IBM

Personal Computer
Computer Language
Series

COBOL Compiler

by Microsoft

6172258

International Business Machines Corporation Boca Raton, Florida 33432

IBM Program License Agreement

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THIS DISKETTE(S) OR CASSETTE(S) PACKAGE. OPENING THIS DISKETTE(S) OR CASSETTE(S) PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED; AND YOUR MONEY WILL BE REFUNDED.

IBM provides this program and licenses its use in the United States and Puerto Rico. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from the program.

LICENSE

You may:

a. use the program on a single machine;

b. copy the program into any machine readable or printed form for backup or modification purposes in support of your use of the program on the single machine (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected.");

c. modify the program and/or merge it into another program for your use on the single machine (Any portion of this program merged into another program will continue to be subject to the terms and conditions of this Agreement.); and,

d. transfer the program and license to another party if the other party agrees to accept the terms and conditions of this Agreement. If you transfer the program, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copies not transferred; this includes all modifications and portions of the program contained or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM, OR ANY COPY, MODIFICATION OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION OR MERGED PORTION OF THE PROGRAM TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form.

LIMITED WARRANTY

THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT IBM OR AN AUTHORIZED PERSONAL COMPUTER DEALER) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

Continued on inside back cover

Personal Computer
Computer Language
Series

COBOL Compiler

by Microsoft

First Edition (March 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

PREFACE

About This Book

This *IBM Personal Computer COBOL* manual provides both operating information and reference information for the IBM Personal Computer COBOL compiler. In order to use this manual, you should have some knowledge of general programming concepts; we are not trying to teach you how to program in this manual.

This manual is divided into ten chapters plus a number of appendixes. The first three chapters provide operating instructions and an introduction to the COBOL language.

Chapter 1 is a brief introduction and overview of the COBOL language.

Chapter 2 tells you what you need to know to start using COBOL on your IBM Personal Computer. It tells you how to operate your computer using IBM Personal Computer COBOL.

Chapter 3 covers a variety of topics which you need to know before you actually start programming. In this chapter, you learn how to:

- Create the source file for a program.
- Compile the program.
- Link the program.
- Run the program.

Chapters 4, 5, 6, and 7 are reference chapters. They describe the four divisions of a COBOL program. These chapters also provide the syntax and function of all of the paragraphs, sentences, clauses, and phrases used in each division. The syntax descriptions are in alphabetic order at the end of each of the four chapters.

Chapter 4. Identification Division

Chapter 5. Environment Division

Chapter 6. Data Division

Chapter 7. Procedure Division

The remaining three chapters discuss additional features of your IBM Personal Computer COBOL.

Chapter 8 tells you about file organization and the Procedure Division statements applicable to sequential, relative, and indexed files.

Chapter 9 discusses table handling by the indexing method. It includes the use of the SET and SEARCH statements.

Chapter 10 describes interprogram communication.

The appendixes contain other useful information, such as lists of messages, reserved words, ASCII codes, and math functions. You can also find detailed information on more advanced subjects for the experienced programmer.

We suggest you read through all of chapters 1, 2, and 3 to become familiar with IBM Personal Computer COBOL. Then you can refer to chapters 4, 5, 6, and 7 while you are actually programming to get information you need about each division.

Related Publications

While you are using this manual, you may find references to one or more of the other books in the IBM Personal Computer library. These books include:

- *IBM Personal Computer Disk Operating System*
- *IBM Personal Computer BASIC*
- *IBM Personal Computer FORTRAN*
- *IBM Personal Computer Pascal*
- *IBM Personal Computer MACRO ASSEMBLER*

The *BASIC* manual is provided with your IBM Personal Computer system.

The *Disk Operating System*, *FORTRAN*, *Pascal*, and *MACRO ASSEMBLER* manuals are provided when you purchase the individual software packages.

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.

"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 Processing.

"The authors and copyright holders of the copyrighted material used herein:

FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC® I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation;

IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM;

FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specification in programming manuals or similar publications."[1]

Runtime Module

Application programs written in IBM Personal Computer COBOL require the COBRUN.EXE runtime module. Information about this runtime module can be obtained by writing to IBM at:

Runtime Module
IBM Personal Computer
P.O. Box 1328-A
Boca Raton, Florida 33432

[1] The ANSI COBOL STANDARD (X3.23-1974)

CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Contents

What Is COBOL?

COBOL (Common Business Oriented Language) is a widely used language for computer applications. It is a means of communicating with your IBM Personal Computer in an English-like language. You use COBOL primarily for business applications, because it lacks floating point capabilities and transcendental mathematical functions.

COBOL does have extensive formatting and I/O capabilities, which let you organize, access, update, and report data in files. All of these tasks are important in business applications and the production of reports.

IBM Personal Computer COBOL and the National Standard

Your IBM Personal Computer COBOL conforms to the "Low-Intermediate" Level of the American National Standard X3.23-1974. It provides nine out of the 12 standard COBOL functional modules. These nine modules are implemented at least to Level 1 capabilities, and in many cases, include much of Level 2 (see explanation below).

The standard COBOL language has 12 functional processing *modules*:

Nucleus
Sequential I/O
Relative I/O
Indexed I/O
Library
Communication
Interprogram communication
Table handling
Sort/Merge
Debugging
Report writer
Segmentation

Each module of the COBOL Standard has two *levels*. Level 1 is 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 other nine modules may or may not be implemented.

The following list summarizes the characteristics of the 12 modules in IBM Personal Computer COBOL.

Module	Features of IBM Personal Computer 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

Module	Features of IBM Personal Computer COBOL
Nucleus (continued)	• Verbs: – Extensions to the functions of ACCEPT and DISPLAY for formatted screen handling – Acceptance of data from DATE/DAY/TIME – STRING and UNSTRING statements – COMPUTE with multiple receiving fields – PERFORM VARYING. . .UNTIL • Identifiers: – Mnemonic names for accept or display devices – Procedure names consisting of digits only – Qualification of names (in Procedure Division statements 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 • VALUE OF FILE-ID is data name • Sequential I/O: – EXTEND mode for OPEN – WRITE ADVANCING data name lines – LINAGE phrase – AT END-OF-PAGE clause

Module	Features of IBM Personal Computer COBOL
Sequential, Relative, and Indexed I/O (continued)	• Relative and Indexed I/O: – Dynamic access mode (with READ NEXT) – START (with key relations EQUAL, GREATER, or NOT LESS)
Library	All of Level 1
Communication	IBM Personal Computer COBOL does not provide this.
Interprogram Communication	All of Level 1
Table Handling	All of Level 1 plus full Level 2 formats for SEARCH statement
Sort/Merge	IBM Personal Computer COBOL does not provide this.
Debugging	• Special extensions to ANSI-74 Standard, providing convenient trace-style debugging. • Conditional compilation: lines with *D in column 7* are bypassed unless WITH DEBUGGING MODE is given in the SOURCE-COMPUTER paragraph.
Report Writer	IBM Personal Computer COBOL does not provide this.
Segmentation	All of Level 1

Exceptions

Referring to the Nucleus and Table Handling modules, your IBM Personal Computer COBOL includes all Level 2 features *except*:

- General
  - You cannot use figurative constant ALL *literal* for literals greater than one character.
  - You cannot qualify names allowed in the Environment Division.
- Data Division
  - OCCURS DEPENDING ON . . . is not supported.
  - You cannot intermix a Level 88 item containing a list of items with a range of items (either list or range may be used but not *both* at one time).
  - Binary data items always require 2 bytes:
    - PICTURE 9(5) only allows a range of -32768 to 32767.
    - PICTUREs 9, 99, 999, and 9999 are equivalent to PIC 9(5) for binary items.
    - An error message is given when more than five digits are specified.
  - Unsigned binary data items:
    - PIC 9 is equivalent to PIC S9.
  - RENAMES phrase is not supported.

- Procedure Division
  - MOVE, ADD, and SUBTRACT do not support CORRESPONDING.
  - Multiple destinations for results of arithmetic statements are not supported.
  - Division remainders are not provided.
  - INSPECT in Level 2 is not supported.
  - Arithmetic expressions in conditions are not supported.
  - ALTER series of procedure names is not supported.
  - Multiple index keys are not supported.
  - Special language for tape handling is not supported.
  - Level 1 RERUN facility is not supported.
  - Interprogram communication and Library modules are implemented to Level 1 only.

Summary

IBM Personal Computer COBOL is a powerful language for business applications. It includes all of 1974 ANSI COBOL Level 1 facilities and many Level 2 features.

Features of particular interest are trace style debugging and the extensions we have incorporated in interactive screen control, allowing special options to the ACCEPT and DISPLAY statements to handle fully formatted screens.

Still another extension is the COMP-3 data format which allows numeric data to be packed two digits to the byte so that diskette requirements are reduced.

> **Note:** For the remainder of this manual, we will use the term *IBM COBOL* to mean *IBM Personal Computer COBOL*.

CHAPTER 2. HOW TO WRITE A COBOL PROGRAM

Contents

Program Structure

COBOL is a highly structured language. When you write a COBOL program, you must follow specific rules about organization and formats. Once you learn those rules, your programming will become easy.

In this chapter, we present the divisions of a program, program coding structure, and fundamental COBOL concepts.

Divisions 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 while the program is running.

IBM COBOL cannot compile source code correctly if the division headers are omitted or are accidentally commented out.

