTION 2

LINK-80 Linking Loader

2.1 Format of LINK-80 Commands

2.1.1 LINK-80 Command Strings

To run LINK-80, type L80 followed by a carriage return. LINK-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. Each command to LINK-80 consists of a string of filenames and switches separated by commas:

objdev1:filename.ext/switch1,objdev2:filename.ext,...

If the input device for a file is omitted, the default is the currently logged disk. If the extension of a file is omitted, the default is .REL. After each line is typed, LINK will load or search (see /S below) the specified files. After LINK finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Example:

```
*MAIN

 DATA    0100    0200

 SUBR1*      (SUBR1 is undefined)

 DATA    0100    0300

*SUBR1
*/G          (Starts Execution - see below)
```

Typically, to execute a FORTRAN and/or COBOL program and subroutines, the user types the list of filenames followed by /G (begin execution). Before execution begins, LINK-80 will always search the system library (FORLIB.REL or COBLIB.REL) to satisfy any unresolved external references. If the user wishes to first search libraries of his own, he should append the filenames that are followed by /S to the end of the loader command string.

2.1.2 LINK-80 Switches

A number of switches may be given in the LINK-80 command string to specify actions affecting the loading process. Each switch must be preceded by a slash (/). These switches are:

Switch	Action
R	Reset. Put loader back in its initial state. Use /R if you loaded the wrong file by mistake and want to restart. /R takes effect as soon as it is encountered in a command string.
E or E:Name	Exit LINK-80 and return to the Operating System. The system library will be searched on the current disk to satisfy any existing undefined globals. The optional form E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program. Use /E to load a program and exit back to the monitor.
G or G:Name	Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on the current disk to satisfy any existing undefined globals if they exist. Before execution actually begins, LINK-80 prints three numbers and a BEGIN EXECUTION message. The three numbers are the start address, the address of the next available byte, and the number of 256-byte pages used. The optional form G:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.
N	If a <filename>/N is specified, the program will be saved on disk under the selected name (with a default extension of .COM for CP/M) when a /E or /G is done. A jump to the start of the program is inserted if needed so the program can run properly (at 100H for CP/M).

P and D

/P and /D allow the origin(s) to be set for the next program loaded. /P and /D take effect when seen (not deferred), and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in the current typeout radix. (Default radix for non-MITS versions is hex. /O sets radix to octal; /H to hex.) LINK-80 does a default /P:<link origin>+3 (i.e., 103H for CP/M and 4003H for ISIS) to leave room for the jump to the start address.

NOTE: Do not use /P or /D to load programs or data into the locations of the loader's jump to the start address (100H to 102H for CPM and 2800H to 2802H for DTC), unless it is to load the start of the program there. If programs or data are loaded into these locations, the jump will not be generated.

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin. Example:

```
*/P:200,FOO
Data    200     300
*/R
*/P:200 /D:400,FOO
Data    400     480
Program 200     280
```

U

List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

M

List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information

is only printed if a /D has been done. Otherwise, the program is stored in the data area.

S — Search the filename immediately preceding the /S in the command string to satisfy any undefined globals.

Examples:

```
*/M                 List all globals

*MYPROG,SUBROT,MYLIB/S
                    Load MYPROG.REL and SUBROT.REL and
                    then search MYLIB.REL to satisfy
                    any remaining undefined globals.

*/G                 Begin execution of main program
```

2.2 Sample Link

```
A>L80
*EXAMPL,EXMPL1/G
DATA    3000    30AC
[304F   30AC    49]
[BEGIN EXECUTION]

         1792           14336
        14336          -16383
       -16383              14
           14             112
          112             896
A>
```

2.3 Format of LINK Compatible Object Files

NOTE

Section 2.3 is reference material for users who wish to know the load format of LINK-80 relocatable object files. Most users will want to skip this section, as it does not contain material necessary to the operation of the package.

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. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

00	Special LINK item (see below).
01	Program Relative. Load the following 16 bits after adding the current Program base.
10	Data Relative. Load the following 16 bits after adding the current Data base.
11	Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that is the same as the two-bit field above except 00 specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

```
1 00 xxxx    yy     zzz + characters of symbol name
           --------  ------------------------------
           A field            B field
```

xxxx	Four-bit control field (0-15 below)
yy	Two-bit address type field
zzz	Three-bit symbol length field

The following special types 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 special 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 (A is address, B is name)
8 Reserved for future expansion

The following special LINK items have an A field only:

9 External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10 Define size of Data area (A is size)
11 Set loading location counter to A
12 Chain address. A is head of chain, replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13 Define program size (A is size)
14 End program (forces to byte boundary)

The following special Link item has neither an A nor a B field:

15 End file

2.4 LINK-80 Error Messages

LINK-80 has the following error messages:

?No Start Address	A /G switch was issued, but no main program had been loaded.
?Loading Error	The last file given for input was not a properly formatted LINK-80 object file.
?Out of Memory	Not enough memory to load program.
?Command Error	Unrecognizable LINK-80 command.
?<file> Not Found	<file>, as given in the command string, did not exist.

%2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Re-order module loading sequence or change COMMON block definitions.

%Mult. Def. Global YYYYYY

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

%Overlaying [Program / Data] Area

A /D or /P will cause already loaded data to be destroyed.

?Intersecting [Program / Data] Area

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 given, the symbol specified was not defined.

Origin [Above / Below] Loader Memory, Move Anyway (Y or N)?

After a /E or /G was given, 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 given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit. In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File

A disk error occurred when the file was being saved.

2.5 Program Break Information

LINK-80 stores the address of the first free location in a global symbol called $MEMRY if that symbol has been defined by a program loaded. $MEMRY is set to the top of the data area +1.

NOTE

If /D is given and the data origin is less than the program area, the user must be sure there is enough room to keep the program from being destroyed. This is particularly true with the disk driver for FORTRAN-80 which uses $MEMRY to allocate disk buffers and FCB's.

SECTION 3

LIB-80 Library Manager (CP/M Versions Only)

LIB-80 is the object time library manager for CP/M versions of FORTRAN-80 and COBOL-80. LIB-80 will be interfaced to other operating systems in future releases of FORTRAN-80 and COBOL-80.

3.1 LIB-80 Commands

To run LIB-80, type LIB followed by a carriage return. LIB-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. Each command in LIB-80 either lists information about a library or adds new modules to the library under construction.

Commands to LIB-80 consists of an optional destination filename which sets the name of the library being created, followed by an equal sign, followed by module names separated by commas. The default destination filename is FORLIB.LIB. Examples:

```
*NEWLIB=FILE1 <MOD2>, FILE3,TEST

*SIN,COS,TAN,ATAN
```

Any command specifying a set of modules concatenates the modules selected onto the end of the last destination filename given. Therefore,

```
*FILE1,FILE2 <BIGSUB>, TEST
```

is equivalent to

```
*FILE1
*FILE2 <BIGSUB>
*TEST
```

3.1.1 Modules

A module is typically a FORTRAN or COBOL subprogram, main program or a MACRO-80 assembly that contains ENTRY statements.

The primary function of LIB-80 is to concatenate modules in .REL files to form a new library. In

order to extract modules from previous libraries or .REL files, a powerful syntax has been devised to specify ranges of modules within a .REL file.

The simplest way to specify a module within a file is simply to use the name of the module. For example:

```
SIN
```

But a relative quantity plus or minus 255 may also be used. For example:

```
SIN+1
```

specifies the module after SIN and

```
SIN-1
```

specifies the one before it.

Ranges of modules may also be specified by using two dots:

..SIN means all modules up to and including SIN.

SIN.. means all modules from SIN to the end of the file.

SIN..COS means SIN and COS and all the modules in between.

Ranges of modules and relative offsets may also be used in combination:

```
SIN+1..COS-1
```

To select a given module from a file, use the name of the file followed by the module(s) specified enclosed in angle brackets and separated by commas:

```
FORLIB <SIN..COS>

 or

MYLIB.REL <TEST>

 or

BIGLIB.REL <FIRST,MIDDLE,LAST>

 etc.
```

If no modules are selected from a file, then all

the modules in the file are selected:

```
TESTLIB.REL
```

3.2 LIB-80 Switches

A number of switches are used to control LIB-80 operation. These switches are always preceded by a slash:

/O Octal - set Octal typeout mode for /L command.

/H Hex - set Hex typeout mode for /L command (default).

/U List the symbols which would remain undefined on a search through the file specified.

/L List the modules in the files specified and symbol definitions they contain.

/C (Create) Throw away the library under construction and start over.

/E Exit to CP/M. The library under construction (.LIB) is revised to .REL and any previous copy is deleted.

/R Rename - same as /E but does not exit to CP/M on completion.

3.3 LIB-80 Listings

To list the contents of a file in cross reference format, use /L:

```
*FORLIB/L
```

When building libraries, it is important to order the modules such that any intermodule references are "forward." That is, the module containing the global reference should physically appear ahead of the module containing the entry point. Otherwise, LINK-80 may not satisfy all global references on a single pass through the library.