Coding Structure

The following skeletal coding defines program 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 ...]...]
  [SCREEN SECTION.
    [screen-description-entry ...]...]
PROCEDURE DIVISION [USING|CHAINING [identifier-1]...].
  [DECLARATIVES.
    [section-name SECTION. use-sentence.
    [paragraph-name. [sentence]...]...]...
  END DECLARATIVES.]
    [[section-name SECTION. [segment number]]
    [paragraph-name. [sentence]...]...]...
```

Coding Rules

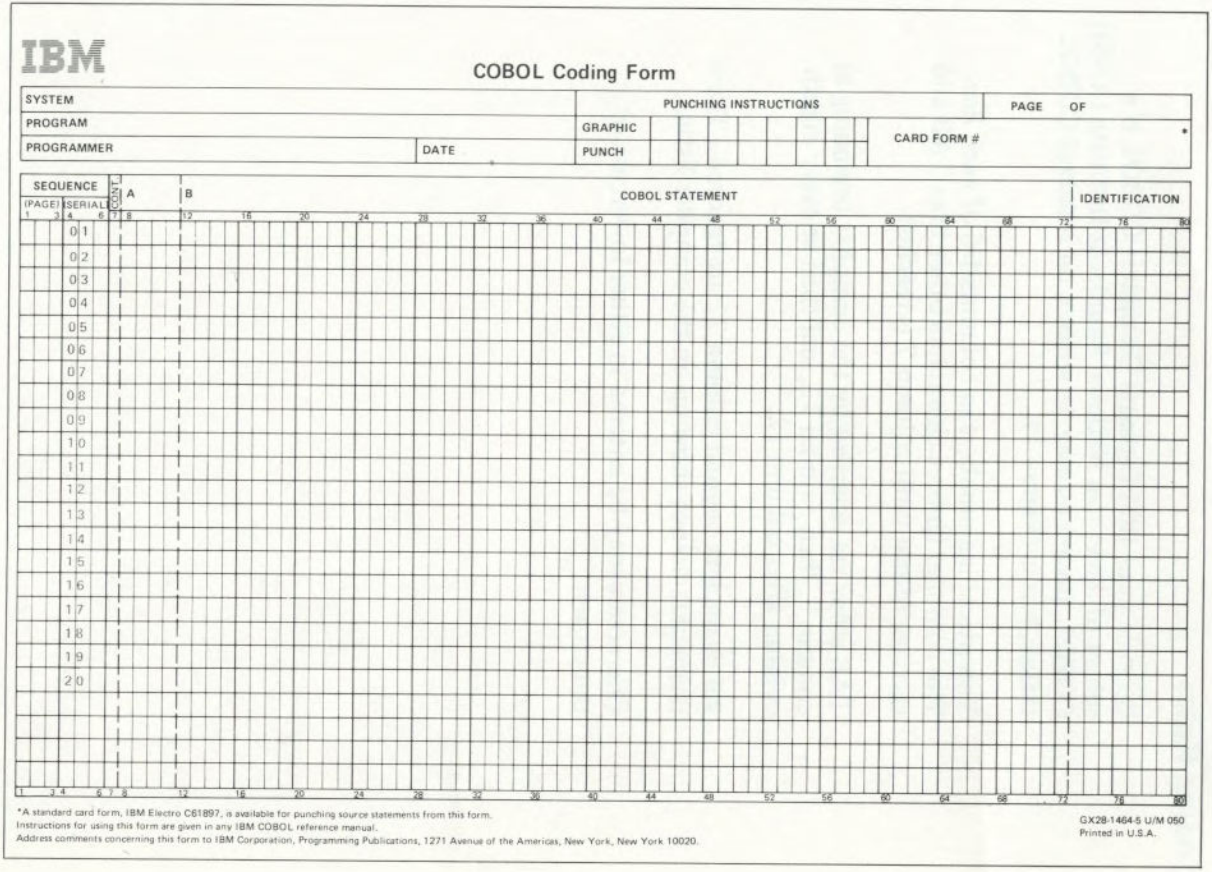
Because your IBM Personal Computer COBOL is a subset of American National Standards Institute (ANSI) COBOL, programs may be written on standard COBOL coding forms (Figure 1).

You place line numbers in columns 1-6 of each line. The compiler ignores characters other than TAB and carriage return until column 7 is reached.

TAB stops are assumed by the compiler beginning at column 8, then column 12, and then at every eighth column after column 12.

All characters beyond column 72 are ignored. These characters do not show up on the compiler listing.

Characters may be entered in either lowercase or uppercase.



IBM

COBOL Coding Form

SYSTEM		PUNCHING INSTRUCTIONS		PAGE OF
PROGRAM		GRAPHIC		CARD FORM # *
PROGRAMMER	DATE	PUNCH		

SEQUENCE (PAGE)	SEQUENCE (SERIAL)	CONT.	A	B	COBOL STATEMENT	IDENTIFICATION
1 2 3 4	5 6	7	8	12	16 20 24 28 32 36 40 44 48 52 56 60 64 68 72	76 80
	01					
	02					
	03					
	04					
	05					
	06					
	07					
	08					
	09					
	10					
	11					
	12					
	13					
	14					
	15					
	16					
	17					
	18					
	19					
	20					

*A standard card form, IBM Electro C61897, is available for punching source statements from this form.
Instructions for using this form are given in any IBM COBOL reference manual.
Address comments concerning this form to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020.

GX28-1464-5 U/M 050
Printed in U.S.A.

Figure 1. Example of Standard COBOL Coding Form

Refer to Figure 1 when reading the following rules:

1. Each line of code can have a 6-digit sequence number in columns 1-6. You should enter the program source lines in ascending order.

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 Area B (columns 12-72), governed by the other rules of statement punctuation.

4. Columns 73-80 are ignored by the compiler.

5. You may put comments or remarks on any line within a source program by placing an asterisk (*) or a slash (/) in column 7 of that line. The line shows up on the source listing but serves no other purpose.

   Additionally, the slash (/) causes a page eject, and the comment is written at the top of a new page.

6. Any program element may be "continued" on the following line of a source program. The rules for continuation of a nonnumeric ("quoted") literal are listed under "Nonnumeric Literals" in Chapter 2.

   You may continue other literals, words, or program elements by placing a hyphen in the column 7 position of the continuation line. When you do so, successive word parts are concatenated, except for all trailing spaces of the last predecessor word and all leading spaces of the first word on the continuation line. On a continuation line, Area A must be blank.

7. Any tab characters in a line are expanded as if there were tab stops at columns 8, 12, 20, 28, 36, ..., 73.

   Note on tabs: EDLIN (the editor provided with the IBM Personal Computer DOS) provides tab stops at every eighth position. This is different from the interpretation that the COBOL compiler uses. For example, tab position 2 is at column 16 in EDLIN and at column 12 in this compiler. Thus, all positions following tab stop 2 appear differently when viewed in EDLIN than when viewed in the **.LST** file.

   Also, you must remember when you used tabs and when you didn't, as this affects the insertion/deletion of characters whenever you edit the file.

Syntax Notation

The divisions, paragraphs, sentences, clauses, and phrases in this manual have their syntax described according to the following conventions:

1. Words in capital letters are *keywords* or *reserved words*. They must be input as shown. They may be entered in any combination of uppercase and/or lowercase.

2. All underlined words are required and must be entered as shown.

3. You must supply any items that are represented by lowercase *italic* letters. (For example, *filename*.)

4. Items in square brackets ([ ]) are optional.

5. When two or more items are separated by a vertical bar (|), you must select only one of the items. Also, items between matching braces ({}) represent a choice of mutually exclusive options.

6. An ellipsis (. . .) indicates that an item may be repeated as many times as you wish.

7. All punctuation except square brackets (such as periods, commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

Character Set

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

Letters A through Z (and a-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 for quotation mark)
/	Slash

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

0 through 9
A through Z (and a-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:

>	Greater than
<	Less than
=	Equal

In the case of nonnumeric (quoted) literals, comment entries, and comment lines, you can use any of the computer's entire character set.

Punctuation

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

1. As punctuation, a period, semicolon, or comma *must* be followed by a space.

2. At least one space *must* appear between two successive words and/or literals.

   Your computer treats two or more successive spaces as a single space, except in nonnumeric literals.

3. Relation characters should always be preceded by a space and followed by another space.

4. When you use the period, comma, plus, or minus characters in the PICTURE clause, you must follow specific rules for report formatting.

5. You may use a comma as a separator between successive operands of a statement, or between two subscripts.

6. You may use a semicolon or comma to separate a series of statements or clauses.

7. You may use an apostrophe (') in place of quotation marks (") when delimiting literals.

Word Formation

Reserved words and words you define yourself are from 1 to 30 characters long. You can use any combination chosen from the following set of 63 characters:

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

All words must contain at least one letter or hyphen, except procedure names, which may consist entirely of digits. A word may not begin or 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. (Remember that you can use any combination of uppercase or lowercase letters.)

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; for example, TAX-RATE IN STATE-TABLE.

Primarily, a nonreserved word identifies a data item or field and is called a *data name*. Other cases of nonreserved words are filenames, condition names, mnemonic names, and procedure names.

Statements, Sentences, and Names

The procedure portion of a source program specifies those procedures needed to solve a given problem. These steps, such as computations and logical decisions, are expressed in English-like statements, which use the concept of verbs to denote actions, and statements and sentences to describe procedures.

Statements

A *statement* is an instruction to the computer. It 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, as in:

```
MOVE 15 TO AGE.
ADD 1 TO YEAR.
WRITE PRINT-LINE FROM DATA-LINE.
```

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 ended by a period and followed by a space. You can use semicolons or commas 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.

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 names, followed by the word SECTION, an optional segment number, and a period. A section header must appear on a line by itself. Each section name must be unique.

Level Numbers and Data Names

In COBOL, data is defined in the Data Division. The data itself may be stored within the Working-Storage or Linkage Section of the program. Outside the program, data is stored in files, the format of which is described in the File Section of the program.

You refer to the data by using the name of the group in which the data is located, or by the name of the data item itself.

What is a Record?

Data is divided into logical records. A *record* is a collection of related data or words, treated as a unit. For example, an invoice or a time card could be considered a record. A *logical record* is the most inclusive record. A logical record is identified by the level number 01.

Data Items

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

Logical records are divided into more specific data items or levels. *Levels* allow you to subdivide records in order to refer to specific data items. Once a subdivision is specified, it may be further subdivided to permit more detailed data reference.

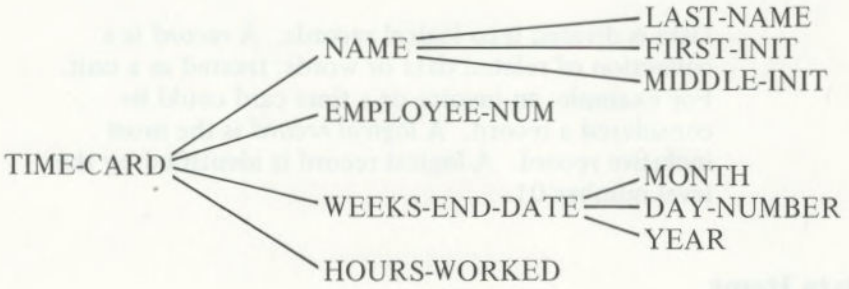
More specific data items in a logical record are grouped in a hierarchy and identified with level numbers 02 to 49. Level numbers of subordinate items are greater than those of the group items they are under.

Note: A level number less than 10 may be written as a single digit (for example, 1, 2, 3, etc.).

Level number 77 identifies a *stand alone* item in the Working-Storage or Linkage Sections. A stand alone item does not (and cannot) have subordinate elementary items as does level 01.

Level number 88 defines condition names and associated conditions.

Look at the following record (TIME-CARD), which is divided into four major items: NAME, EMPLOYEE-NUM, WEEKS-END-DATE, and HOURS-WORKED. More specific information appears for NAME and WEEKS-END-DATE.



Elementary Item

Any subdivision of a record that is not further subdivided is called an *elementary item*. In the TIME-CARD example, EMPLOYEE-NUM is an elementary item. All elementary items must be described with a PICTURE or USAGE IS INDEX clause. (See "PICTURE Clause" and "USAGE Clause" in Chapter 6 for information on the use of these clauses.)

Group Item

A data item that contains one or more elementary items (an item with subdivisions) is known as a *group item*. In the TIME-CARD example, NAME is a group item with three subdivisions. When a procedure statement makes reference to a group item, the reference applies to the area reserved for the entire group.

Ordinarily, the maximum size of any data item is 4095 bytes. In order to allow tables to exceed this limit, however, level 01 group items are not checked for length. Any such item longer than 4095 bytes is not allowed by the compiler as an operand of a Procedure Division statement.

Less inclusive groups are assigned numerically higher level numbers. Level numbers of items within groups need not be consecutive. In a record, for example, you could have level numbers 01, 02, 03; or, you could have level numbers 01, 02, 10.

A group whose level is "N" includes all groups and elementary items described under it until a level number less than or equal to "N" is encountered. In the example below, the level 02 NAME group in our TIME-CARD record includes the LAST-NAME, FIRST-INIT, and MIDDLE-INIT.

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. (The level numbers are the first two digits in each line.) (The PICTURE definitions are explained under "PICTURE Clause" in Chapter 6.)