Use /U to list the symbols which could be undefined in a single pass through a library. If a module in the library makes a backward reference to a symbol in another module, /U will list that symbol. Example:

```
*SYSLIB/U
```

NOTE: Since certain modules in the standard FORTRAN and COBOL systems are always force-loaded, they will be listed as undefined by /U but will not cause a problem when loading FORTRAN or COBOL programs.

Listings are currently always sent to the terminal; use control-P to send the listing to the printer.

3.4 Sample LIB Session

```
A>LIB

*TRANLIB=SIN,COS,TAN,ATAN,ALOG
*EXP
*TRANLIB.LIB/U
*TRANLIB.LIB/L
  .
  .
  .
(List of symbols in TRANLIB.LIB)
  .
  .
  .
*/E
A>
```

3.5 Summary of Switches and Syntax

```
/O  Octal - set listing radix
/H  Hex - set listing radix
/U  List undefineds
/L  List cross reference
/C  Create - start LIB over
/E  Exit - Rename .LIB to .REL and exit
/R  Rename - Rename .LIB to .REL
```

module::=module name {+ or - number}

module sequence ::=

module | ..module | module.. | module1..module2

file specification::=filename {<module sequence>{,<module sequence>}}

command::= {library filename=} {list of file specifications}
{list of switches}

SECTION 4

Operating Systems

This section describes the use of MACRO-80 and LINK-80 under the different disk operating systems. The examples shown in this section assume that the FORTRAN-80 compiler is in use. If you are using the COBOL-80 compiler, substitute "COBOL" wherever "F80" appears, and substitute the extension ".COB" wherever ".FOR" appears.

4.1 CPM

Create a Source File

Create a source file using the CPM editor. Filenames are up to eight characters long, with 3-character extensions. FORTRAN-80 source filenames should have the extension FOR, COBOL-80 source filenames should have the extension COB, and MACRO-80 source filenames should have the extension MAC.

Compile the Source File

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on a source file called MAX1.FOR, type

```
A>F80 ,=MAX1
```

This command compiles the source file MAX1.FOR without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file and produce an object and listing file, type

```
A>F80 MAX1,MAX1=MAX1
```

or

```
A>F80 =MAX1/L
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.PRN.

Loading, Executing and Saving the Program (Using LINK-80)

To load the program into memory and execute it, type

```
A>L80 MAX1/G
```

To exit LINK-80 and save the memory image (object code), type

```
A>L80 MAX1/E,MAX1/N
```

When LINK-80 exits, three numbers will be printed: the starting address for execution of the program, the end address of the program and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

If you wish to use the CPM SAVE command to save a memory image, the number of pages used is the argument for SAVE. For example

```
A>SAVE 48 MAX1.COM
```

NOTE

> CP/M always saves memory starting at 100H and jumps to 100H to begin execution. Do not use /P or /D to set the origin of the program or data area to 100H, unless program execution will actually begin at 100H.

An object code file has now been saved on the disk under the name specified with /N or SAVE (in this case MAX1). To execute the program simply type the program name

```
A>MAX1
```

CPM - Available Devices

```
A:, B:, C:, D:  disk drives
HSR:    high speed reader
LST:    line printer
TTY:    Teletype or CRT
```

CPM Disk Filename Standard Extensions

```
FOR     FORTRAN-80 source file
COB     COBOL-80 source file
MAC     MACRO-80 object file
REL     relocatable object file
PRN     listing file
COM     absolute file
```

CPM Command Lines
CPM command lines and files are supported; i.e., a COBOL-80, FORTRAN-80, MACRO-80 or LINK-80 command line may be placed in the same line with the CPM run command. For example, the command

```
A>F80 =TEST
```

causes CPM to load and run the FORTRAN-80 compiler, which then compiles the program TEST.FOR and creates the file TEST.REL. This is equivalent to the following series of commands:

```
A>F80
*=TEST
*^C
A>
```

4.2 DTC Microfile

Create a Source File
Create a source file using the DTC editor. Filenames are up to five characters long, with 1-character extensions. COBOL-80, FORTRAN-80 and MACRO-80 source filenames should have the extension T.

Compile the Source File
Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1, type

```
*F80 ,=MAX1
```

This command compiles the source file MAX1 without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1 and produce an object and listing file, type

```
*F80 MAX1,MAX1=MAX1
```

or

```
*F80 =MAX1/L/R
```

The compiler will create a relocatable file called MAX1.O and a listing file called MAX1.L.

Loading, Executing and Saving the Program (Using LINK-80)
To load the program into memory and execute it,

type

```
*L80 MAX1/G
```

To save the memory image (object code), type

```
*L80 MAX1/E
```

which will exit from LINK-80, return to the DOS monitor and print three numbers: the starting addressfor execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

Use the DTC SAVE command to save a memory image. For example