```
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.
```

Types of Data Items

Alphanumeric: 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*. This type of item is called "an-form" in the syntax diagrams in this book.

Report (Edited): A *report (edited) 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. This type of item is called "report-form" in the syntax diagrams in this book.

Numeric: A *numeric item* is an elementary item that contains numeric data only. The method used to store numeric items is specified in the (optional) USAGE clause. This type of item is called "numeric-form" in the syntax diagrams in this book.

An *external decimal item* is an item in which one computer character (byte) represents one digit. A maximum number of 18 digits is permitted; the exact number of digit positions is defined by writing a picture description with a specific number of **9** characters. For example, PICTURE 999 defines a three-digit item. That is, the maximum decimal value of the item is 999.

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. Thus, displaying a value whose PIC is S9 gives A for 1 and J for –1. To display the sign separately, you must use the form PIC S9 SIGN SEPARATE. (See "USAGE Clause" in Chapter 6.)

Regardless of the form of representation of an operational sign, its purpose is to provide a sign that functions in an algebraic manner.

The USAGE of an external decimal item is COMPUTATIONAL, or DISPLAY.

An *internal decimal item* is stored in *packed decimal* format. You can specify packed decimal format with the COMPUTATIONAL-3 USAGE clause.

A packed decimal item defined by n **9**s in its PICTURE occupies one-half of (n + 2) (rounded down) bytes in memory. All bytes, except the right-most byte, 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 right-most 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. For example, a decimal value of 12345 with PIC 99999 is stored in three bytes as 12 34 5F.

A *binary item* uses the base-2 number system to represent an integer in the range from -32768 to 32767. It occupies two 8-bit bytes. The left-most bit of the reserved area is the operational sign. You specify a binary item with USAGE IS COMPUTATIONAL-0.

An *index data item* is specified by the USAGE IS INDEX clause. It has no PICTURE and is stored as a binary item. (Refer to Chapter 9, "Table Handling by the Indexing Method.")

Data Names

A *data name* is a word that you assign to identify a data item used in a program. A data name always refers to the contents of 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. For example, the value of HOURS-WORKED could be changed from 40 to 50, to 32, or to any other number while the program is running.

A data name must begin with an alphabetic character. A data name or the keyword 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 refers to the associated data area (containing the value of a data item).

If some of the contents 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 instead of a *data-name* after the level number.

Qualifier Names

You can refer to a data name, condition name or paragraph name that is not unique by using a *qualifier name*. For example, if there were two or more items named YEAR, you could use the qualifiers HIRE-DATE and TERMINATION-DATE to differentiate between the two year fields (as in YEAR OF HIRE-DATE, YEAR OF TERMINATION-DATE).

A qualifier must be preceded by either the word OF or the word IN. Qualifiers of data names or condition names must be group items. They must be subdivided into more specific data items. For example, a level 02 qualified data item HIRE-DATE could contain level 03 entries of MONTH, DAY, and YEAR. Then, you would have MONTH OF HIRE-DATE, DAY OF HIRE-DATE, and YEAR OF HIRE-DATE.

Paragraph names may be qualified by a section name. The maximum number of qualifiers is five. Filenames and mnemonic names (see below) must be unique.

A name that is qualified may only be written in the Screen Section or Procedure Division of a program. When you refer to a paragraph name that is defined more than once, you need not qualify it more than once within the same section.

Condition Names

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

Mnemonic Names

A *mnemonic name* assigns a word that you choose, such as PRN, to the printer. It is assigned in the Environment Division for reference in ACCEPT or DISPLAY statements.

Data Description Entry

A *data description entry* specifies the characteristics of each field (item) in a data record. Each item must be described as a separate entry in the same order in which the items appear in the record.

Each data description entry consists of a level number, a data name, and a series of independent clauses followed by a period. The general format of a data description entry is:

```
level number data-name|FILLER [REDEFINES-clause]
    [JUSTIFIED-clause] [PICTURE-clause]
    [USAGE-clause] [SYNCHRONIZED-clause]
    [OCCURS-clause] [BLANK-clause]
    [VALUE-clause] [SIGN-clause].
```

When this format is applied to specific items of data, it is limited by the nature of the data being described. The format for 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 brackets.

The clauses may appear in any order except that a REDEFINES clause, if used, should come first. Don't forget that you need a period at the end of the entry.

Group Item Format

```
level number data-name|FILLER [REDEFINES-clause]
    [USAGE-clause] [OCCURS-clause]
    [VALUE-clause] [SIGN-clause].
```

Example:

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

Note: You may write the USAGE-clause at a group level to avoid writing it again and again at subordinate levels.

Elementary Item Format

ALPHANUMERIC ITEM (also called a character-string item)

```
level number data-name|FILLER [REDEFINES-clause]
    [OCCURS-clause] PICTURE IS an-form
    [USAGE IS DISPLAY] [JUSTIFIED-clause]
    [VALUE IS nonnumeric-literal]
    [SYNCHRONIZED-clause].
```

Examples:

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

REPORT ITEM (also called a numeric-edited item)

```
level number data-name|FILLER [REDEFINES-clause]
    [OCCURS-clause] PICTURE IS report-form
    [BLANK WHEN ZERO] [USAGE IS DISPLAY]
    [VALUE IS nonnumeric-literal]
    [SYNCHRONIZED-clause].
```

Example:

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

DECIMAL ITEM

```
level number data-name|FILLER [REDEFINES-clause]
    [OCCURS-clause] PICTURE IS numeric-form
    [SIGN-clause] [USAGE-clause]
    [VALUE IS numeric-literal] [SYNCHRONIZED-clause].
```

Examples:

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

BINARY ITEM

```
level-number data-name|FILLER [REDEFINES-clause]
    [OCCURS-clause] PICTURE IS numeric-form
    USAGE IS COMPUTATIONAL-0|COMP-0|INDEX
    [VALUE IS numeric-literal] [SYNCHRONIZED-clause].
```

Examples:

```
02 SUBSCRIPT COMP-0, VALUE ZERO.
02 YEAR-TO-DATE COMPUTATIONAL-0.
02 INDEX-1 USAGE IS INDEX.
```

Note: A PICTURE or VALUE must not be given for an index data item.

Filenames

A *file* is a collection of data records, such as the TIME-CARD record used in the description of data items. The section of storage on a diskette containing the individual records of a file 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 *filename*. Rules for composition of the filename word are identical to those for data-names. References to a filename appear in the Procedure Division's OPEN, CLOSE, and READ statements, as well as in the Environment Division.

CAUTION

Do not confuse filenames with file ID's as described in Chapter 6.

Literals

A *literal* is a constant that is not identified by a data name in a program. The value of a literal does not change as you compile and/or run a program.

A literal can be either nonnumeric, numeric, or a figurative constant. For example:

```
IF A = 1 MOVE "X" TO MARK.

IF B IS ZERO GO TO CLOSE-OUT.
```

In the first example, the value **1** is a numeric literal, and the value "**X**" is a nonnumeric literal. In the second example, the value **ZERO** is a figurative constant.

Nonnumeric Literals

A *nonnumeric literal* must be enclosed by matching quotation marks (single or double) and may consist of any combination of characters in the ASCII set, except quotation marks. All spaces enclosed in the quotation marks are included as part of the literal. A nonnumeric literal must not be longer than 120 characters.

The following are examples of nonnumeric literals:

```
"ILLEGAL CONTROL CARD"
'CHARACTER-STRING'
"DO'S & DON'T'S"
```

Each character of a nonnumeric literal, following the delimiter, may be any character other than the delimiter. That is, if the delimiters are apostrophes ('literal'), then quotation marks (") may be within the literal, and vice versa.

The length of a nonnumeric literal is the number of characters (including spaces) in the literal. You do not count the delimiters in the length of a literal. The minimum length is one. Two delimiters in a row within a literal are interpreted as a single delimiter.

Nonnumeric literals may be *continued* from one line to the next. When a nonnumeric literal is too long to be contained on one line, the following rules apply to the *continuation line* (the next line of coding):

- A hyphen is placed in column 7 of the continuation line.
- A delimiter is placed after column 11 and preceding the continuation of the literal.
- All spaces at the end of the previous line and any spaces following the delimiter in the continuation line and preceding the next character of the literal are considered to be part of the literal.
- On any continuation line, columns 8-11 should be blank.

Numeric Literals

A *numeric literal* consists of the characters 0 through 9 (optionally preceded by a sign) and the decimal point. It must contain at least one and not more than 18 digits. It may contain only one sign character and only one decimal point. The sign, if present, must appear as the left-most character in the numeric literal. If a numeric literal does not have a sign, it is positive.

A decimal point may appear anywhere within the numeric literal, except as the right-most 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   0.5   -.333
```

If you enter DECIMAL-POINT IS COMMA (common European notation) in the Environment Division, the functions of the period and comma are interchanged. In this case, the value of "π" would be 3,1416 when written as a numeric literal.

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.

One figurative constant is *ZERO*. It may be used in many places in a program as a numeric literal. Other figurative constants are available to provide nonnumeric data; the reserved words representing various characters are as follows:

SPACE	The blank character whose ASCII decimal representation is 32.
LOW-VALUE	The null character whose ASCII decimal representation is 0.
HIGH-VALUE	The character whose ASCII decimal representation is 255.
QUOTE	The quotation mark whose ASCII decimal representation is 34.
ALL literal	One or more instances of the literal, which must be a one-character nonnumeric literal or a figurative constant. In the latter case, ALL is redundant but serves for readability.

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.

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

The following are examples of figurative constants in lines of code:

```
77  COUNTER PIC 99 COMP-0 VALUE IS ZERO.
77  DOTLINE PICTURE X(80) VALUE IS ALL '.'.
```

Arithmetic Expressions

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:

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

When more than one operation is to be processed using a given variable or term, the order of precedence (highest to lowest) is:

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

The highest precedence operation is performed first. Operations of equal precedence are performed from left-to-right.

Parentheses may be used when the normal left-to-right 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 sequence:

1. Compute the product D times E, considered as the intermediate result, R1.

2. Compute the difference C minus R1 as the intermediate result, R2.

3. Divide B by R2, providing intermediate result, R3.

4. Add A plus R3 to get the final result.

Without parentheses, the expression

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

is evaluated as:

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

When you use parentheses, the following punctuation rules should be used:

- A left parenthesis is preceded by one or more spaces.
- 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. For example,

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

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 *conditional*, because the size error condition is data-dependent.

If an arithmetic statement does not include either a GIVING option, ROUNDED option, or SIZE ERROR option, it is called an *imperative statement*.

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".
```

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 processed.

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.

SIZE ERROR Option

If, after decimal-point alignment and any low-order rounding, the value of a calculated result exceeds the largest value that 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 not changed, and the series of imperative statements specified for the condition is processed.

If the SIZE EROR 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 the 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.

ROUNDED Option

If, after decimal-point alignment, the number of places in the fraction of the calculated 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	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

Figure 2. Illustration of Rounding and Truncating

When the low-order integer positions in a resultant data item are represented by the character **P** in the PICTURE clause, rounding or truncation occurs relative to the right-most integer position for which storage is allowed.

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.

CHAPTER 3. DEVELOPING A PROGRAM

Contents

What You Need

To successfully compile COBOL programs on your IBM Personal Computer, you need the following:

- Your COBOL package, which includes:
  - Two 5-1/4 inch master diskettes labeled COBOL and LIBRARY.
    - COBOL contains the following files:
      - COBOL
      - COBOL1.OVR
      - COBOL2.OVR
      - COBOL3.OVR
      - COBOL4.OVR
      - REBUILD.EXE
      - RUNED.BAT
      - RUNEC.BAT
    - LIBRARY contains the following files:
      - COBOL1.LIB
      - COBOL2.LIB
      - COBRUN.EXE
      - LINK.EXE
  - This manual: *IBM Personal Computer COBOL*

- A minimum of 64K bytes of machine-resident memory

- Two diskette drives
- A printer (recommended)
- A display (we recommend that you use an 80-column display), which can be:
  - An IBM Personal Computer Monochrome Display
  - A monitor
  - A TV with an RF modulator
- The IBM Personal Computer Disk Operating System (DOS) diskette and manual
- Three 5-1/4 inch diskettes:
  - Two to make copies of the master diskettes provided
  - One which we will call the *scratch diskette*

Overview of the Compiler

The compiler consists of a main program and four overlays.

The *main program* consists of overlayable and nonoverlayable code. The *nonoverlayable* code is always memory-resident and controls the transition from each overlay to the next.

A compilation is performed in two passes:

1. Pass One—This pass creates an intermediate binary file called COBIBF.TMP. This is a temporary file that is stored on the diskette in drive B during the following steps:
   a. The overlayable portion of the main program compiles the Identification and Environment Divisions.
   b. Overlay 1 (COBOL1.OVR) compiles the Data Division.
   c. Overlay 2 (COBOL2.OVR) compiles the Procedure Division.

2. Pass Two—This pass reads the intermediate file and creates the object code file with the following steps:
   a. Overlay 3 (COBOL3.OVR) reads the intermediate file (COBIBF.TMP) and creates the object code.
   b. Overlay 4 (COBOL4.OVR) allocates file control blocks and checks certain error conditions. Then the intermediate file is deleted.

Program Development Steps

To prepare an IBM COBOL program, you must perform three basic steps:

1. Create a source file with a text editor.
2. Compile the source file with the IBM COBOL compiler.
3. Link the library modules, any subroutines you have written, and the object code file together to create an executable program file.

After you complete these steps, you are ready to run your program.

After you become familiar with IBM COBOL, you will probably want to copy some of your utility files onto your diskettes, such as: EDLIN, CHKDSK, and possible others, as well as your Batch files.

How to Create a COBOL Source File

The *source program* (or *source file*) is a file which consists of lines of ASCII text terminated by carriage-return line-feed. You can create it with EDLIN (your IBM Personal Computer editor). (Refer to *IBM Personal Computer Disk Operating System* for information on how to use EDLIN.)

Coding Rules

Refer to "Coding Rules" in Chapter 2 for instructions on coding your source program.

How to Compile a COBOL Program

Getting Started

We recommend that you back up your COBOL master diskettes as soon as possible by making copies of the COBOL and LIBRARY diskettes. (This is where you use the additional diskettes we said you would need to compile a COBOL program.)

You should use these copies for your day-to-day operations and put your master diskettes away in a safe place.

Now that you have made copies of the COBOL and LIBRARY diskettes, you will need to copy COMMAND.COM from the DOS diskette onto the LIBRARY diskette. You should do this because when the linker is used, it may overwrite COMMAND.COM in storage. (COMMAND.COM is loaded when the system starts with DOS.)

We recommend the following sequence of steps as a general rule when compiling an IBM COBOL program.

1. Format your scratch diskette. See *IBM Personal Computer Disk Operating System (DOS)* for information about formatting.

2. Put your program onto the scratch diskette by either of the following methods:
   - Copy it from the diskette it is currently on.
   - Create a new program by using the line editor (see "EDLIN" in *IBM Personal Computer Disk Operating System (DOS)* for information on EDLIN).

3. Add the COBOL filename extension *.COB* to your program name.

4. Change the default drive to B by entering:

```
B:
```

   In general, the IBM COBOL compiler is set up to have user software in drive B and system software in drive A.

You are now ready to compile your COBOL program.

Note: You may enter compiler commands by using all uppercase, all lowercase, or a combination of both uppercase and lowercase letters.

File Specification

As you perform the steps to compile your COBOL program, you often need to enter filenames. Each filename can be the name of a diskette file or the name of a system device. A file description has the form:

```
device:filename.extension
```

Here the separators are the colon and the period, and the terms mean:

device The name of the system device, which can be a diskette drive, display, printer, or RS232 port. If the device is a diskette, the filename must also be given. If not, the device name itself is the full file description. COBOL recognizes the following symbolic device names:

NUL	Do not create
CON	Display
A: or B:	Diskette drive
PRN or LPT1	Printer
AUX or COM1	RS232

Note: The colon (:) must be used when addressing a drive.

filename The name of the file on diskette. If filename is specified without a device, the current diskette drive is assumed as the device. A maximum of 8 characters is allowed.

extension The extension of the filename given. The following are the extensions that you should use with IBM COBOL:

.COB The source program file
.LST The listing file
.OBJ The object program file

Compilation Steps

We recommend that you follow these steps when you want to compile a COBOL program. (Also see the Sample Session in Appendix D.)

1. If you have not already done so, change the default drive to B.
2. Insert the scratch diskette that contains your program into drive B.
3. Insert your COBOL diskette into drive A.
4. Enter:

```
A:COBOL
```

These four steps load COBOL into the computer. After a short time, the compiler displays a heading and the following prompt:

```
Source filename [.COB]:_
```

Notes:

1. The name shown within the brackets is the *default* filename extension that IBM COBOL uses if you do not choose a filename extension of your own.

2. Although IBM COBOL supplies a default filename extension if you do not supply one, you may override all extensions by explicitly specifying the filename with the new extension.

3. The default diskette drive is the DOS default drive (*B:* in this case). You may override it by including the drive ID as part of the file specification.

Source filename is the name of the file in which you have stored your program. For example, if you respond with *myfile* to the previous prompt, the display shows:

```
Source filename [.COB]:myfile
```

You do not need to enter the *.COB* filename extension, because the compiler automatically looks for *.COB*. After you enter your source filename, you see this prompt:

```
Object filename [MYFILE.OBJ]:_
```

Object filename is the name you want the object (machine-readable) file to have. If you wish to have your object file stored under the name MYFILE.OBJ, simply press the Enter key. If you wish to give the file another name, be sure to add the filename extension *.OBJ*. For our example, assume we have simply pressed the Enter key:

```
Object filename [MYFILE.OBJ]:
```

The last prompt looks like this:

```
Source listing [NUL.LST]:_
```

Source listing is the name you wish to give to the file that contains the compiled source listing. If you do *not* want a listing, press the Enter key. This gives you the default filename NUL.LST, which tells the compiler not to create a source listing file.

> **Note:** As you compile and debug your program, it is not necessary to get a listing every time. Errors are always listed on the display (as well as in the listing). You can get a listing of just the errors by using the Ctrl-Prtsc or Ctrl-Shift keys. The defaults are set up to allow this.

For our example, assume that we do want a listing file, and enter:

```
Source listing [NUL.LST]:myfile
```

> **Note:** The compiler adds the default extension and produces the listing file, MYFILE.LST.

The completed screen looks like this if you use our example filenames:

```
Source filename [.COB]:myfile
Object filename [MYFILE.OBJ]:
Source listing [NUL.LST]:myfile
```

As soon as you enter the last filename, the compiler begins. If the program contains any syntax errors, the compiler displays the error messages on the screen, as well as in the listing file (see "Output Listings and Error Messages" at the end of this chapter).

Note: After you type any of these responses, you may continue the response before you press the Enter key by typing a comma and the answer to what would have been the next prompt, without waiting for that prompt. If you end any response with a semicolon (;), the remaining responses are all assumed to be the default values; processing begins immediately, with no further prompts.

When the compiler finishes, it displays a message with the number of errors it has found. The message looks like this if you send the source listing to a file and no errors are detected:

```
No Errors or Warnings
```

If the compiler detects an error or sends a warning, the error or warning is displayed on the screen along with the following message:

```
1 Error or Warning
```

If the compiler finds errors, you must locate and fix those problems in your source program before linking.

How to Link a COBOL Program

Once you have compiled the source program, the final step before you can run it is to link the program by using the linker. The IBM Personal Computer Linker Version 1.10 provided on your LIBRARY diskette is an upward compatible version of the IBM Personal Computer Linker Version 1.00. A full explanation of this linker is provided in Appendix C, "The Linker (LINK) Program." You must use the linker provided on the LIBRARY diskette to successfully run an IBM COBOL program.

We recommend the following steps when linking your program:

1. Remove COBOL from drive A.

2. Insert your copy of the LIBRARY diskette into drive A.

3. Enter:

```
A:LINK
```

(Be sure that you are using the linker on the COBOL LIBRARY diskette.)

LINK starts the linker and gives the following prompt:

```
Object Modules [.OBJ]:_
```

Enter the name of your *object file*. You do not need to enter the *.OBJ* extension here. For example:

```
Object Modules [.OBJ]:myfile
```

The next prompt is:

```
Run File [MYFILE.EXE]:_
```

Enter the name you want to give to the file containing the executable code for your program. This filename is given the default extension *.EXE* and put onto the default diskette drive (*B*). This filename extension may not be overridden. In our example, we will use the default and just press the Enter key.

The next prompt is:

```
List File [NUL.MAP]:_
```

MAP file is the name you wish to give to the file that contains the linker printed output. If you do not want a map file, press the Enter key. This gives you the default filename NUL.MAP, which tells the linker not to create a map file.

For our example, assume that we do not want a map file, and press the Enter key.

The next prompt is:

```
Libraries [.LIB]:_
```

Libraries refers to the runtime routines needed by IBM COBOL to run your program. All of these routines are included in COBOL1.LIB, COBOL2.LIB, and COBRUN.EXE.

The names of the libraries are automatically supplied by the object file. COBOL1 and COBOL2 are used during the linking, and COBRUN is used when the program runs. See "The Runtime System" later in this chapter for an explanation of the libraries.

The linker assumes that the libraries are in drive A. If they are not in drive A, you must enter a new drive specification. In response to this prompt, you may press the Enter key.

Filenames are specified in the same manner as for the compiler, except that the default extension is always *.OBJ* for files to be read by the linker. Such files are all expected to be in relocatable object format, so they must have been previously compiled (or assembled).

The screen of prompts would look like this if you used our example filenames:

```
B>A:LINK
IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982
Object Modules [.OBJ]:  myfile
Run File [MYFILE.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

After the final entry, the linker starts. The linker may need more memory space to link your program than is resident in your computer. In this case, the linker creates a file called VM.TMP on the diskette in the default drive (the scratch diskette) and displays a message to this effect on the screen. You must not remove this diskette during linking. When finished, the linker erases VM.TMP from the diskette. Any error that occurs during linking produces an error message on the screen. These messages are listed at the end of Appendix C.

When linking is completed, you should have the *Run File* stored on your scratch diskette in drive B. We recommend that you display the diskette directory for the scratch diskette to confirm that the run filename is there. (It will have the *.EXE* filename extension.) Using our example filename, you would see *MYFILE.EXE* listed in the directory.

How to Run a COBOL Program

To run your program, simply enter your run filename, without the .EXE filename extension. For example, enter:

```
myfile
```

You must have the common runtime system resident on the diskette in either the default drive or drive A (see below).

You may want to copy this file to another diskette once you are sure that it does what you intended it to do.

The Runtime System

The relocatable object version of your program produced by the compiler is not 8088 machine code. Instead, it is in the form of a special object language designed specifically for IBM COBOL instructions. The IBM COBOL runtime system runs your program by examining each object language instruction and performing the function required. This includes all processing needed to handle display, printer, and diskette file input and output.

The amount of memory required for a COBOL program to execute equals the amount required to store the data items defined in the Data Division, plus about 500 bytes per file, plus about 12 bytes per line of the Procedure Division, plus up to 32K bytes for the runtime system.

The runtime system consists of a number of machine language subroutines collected into two libraries and a common runtime program:

- COBOL1.LIB–Optional routines
- COBOL2.LIB–Routines necessary to use COBRUN
- COBRUN.EXE–Common runtime library

COBOL1 and COBOL2 are used while linking, and COBRUN is used while running the program.

The routines in COBOL1.LIB are searched by the linker to find and link those additional routines that may be required to perform specific instructions in your source program. The number of routines needed depends on the number of COBOL language features you have used in your main program and subprograms. For example, if the STRING or UNSTRING statements are included in the source program, the linker searches the file COBOL1.LIB to include these optional functions.

In simple terms, COBOL2.LIB prepares your program to use the common runtime library (COBRUN) when you run your program. COBRUN can be resident on either your scratch diskette or the LIBRARY diskette. COBRUN is automatically loaded when you run your program.

First, the system looks for COBRUN on the default drive (B in this case). Then, if COBRUN is not found, it is looked for on Drive A (the drive used for your system software). This search occurs automatically. If COBRUN is not found, then a message is displayed. COBRUN also acts as the executor for running your program.

Runtime Module

Application programs written in IBM Personal Computer COBOL require the COBRUN.EXE runtime module. Information about this runtime module can be obtained by writing to IBM at:

Runtime Module
IBM Personal Computer
P.O. Box 1328-A
Boca Raton, Florida 33432

Optional COBOL Commands

The following commands can also be used to run the compiler. (Be sure you are familiar with the basic command before you attempt to use these.)

You can start IBM COBOL by using the following command line (substituting your filenames for the three files shown):

```
COBOL Source File,Object File,Source List;
```

When you use this command line, the compiler prompts described in the earlier example are not displayed if:

- You specify an entry for all three files

  or

- The command line ends with a semicolon (;)

If you enter an incomplete list and no semicolon, the compiler prompts for the remaining unspecified files. Each prompt displays its default, which you may accept by pressing the Enter key. You may override it by entering another filename or device name. However, if you enter an incomplete list *and* a final semicolon, the unspecified files are defaulted without further prompting.

Examples

The following examples (drive B is the default drive) illustrate the command string used with the command:

```
A:COBOL command-string.
```

Command String	Effect
MYFILE;	Compiles the source from MYFILE.COB. Although there is no comma, this command produces the file MYFILE.OBJ.
MYFILE,;	This does exactly the same thing as the previous example.
MYFILE,,;	Compiles the source from MYFILE.COB and produces the files MYFILE.OBJ and MYFILE.LST.
MYFILE,,CON;	Compiles the source from MYFILE.COB and places the program listing on the display. The object program is MYFILE.OBJ.
MYFILE,MYOBJ,PRN;	Compiles the source from MYFILE.COB, places the listing on the printer, and places the object into MYOBJ.OBJ.
A:MYFILE,MYOBJ,A:;	Compiles MYFILE.COB from diskette A and places the object into MYOBJ.OBJ, and the listing into MYFILE.LST on drive A.

The / Parameter

You can add one or more / (slash) parameters to the command string, which affects the compilation procedure as described below. To add this parameter, type a slash followed by the one-character switch name.

Parameter	Action
/C	The compiler looks for the four overlay files on drive A: • COBOL1.OVR • COBOL2.OVR • COBOL3.OVR • COBOL4.OVR To override drive A, use the /C parameter with the letter of the drive you want. For example, /CB.
/T	The compiler puts its temporary file COBIBF.TMP on drive B unless you use the /T switch. For example, /TA. If you enter /TA, the diskette in drive A must not be write-protected.
/P	Each /P allocates an extra 100 bytes of stack space for the compiler's use. Use /P if a stack overflow occurred during compilation (see Appendix A, "COBOL Error Messages"). Otherwise, /P is not needed.
/D	Suppress object line numbers. If you use this parameter, the resulting Procedure Division code will be about 16% smaller. However, the runtime system will not be able to note the line number at which an error occurs.

Parameter	Action
/Fn	Activate Federal Information Processing Standard (FIPS) flagging, where **n** is a digit from 0 through 4, with the following meanings: 0 Flag everything above low level. 1 Flag everything above low intermediate level. 2 Flag everything above high intermediate level. 3 Flag everything above high level. 4 No flagging. If you do not use this parameter, F4 is the default.

FIPS flagging lets you tell the compiler to put a warning out for each COBOL statement above a chosen standard level, as explained above.

Examples of command strings using the / parameters:

```
COBOL PAYROLL,,/CB;
COBOL PAYROLL,,A:PAYLIST /TA;
COBOL PAYROLL/P/P/P;
COBOL PAYROLL/D/F1;
```

Optional Linker Commands

The following commands can also be used to run the linker. (Be sure that you are familiar with the basic command before you attempt to use these. You should also refer to Appendix C for further information.)

You can also start the linker by using the following command line (substituting your filenames for the filenames shown):

```
LINK objlist,runfile,mapfile,liblist/parms;
```

When you use this command line, the linker prompts that we described earlier are not displayed if:

- You specify an entry for all three files.

  or

- The command line ends with a semicolon (;).

If you enter an incomplete list and no semicolon, the compiler prompts for the remaining unspecified files. Each prompt displays its default, which you may accept by pressing the Enter key, or override by entering another filename or device name. However, if you enter an incomplete list *and* a final semicolon, the unspecified files are defaulted without further prompting.

Examples

The following examples illustrate the command string used with the command:

```
A:LINK command-string.
```

Command String	Effect
MYFILE;	Links MYFILE and puts the sourcefile into MYFILE.EXE. No map file is produced. The COBOL library is automatically supplied.
MYFILE,,;	Same as the first example, except that a listing is produced in MYFILE.MAP.
MYFILE+SUBFILE1+SUBFILE2,,;	Same as the second example, except that SUBFILE1 and SUBFILE2 will be linked with MYFILE.
MYFILE/MAP;	The /MAP parameter causes all public (global) symbols defined in the input modules to be listed in the MAPFILE.
MYFILE/STACK:1024	COBOL uses 512 bytes (default), which is sufficient for most programs. You can specify a larger stack by using this parameter.

Note: The /LINE parameter is not supported for COBOL. All other parameters, other than the /P parameter, are preset by COBOL and should not be overridden. See Appendix C, "The Linker (LINK) Program" for additional instructions.

Automatic Response File

You may optionally use an *automatic response file* to start the linker. This is a file that you create that contains responses to the linker prompts. When the linker is started, it looks in this file for the responses it needs instead of receiving the responses from the keyboard (see Appendix C also). The automatic response file is especially useful when you are running multiple object modules.

To specify this option on the command line, enter:

```
LINK @ARFILE
```

Where *ARFILE* stands for *a*utomatic *r*esponse *f*ile. You must include the drive name for the automatic response file if it is located on a drive other than the default drive.

For example:

```
A:LINK @A:ARFILE
```

Creating an automatic response file for the linker is described in Appendix C, "The Linker (LINK) Program."

Linking a Subprogram

If you have organized your program into a main module and one or more subprogram modules, the linker can combine them into one program. (Refer to Chapter 10, "Interprogram Communication.")

Before linking, compile (or assemble) all modules so that you have a relocatable object version of each. Then start the linker and specify on the object modules prompt each module you want to link.

For example,

```
Object Modules [.OBJ]:myfile+subfile1+subfile2
```

Enter responses for the remainder of the prompts as usual.

Linking With Segmentation

The IBM COBOL segmentation facility lets you run programs that are larger than physical memory. Segmentation is explained further in Chapter 7 under "Segmentation."

IBM COBOL programs that use segmentation cause the linker to create a file for each independent segment of the program. The filenames are formed as follows:

NAME_nn.OVL

Where,

NAME is the PROGRAM-ID which you defined in the Identification Division. If the PROGRAM-ID is less than six characters, it is extended to six characters by adding "_" characters to the end.

_nn is a two-digit hexadecimal number that is the program segment number (decimal) minus 49.

For example, in NAME_32.OVL, the segment number is 99.

Using a Batch File

See the *IBM Personal Computer Disk Operating System (DOS)* for a detailed description of the batch command facility, which you can use to start the compiler.