```
*SA MAX1 2800 401A 2800
```

2800H (24000Q) is the load address used by the DTC Operating System.

NOTE

If a /P:<address> or /D:<address> has been included in the loader command to specify an origin other than the default (2800H), make sure the low address in the SAVE command is the same as the start address of the program.

An object code file has now been saved on the disk under the name specified in the SAVE command (in this case MAX1). To execute the program, simply type

```
*RUN MAX1
```

DTC Microfile - Available Devices

DO:, D1:, D2:, D3:	disk drives
TTY:	Teletype or CRT
LIN:	communications port

DTC Disk Filename Standard Extensions

T	COBOL-80, FORTRAN-80 or MACRO-80 source file
O	relocatable object file
L	listing file

DTC Command Lines
DTC command lines are supported as described in Section 4.1, CPM Command Lines.

4.3 Altair DOS

Create a Source File
Create a source file using the Altair DOS editor. The name of the file should have four characters, and the first character must be a letter. For example, to create a file called MAX1, initialize DOS and type

```
.EDIT MAX1
```

The editor will respond

```
CREATING FILE
00100
```

Enter the program. When you are finished entering and editing the program, exit the editor.

Compile the Source File
Load the compiler by typing

```
.F80
```

The compiler will return the prompt character "*".

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1, type

```
*,=&MAX1.
```

(The editor stored the program as &MAX1) Typing ,=&MAX1. compiles the source file MAX1 without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1 and produce an object and listing file, type

```
*MAX1R,&MAX1=&MAX1.
```

The compiler will create a REL (relocatable) file called MAX1RREL and a listing file called &MAX1LST. The REL filename must be entered as five characters instead of four, so it is convenient to use the source filename plus R.

After the source file has been compiled and a prompt has been printed, exit the compiler. If the computer uses interrupts with the terminal, type Control C. If not, actuate the RESET switch on the computer front panel. Either action will return control to the monitor.

Using LINK-80
Load LINK-80 by typing

```
.L80
```

LINK-80 will respond with a "*" prompt. Load the program into memory by entering the name of the program REL file

```
*MAX1R
```

Executing and Saving the Program
Now you are ready to either execute the program that is in memory or save a memory image (object code) of the program on disk. To execute the program, type

```
*/G
```

To save the memory image (object code), type

```
*/E
```

which will exit from LINK-80, return to the DOS monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[26301 44054 35]
```

Use the DOS SAVE command to save a memory image. Type

```
.SAV MAX1 0 17100 44054 26301
```

17100 is the load address used by Altair DOS for the floppy disk. (With the hard disk, use 44000.)

An object code file h[s now been saved on the disk under the name specified in the SAVE command (in this case MAX1). To execute the program, simply type the program name

```
.MAX1
```

Altair DOS - Available Devices

```
FO:, F1:, F2:, ...      disk drives
TTY:                    Teletype or CRT
```

Altair DOS Disk Filename Standard Extensions

```
FOR     FORTRAN-80 source file
COB     COBOL-80 source file
MAC     MACRO-80 source file
REL     relocatable object file
LST     listing file
```

Command Lines
Command lines are not supported by Altair DOS.

4.4 ISIS-II

Create a Source File
Create a source file using the ISIS-II editor. Filenames are up to six characters long, with 3-character extensions. FORTRAN-80 source filenames should have the extension FOR and COBOL-80 source filenames should have the extension COB. MACRO-80 source filenames should have the extension MAC.

Compile the Source File
Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1.FOR, type

```
-F80 ,=MAX1
```

This command compiles the source file MAX1.FOR without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1.FOR and produce an object and listing file, type

```
-F80 MAX1,MAX1=MAX1
```

or

```
-F80 =MAX1/L/R
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.LST.

Loading, Saving and Executing the Program (Using LINK-80)
To load the program into memory and execute it, type

```
-L80 MAX1/G
```

To save the memory image (object code), type

```
-L80 MAX1/E,MAX1/N
```

which will exit from LINK-80, return to the ISIS-II monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

An object code file has now been saved on the disk under the name specified with /N (in this case MAX1).

ISIS-II - Available Devices

F0:, F1:, F2:, ...	disk drives
TTY:	Teletype or CRT
LST:	line printer

ISIS-II Disk Filename Standard Extensions

FOR	FORTRAN-80 source file
COB	COBOL-80 source file
MAC	MACRO-80 source file
REL	relocatable object file
LST	listing file

ISIS-II Command Lines
ISIS-II command lines are supported as described in Section 4.1, CPM Command Lines.