If you have your program debugged, and you are in the process of making small modifications to it, the following batch file will compile, link, and run your program:

```
1. A:COBOL %1,,;
2. PAUSE...(Insert LIBRARY/LINKER in drive
   A:)
3. A:LINK %1;
```

If you store this file (RUN.BAT) on your scratch diskette, then you can use this batch file simply by entering:

```
RUN myfile
```

An example of a batch file to compile and edit is provided on the COBOL diskette. This file is called RUNEC.BAT, and it uses RUNED.BAT. To use this batch file for the first time, type:

```
A:RUNEC
```

Then follow the instructions.

Compiling a Large Program

You may find that the scratch diskette does not have enough space to hold all of the files produced by the compiler. In this event, we recommend doing one of the following:

- Do not request a listing file (NUL).

  or

- Send the listing to the display (CON) or printer (PRN).

Very large programs can be broken down into a smaller main program and several subprograms. These can be separately compiled and then linked together. (See Chapter 10, "Interprogram Communication.")

The following table shows the interaction between system files, input files, and output files while compiling and linking an IBM COBOL program.

Input Files		System Files	Output Files
SOURCE.COB	Identification Division Environment Division	COBOL.COM	COBIBF.TMP
COPYFILE.EXT	Data Division Procedure Division	COBOL1.OVR COBOL2.OVR	SOURCE.LST
COBIBF.TMP		COBOL3.OVR COBOL4.OVR	SOURCE.OBJ
SOURCE.OBJ		LINK.EXE COBOL1.LIB COBOL2.LIB	SOURCE.MAP SOURCE.EXE NAME_01.OVL
SOURCE.EXE		COBRUN.EXE	Program files (if any)

Figure 3. Files Used while Compiling and Linking

SOURCE.COB	The file that contains your COBOL program.
COPYFILE.EXT	The file which contains the information to be copied when you use the COPY verb. (More than one COPYFILE may exist.)
COBIBF.TMP	The intermediate binary file created when the compiler needs additional memory space.
SOURCE.OBJ	The file that contains your source object code.
SOURCE.EXE	The final, executable, program.
COBOL.COM	The COBOL compiler main program.
COBOL1.OVR	Overlay 1.
COBOL2.OVR	Overlay 2.
COBOL3.OVR	Overlay 3.
COBOL4.OVR	Overlay 4.
LINK.EXE	The IBM Personal Computer Linker.
COBOL1.LIB	The file that contains the optional library routines.
COBOL2.LIB	Prepares your program to use COBRUN.
COBRUN.EXE	The Common Runtime Library.
SOURCE.LST	The file that contains your compilation listing.
SOURCE.MAP	The file that contains the linker listing of your source.
NAME_01.OVL	The independent segments created by the linker when a program uses segmentation.

Output Listings and Error Messages

The listing file output by IBM COBOL is a line-by-line account of the source file with page headings, line numbers, and error messages (if necessary). See the "Sample Listing" at the end of this chapter.

Each source line listed is preceded by a consecutive decimal number. This number is used by the error messages at the end to refer to lines in error. It is also used by the runtime system to indicate what statement caused a runtime error.

Diagnostic error messages may be produced during compilation for syntax errors. Refer to Appendix A, "COBOL Error Messages" for an explanation of the error messages.

Additionally, if the compiler command line is entered incorrectly, or a filename error is made, you will be given an appropriate error message, and then you will be prompted for the entries. Or, you may exit by pressing Ctrl-Break.

Some programming errors cannot be detected when the program is compiled, but they cause the program to end prematurely while you are running it. Each of those errors produces a four-line summary on the screen. (Refer to "Runtime Errors" in Appendix A for a complete list of runtime errors.)

The following message means that an internal error has occurred:

```
?Compiler Error in Phase n at address m.
```

This should not occur in a correct program. If this error occurs, you probably have an error in your program that has caused the compiler to become confused. Make sure the program is correct. If the error still occurs, it could mean that your compiler is defective. In this case, you should perform the following steps in order until you correct the problem:

1. Again, double check your work to make sure that you did not make an error.

2. Make a fresh copy of your backup master COBOL diskette. (Your present copy may have been damaged.)

3. Try recoding your program to get around the problem.

4. Report the problem to your authorized IBM Personal Computer dealer.

Additionally, the stack may have overflowed. This condition is highly unlikely, but if it occurs, the use of the /P parameter may solve it.

COPY Statement

Purpose: You use the COPY statement to logically insert the text of a diskette file (other than the source file) in the source code input to the COBOL compiler. This is useful when large sections of identical code are used in many programs.

Format:

```
COPY text-name.
```

Remarks: *Text-name* is a diskette filename in the format required by DOS.

We recommend, although it is not a requirement, that the COPY statement be the last statement on a line.

Example: Suppose BDEF.COB is a text file containing the following source code:

```
05 B.
   10 B1 PIC X.
   10 B2 PIC X.
```

Then a source file containing

```
05 A.
   10 A1 PIC 9.
COPY BDEF.COB.
05 C.
   10 C1 PIC Z.
```

compiles exactly as if this had been coded:

```
05 A.
   10 A1 PIC 9.
05 B.
   10 B1 PIC X.
   10 B2 PIC X.
05 C.
   10 C1 PIC Z.
```

Sample Listing

On the following pages, we show an example of a listing created by the compiler. This listing includes warnings and error messages.

```
sample  COB                                          09:33:39   07-Jun-82    PAGE   1
Line Number Source Line   IBM Personal Computer COBOL Compiler  Version 1.00

  1          IDENTIFICATION DIVISION.
  2          PROGRAM-ID. SAMPLE LISTING.
  3         *REMARKS.  THIS PROGRAM ILLUSTRATES THE LISTING FILE.
  4          ENVIRONMENT DIVISION.
  5          INPUT-OUTPUT SECTION.
  6          FILE-CONTROL.
  7              SELECT DISK-FILE ASSIGN TO DISK.
  8              SELECT OUT-FILE ASSIGN TO PRINTER.
  9          DATA DIVISION.
 10          FILE SECTION.
 11          FD  DISK-FILE
 12              LABEL RECORDS ARE STANDARD
 13              VALUE OF FILE-ID IS "DEPT.DAT".
 14          01  NAME                PIC X(36).
 15          01  DEPT                PIC XXX.
 16          FD  OUT-FILE.
 17          01  PRINT-LINE          PIC X(80).
```

```
18          WORKING-STORAGE SECTION.
19          77  TOTAL-E             PIC 999 VALUE ZERO.
20          77  TOTAL-D             PIC 99 VALUE ZERO.
21          01  OUT-LINE.
22              02  FILLER          PIC X(10) VALUE SPACES.
23              02  TOTAL-EMPLS     PIC ZZZ9.
24              02  FILLER          PIC X(18) VALUE SPACES.
25              02  TOTAL-DEPTS     PIC Z9.
26              02  FILLER          PIC X(46) VALUE SPACES.
27          01  HEADER.
28              02  FILLER          PIC X(5) VALUE SPACES.
29              02  FILLER          PIC X(15) VALUE "TOTAL EMPLOYEES".
30              02  FILLER          PIC X(5) VALUE SPACES.
31              02  FILLER          PIC X(17) VALUE "TOTAL DEPARTMENTS".
32              02  FILLER          PIC X(39) VALUE SPACES.
```

```
   33          PROCEDURE DIVISION.
   34          BEGIN.
   35              OPEN INPUT DISK-FILE.
   36              READ DISK-FILE AT END GO TO FINISH.
   37              ADD 1 TO TOTAL.
   38              CALL "CNT-DEPTS" USING DEPT, TOTAL-D.
   39              GO TO BEGIN.
   40          FINISH.
   41              MOVE HEADER TO PRINT-LINE.
   42              WRITE PRINT-LINE AFTER 5.
   43              MOVE TOTAL-E TO TOTAL-EMPLS.
   44              MOVE TOTAL-D TO TOTAL-DEPST.
   45              MOVE OUT-LINE TO PRINT-LINE.
   46              WRITE PRINT-LINE AFTER 2.
   47              CLOSE OUT-FILE, DISK-FILE.
   48              STOP RUN.
0016:  /W/ LABEL RECORDS OMITTED ASSUMED FOR PRINTER FILE.
0037:  STATEMENT DELETED DUE TO NON-NUMERIC OPERAND.
0044:  UNRECOGNIZABLE ELEMENT IS IGNORED.  TOTAL-DEPST
0002:  /F/ FILE NEVER OPENED.
0002:  /F/ INCONSISTENT WRITE USAGE.
```

CHAPTER 4. IDENTIFICATION DIVISION

Contents

Purpose

Every COBOL program begins with an *Identification Division*. As the name implies, this division "identifies" the program. It states the program name, its author, its purpose, and other characteristics.

Format

The Identification Division is divided into a *header* and *paragraphs*. The header for this division is required and is always in the following form:

IDENTIFICATION DIVISION.

A paragraph is always in the following form:

paragraph-name. sentence-sequence.

The paragraphs in this division have preassigned names. The header and paragraphs are entered in the following order:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
[AUTHOR. comment-entry ...]
[INSTALLATION. comment-entry ...]
[DATE-WRITTEN. comment-entry ...]
[DATE-COMPILED. comment-entry ...]
[SECURITY. comment-entry ...]
```


Remarks

Only the PROGRAM-ID paragraph is required, and it must be the first paragraph. The contents of the other paragraphs are not important. They serve only as remarks for your use.

The asterisk (*) in column 7 is used anywhere in your program when you wish to include a line of remarks.

> **Note:** The period (.) is required at the end of both the header and the paragraphs in this division.

On the following pages, we present in alphabetic order the header and each of the paragraphs in the Identification Division. We have provided an example of each header and paragraph.

Example

The following example shows a complete Identification Division. This example shows the paragraphs in the order in which they are normally entered.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. payroll.
AUTHOR. S A Schmoekel.
INSTALLATION. Account A.
DATE-WRITTEN. 6-1-82.
DATE-COMPILED. 6-1-82.
SECURITY. FOR INTERNAL USE ONLY.
*REMARKS.Comments must be preceded by
*        an asterisk in column 7.
```

AUTHOR Paragraph

Purpose: This paragraph tells who wrote the program.

Format: AUTHOR. *comments*.

Remarks: This paragraph is optional, serving only as documentary remarks.

Example:

```
AUTHOR. S A Schmoekel.
```

DATE-COMPILED
Paragraph

Purpose: This paragraph tells when you compiled the program.

Format: DATE-COMPILED. *comments*.

Remarks: This paragraph is optional, serving only as documentary remarks.

Example:

```
DATE-COMPILED. 6-1-82.
```

DATE-WRITTEN
Paragraph

Purpose: This paragraph tells when you wrote the program.

Format: DATE-WRITTEN. *comments*.

Remarks: This paragraph is optional, serving only as documentary remarks.

Example:

```
DATE-WRITTEN. 6-1-82.
```

IDENTIFICATION DIVISION
Header

Purpose: This header starts the program.

Format: `IDENTIFICATION DIVISION.`

Remarks: This must be the first line in any COBOL program. The period is required at the end of the header.

Example:

```
IDENTIFICATION DIVISION.
```

INSTALLATION
Paragraph

Purpose: This paragraph tells the use of the program.

Format: `INSTALLATION.` *comments*`.`

Remarks: This paragraph is optional, serving only as documentary remarks.

Example:
```
INSTALLATION. Account A.
```

PROGRAM-ID
Paragraph

Purpose: This paragraph tells the name of the program. The program-name identifies the object program.

Format: `PROGRAM-ID.` *program-name*.

Remarks: This paragraph is required and must be the first paragraph in your program.

Program-name can be any alphanumeric string of characters. Imbedded periods are not allowed. The first character must be a letter. Only the first 6 characters of *program-name* are retained by the compiler.

Example:

```
PROGRAM-ID. payroll.
```

SECURITY
Paragraph

Purpose: This paragraph tells the security level of the program.

Format: SECURITY. *comments*.

Remarks: This paragraph is optional, serving only as documentary remarks.

Example:

```
SECURITY. FOR INTERNAL USE ONLY.
```

CHAPTER 5. ENVIRONMENT DIVISION

Contents

Purpose

The Environment Division specifies the aspects of your COBOL program that depend upon the physical characteristics of your computer. It is required in every program.

The Environment Division always begins with the following header:

ENVIRONMENT DIVISION.

The Environment Division has two possible sections:

- Configuration Section
- Input-Output Section

These sections describe the physical characteristics of your computer and the handling of data files.

Format

The sections of the Environment Division follow this general format:

```
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].
    [SWITCH-n IS comment-ID
          {ON|OFF} IS condition-name]].
[INPUT-OUTPUT SECTION.
  [FILE-CONTROL. file-control-entry ...]
  [I-O-CONTROL.
    [SAME RECORD AREA FOR filename ...] ...]]
```

Remarks

On the following pages, we present in alphabetic order the header and each of the paragraphs in the Environment Division. We have provided an example of each header and paragraph.

Example

The following example shows an Environment Division. This example shows the paragraphs in the order in which they are normally entered.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-PERSONAL-COMPUTER.
OBJECT-COMPUTER. IBM-PERSONAL-COMPUTER.
SPECIAL-NAMES. ASCII IS NATIVE CURRENCY IS "L".
INPUT-OUTPUT SECTION.
FILE CONTROL.
    SELECT timecard ASSIGN TO DISK.
    SELECT vac-sched ASSIGN TO DISK.
I-O CONTROL.
    SAME RECORD AREA FOR timecard, vac-sched.
```

CONFIGURATION SECTION
Header

Purpose: This header introduces the Configuration Section. Here you specify the type of computer you are using, as well as special characteristics and/or names.

Format:

```
CONFIGURATION SECTION.
```

Remarks: The Configuration Section is optional. The header must be entered exactly as shown, including the period.

This section has three possible paragraphs:

SOURCE-COMPUTER
OBJECT-COMPUTER
SPECIAL-NAMES

The contents of the first two paragraphs are treated as comments, except for the clause WITH DEBUGGING MODE, if present. The third paragraph, SPECIAL-NAMES, relates implementor names to names you define and allows changes to default editing characters.

If you enter any of these paragraph names, you must include the CONFIGURATION SECTION header. For more information on these paragraphs, refer to the individual paragraph headings in this chapter.

Example:

```
CONFIGURATION SECTION.
```

ENVIRONMENT DIVISION
Header

Purpose: This header tells you and the compiler that the Environment Division begins here.

Format:
```
ENVIRONMENT DIVISION.
```

Remarks: The header must be entered exactly as shown, including the period.

Example:
```
ENVIRONMENT DIVISION.
```

Purpose: This paragraph defines each file that will be accessed and that has records described in the Data Division's File Section. Here you assign the type of input/output processing allowed for each file.

Format:

```
FILE-CONTROL.
    SELECT file-name ASSIGN TO DISK|PRINTER
    [RESERVE integer AREAS|AREA]
    [FILE STATUS IS data-name-1]
    [ACCESS MODE IS SEQUENTIAL|RANDOM|DYNAMIC]
    [ORGANIZATION IS SEQUENTIAL|LINE SEQUENTIAL|
    RELATIVE|INDEXED]
    [RECORD KEY IS data-name-2]
    [RELATIVE KEY IS data-name-3]
```

Remarks: The SELECT entry must begin to the right of Area A of the source line. All phrases after SELECT *filename* can be in any order.

ACCESS: Both the ACCESS and ORGANIZATION clauses are optional for regular sequential input/output processing. For indexed or relative files, alternate formats are available for this section, and are explained in Chapter 8.

ORGANIZATION: Two formats are available for sequential diskette files. One is the default form which is ORGANIZATION IS SEQUENTIAL. The other is requested by ORGANIZATION IS LINE SEQUENTIAL. Both forms assume the records in the file have variable lengths.

Regular sequential organization has a 2-byte count of the record length, followed by the actual record, for as many records as exist in the file. This type of file is created through a COBOL program.

Line sequential organization has the record, followed by a carriage return/line feed delimiter, for as many records as exist in the file. This type of file is created when you build a data file outside a COBOL program (for example, by using EDLIN).

No COMP-0 or COMP-3 information should be written into a line sequential file, because these data items may contain the same binary codes used for carriage return and line feed. This may cause a problem when you read from the file.

Both organizations pad any remaining space of the last physical block with Ctrl-Z characters, indicating end-of-file. All records are placed in the file with no gaps; they span physical block boundaries.

The RECORD and RELATIVE KEY clauses are explained in Chapter 8.

RESERVE: The RESERVE clause is not functional in IBM COBOL, but it is scanned for correct syntax. One physical block buffer is always allocated to the logical record area assigned to it. This allows logical records to be spanned over physical block boundaries. For files assigned to PRINTER, the logical record area is used as the physical buffer as well.

FILE STATUS: In the FILE STATUS entry, *data-name-1* must refer to a two-character Working-Storage or Linkage Section alphanumeric item. The runtime data management facility places status information in this item after an I/O statement.

The left-hand character of *data-name-1* assumes the following values:

0 Successful completion
1 End-of-file condition
2 Invalid key (only for indexed and relative files)
3 A nonrecoverable I/O error
9 Specific conditions

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	Successful completion
1	0	EOF
3	0	Permanent error
3	4	Disk space full
9	1	File damaged

In an OPEN INPUT or OPEN I-O statement, a File Status of "30" means "File Not Found."

For values of status-right when status-left has a value of "2", see Chapter 8.

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 are performed if the clause is present on the statement that gave rise to the error. But, if there is no 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 "Declaratives and the USE Sentence"), 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 you may base actions upon testing the STATUS item, so normal flow to the next sentence is allowed.

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

These remarks apply to processing of any file, whether organization is sequential, line sequential, indexed, or relative.

Example:

```
FILE-CONTROL.
    SELECT timecard ASSIGN TO DISK
    FILE STATUS IS STATUS-FLAG
    ACCESS MODE IS SEQUENTIAL
    ORGANIZATION IS LINE SEQUENTIAL.
```

INPUT-OUTPUT SECTION
Header

Purpose: This header introduces the Input-Output Section. Here you specify file control and I/O control. In this section, you define the file assignment parameters.

Format:

```
INPUT-OUTPUT SECTION.
```

Remarks: The Input-Output Section is mandatory, unless your program has no data files. The header must be entered exactly as shown, including the period.

This section has two types of paragraphs:

FILE-CONTROL
I-O-CONTROL

In this section, the programmer defines the file assignment parameters, including specification of buffering.

For more information on these paragraphs, refer to each individual paragraph heading in this chapter.

Example:

```
INPUT-OUTPUT SECTION.
```

I-O-CONTROL Paragraph

Purpose: This paragraph allows for the sharing of physical buffer space between two or more files.

Format:

```
I-O-CONTROL.
     SAME RECORD AREA FOR filename...
```

Remarks: The SAME RECORD AREA clause is optional.

The SAME RECORD AREA causes all the named files to share the same logical record area in order to conserve memory space.

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

Example:

```
I-O-CONTROL.
     SAME RECORD AREA FOR timecard, vac-sched
```

OBJECT-COMPUTER
Paragraph

Purpose: This paragraph identifies the object computer.

Format:

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

Remarks: The contents of this paragraph are treated as comments.

Example:

```
OBJECT-COMPUTER. IBM-PERSONAL-COMPUTER.
```

SOURCE-COMPUTER
Paragraph

Purpose: This paragraph identifies the source computer.

Format:

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

Remarks: The contents of this paragraph are treated as comments, except for the WITH DEBUGGING MODE clause, if present.

You can include lines in your program to help trace program errors. These lines (for example, EXHIBIT statements) are preceded by the letter D in column 7. If you have entered WITH DEBUGGING MODE in the SOURCE-COMPUTER paragraph, any such lines with D in column 7 are compiled as part of the program. If you have not entered WITH DEBUGGING MODE, the lines with D in column 7 are ignored and not compiled as part of the program.

Example:

```
SOURCE-COMPUTER. IBM-PERSONAL-COMPUTER.
```

SPECIAL-NAMES Paragraph

Purpose: This paragraph relates machine names to user-defined names. It also changes the meaning of characters, such as decimal points and commas.

Format:

```
SPECIAL-NAMES. [PRINTER IS mnemonic-name]
     [ASCII IS STANDARD-1|NATIVE]
     [CURRENCY SIGN IS literal]
     [DECIMAL-POINT IS COMMA]
     [SWITCH-n IS comment-ID
     {OFF|ON} IS condition-name].
```

Remarks: The following rules apply:

- The PRINTER IS phrase lets you define a name to be used in your DISPLAY statements.
- The entry ASCII IS STANDARD-1|NATIVE specifies that data representation adheres to the American Standard code for Information Interchange. However, this convention is assumed, even if you don't specify it. In IBM COBOL, STANDARD-1 and NATIVE are identical, and they both refer to the character set representation specified in Appendix G, "ASCII Character Codes."
- The CURRENCY SIGN clause lets you select a currency symbol other than the dollar sign (for example, L). The symbol you select must be a nonnumeric literal. It cannot be a quotation mark, a number (0-9), nor any of the characters used in a PICTURE representation.
- The DECIMAL-POINT IS COMMA clause lets you use the European convention of using a comma instead of a decimal point to separate the integer portion from the fraction portion of a number.

- The SWITCH-n clause allows you to set switches during runtime. Eight different switches may be set, SWITCH-1 through SWITCH-8. Any number of them may be coded under the SPECIAL-NAMES paragraph.

  If you use this clause, you are prompted at runtime (before the program begins executing) to enter the switch settings. The switch settings are: a "blank" for OFF, and a "non-blank" for ON.

  Example (the first two lines are the prompt):

```
Enter switch settings (blank=OFF, non-blank=ON):
12345678
XX X
```

  In this case, the user entered **X** (a non-blank character) for 1, 2, and 4. These switches will be ON, and switch 3 will be OFF. Also, any unused switches (5-8) will be OFF (default).

  Note: The reserved word IS is required in the PRINTER, CURRENCY SIGN, DECIMAL-POINT, and SWITCH-n entries.

Example:

```
SPECIAL-NAMES. CURRENCY SIGN IS "L"
        DECIMAL-POINT IS COMMA.
        SWITCH-1 IS TEST-1 ON IS ON1 OFF IS OFF1
        SWITCH-3 IS TEST-3 ON IS ON3 OFF IS OFF3.
    .
    .
    .
PROCEDURE DIVISION.
        IF ON1 GO TO TEST-AREA-1.
        IF ON3 GO TO TEST-AREA-3.
```

CHAPTER 6. DATA DIVISION

Contents

Purpose

The Data Division is where you define your files, records, variables, and tables.

Format

The Data Division begins with the following header:

DATA DIVISION.

This division is subdivided into four sections: File, Working-Storage, Linkage, and Screen.

This division is entered in the following order:

```
DATA DIVISION.
  [FILE SECTION.
    [file description entry
    record description entry ...]...]
  [WORKING-STORAGE SECTION.
    [data item description entry ...]...]
  [LINKAGE SECTION.
    [data item description entry ...]...]
  [SCREEN SECTION.
    [screen-description-entry ...]...]
```

Remarks

In the Data Division, you describe the nature and organization of all of the data used by the program. This includes the record layouts of files, as well as the variables and tables in the program.

All the variables you use in a program must be declared within the Data Division. A variable may occur as an individual item, an item within a table (that is, an array), or an element within a file record. You describe the variables in the four sections of the Data Division.

On the following pages, we present in alphabetic order the header and each of the paragraphs in the Data Division. We have provided examples of the headers, paragraphs, and clauses.

Example

The following is an example of a Data Division. The paragraphs are shown in the order in which they are normally entered.

```
DATA DIVISION.
FILE SECTION.
FD  DISK-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS 'B:C404.DAT'
01  REC-FIELD.
    02  NAME              PIC X(8).
    02  REC-KEY           PIC 999.
FD  PRINT-FILE
    LABEL RECORDS OMITTED.
01  PRINT-LINE            PIC X(80).
WORKING-STORAGE SECTION.
01  START-TIME            PIC 9(8) VALUE 0.
01  FINISH-TIME           PIC 9(8) VALUE 0.
01  PRINT-TIME            PIC XXBXXBXXBXX.
LINKAGE SECTION.
01  PARAMETER-1           PIC 99.
01  PARAMETER-2           PIC 99.
SCREEN SECTION.
01  PASSWORD-SCREEN.
    02  BLANK SCREEN.
    02  LINE 5 COLUMN 18 PIC X(10) TO USER-PASSWORD.
            SECURE BELL AUTO.
    02  LINE 10 COLUMN 10 VALUE
            'ENTER PASSWORD ABOVE'.
    02  LINE 15 COLUMN 18 PIC X(25) TO USER-NAME.
    02  LINE 20 COLUMN 10 VALUE
            'ENTER NAME ABOVE'.
```

File Section

Purpose

In the File Section, you use FD (*F*ile *D*escription) entries to describe blocks, records, and linage.

Format

The File Section of the Data Division begins with the header:

FILE SECTION.

Within the File Section, you have two types of entries:

File description

Record description

Remarks

For each file used by a program, you must have one file description (FD), which is followed by one or more record descriptions. The *file description* defines the structure and layout of the file.

The *record description* defines the structure and layout of a logical record in the file. Each 01 level of a record description implicitly redefines the same area of a previous 01 level. For more information, refer to the individual statements in this chapter.

Example

```
FILE SECTION.
FD  DISK-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS 'B:C404.DAT'
01  REC-FIELD.
    02  NAME       PIC X(8).
    02  REC-KEY    PIC 999.
FD  PRINT-FILE
    LABEL RECORDS OMITTED.
01  PRINT-LINE     PIC X(80).
```

Working Storage Section

Purpose

This section describes records and other data which are not part of external data files but which are developed and processed internally. This is where you declare the variables and tables used in the program.

Format

The second section of the Data Division begins with the header:

WORKING-STORAGE SECTION.

Some of the more commonly used clauses you can use to define the variables and tables include:

OCCURS
PICTURE
REDEFINES
USAGE
VALUE

Remarks

Data description entries in this section may use 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.

Example

```
WORKING-STORAGE SECTION.
01  START-TIME          PIC 9(8) VALUE 0.
01  FINISH-TIME         PIC 9(8) VALUE 0.
01  PRINT-TIME          PIC XXBXXBXXBXX.
```

Linkage Section

Purpose

The Linkage Section allows you to run separately compiled program modules as one single program. 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.

Format

The third section of the Data Division is defined by the header:

```
LINKAGE SECTION.
```

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.

Remarks

The Linkage Section is used in the called subprogram to describe data by name and attribute. However, storage space is not allocated. 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. Refer to Chapter 10, "Interprogram Communication" for further information.

Example

```
LINKAGE SECTION.
01  PARAMETER-1  PIC 99.
01  PARAMETER-2  PIC 99.
```

Screen Section

Purpose

You use the Screen Section of the Data Division to define screen formats for the display. A *screen format* allows you to describe the appearance of the entire display screen to the viewer, rather than the normal scrolling sequence of line-by-line.

The Screen Section is composed of screen data description entries. As in the File and Working-Storage Sections, descriptions may be grouped through the assignment of appropriate level numbers.

The two types of screen items are *elementary* and *group*. *Elementary screen items* define the individual display and/or data entry fields within the screen layout. *Group screen items* are used to name any group of elementary screen items which are accepted or displayed with a single Procedure Division statement.

Format

The format of a *group* screen description entry is:

```
level-number screen-name [AUTO] [SECURE]
    [REQUIRED] [FULL].
```

where *level-number* must be an integer in the range 01 through 49. *Screen-name* must conform to the rules for COBOL names.

The format of an *elementary* screen item is:

```
level-number [screen-name]
    [BLANK SCREEN]
    [LINE NUMBER IS [PLUS] integer-1]
    [COLUMN NUMBER IS [PLUS] integer-2]
    [BLANK LINE]
    [BELL]
    [UNDERLINE]
    [REVERSE-VIDEO]
    [HIGHLIGHT]
    [BLINK]
    [FOREGROUND-COLOR integer-3]
    [BACKGROUND-COLOR integer-4]
    [VALUE IS literal-1]
    [PICTURE|PIC IS picture-string
          {[FROM literal-2|identifier-1]
          [TO identifier-2]|
          [USING identifier-3]}]
    [BLANK WHEN ZERO]
    [JUSTIFIED|JUST RIGHT]
    [AUTO]
    [SECURE]
    [REQUIRED]
    [FULL]
```

where *level-number* and *screen-name* are subject to the same rules as in the group screen description.

Remarks

The following rules apply to the Screen Section:

- If AUTO, SECURE, REQUIRED, or FULL is coded for a group screen item, the effect is as if AUTO, SECURE, REQUIRED, or FULL is coded for every elementary screen item subordinate to that group screen item.

- If PICTURE is coded, then USING with either FROM or TO must be present. A screen item may have both a FROM and TO clause.

- AUTO, SECURE, REQUIRED, FULL, BLANK WHEN ZERO, and JUSTIFIED may be given only if PICTURE is specified.

- The maximum length of an elementary screen item is 80 characters.

The clauses specified with each elementary screen data description can affect data input and data display operations when ACCEPT and DISPLAY statements are processed at runtime.

It is important to note which specifications are functional with the ACCEPT *screen-name* statement, and which are functional with the DISPLAY *screen-name* statement. This distinction and the effects of each specification are as follows:

- BLANK SCREEN causes the entire screen to be erased and the cursor to be placed at the *home position* (line 1, column 1) when the screen is displayed. BLANK SCREEN also returns the screen to its default color if FOREGROUND-COLOR or BACKGROUND-COLOR has been used. BLANK SCREEN has no effect on the ACCEPT *screen-name* statement.

- LINE and COLUMN affect the screen location associated with an elementary screen item. As the Screen Section is processed when the program is compiled, a current cursor position is maintained so that each elementary screen item can be identified with a particular region of the screen.

  When a level 01 screen item is encountered, the current screen position is reset to line 1, column 1. Then, as each elementary screen data description is processed, the current position is adjusted for the size of each definition encountered. Therefore, by default, successively defined fields appear end-to-end in successive areas of the screen.

  The screen position at the start of any elementary screen data description may be changed by means of the LINE and COLUMN specifications. If neither LINE nor COLUMN is coded, the current screen position is not changed. If COLUMN is coded without LINE, the current screen line is not adjusted. If LINE is coded without COLUMN, column 1 is assumed.

  The LINE *integer* or COLUMN *integer* clause without PLUS causes the specified integer to be taken as the line or column at which the current screen item should start. The LINE PLUS *integer* or COLUMN PLUS *integer* clause causes the specified *integer* to be added to the current screen line or column, and the result to be used as the line or column at which the current screen item should start. If LINE (COLUMN) is given without *integer-1* (*integer-2*), LINE PLUS 1 (COLUMN PLUS 1) is assumed.

- BLANK LINE erases the screen from the current cursor position to the end of the current line. The cursor stays in the same position.

Note: The following functions are always processed in the order shown below, regardless of the order in which they are specified.

1. BLANK SCREEN
2. LINE/COLUMN positioning
3. BLANK LINE
4. DISPLAY or ACCEPT of data

- BELL sounds the speaker when the system is ready to accept a field with the ACCEPT *screen-name* statement. BELL has no effect on output fields. That is, a screen item for which BELL is specified is ignored by all DISPLAY statements.

- HIGHLIGHT causes a DISPLAY screen item to appear on the screen in high intensity. On a color display, HIGHLIGHT uses colors 8-15 if 0-7 are coded (see below).

- BLINK causes a DISPLAY screen item to blink on the screen.

- REVERSE-VIDEO causes a DISPLAY screen item to appear on the screen with the background and foreground colors inverted.

- UNDERLINE puts a line under a DISPLAY screen item. UNDERLINE is available only for the IBM Monochrome Display and is ignored for a color display.

- FOREGROUND-COLOR is the color of characters on the display. It is followed by an integer in the range of 0-15 (see notes below). The default is white. FOREGROUND-COLOR is only active for the Color Graphics Monitor Adapter.

- BACKGROUND-COLOR is the color of the background. It is followed by an integer in the range of 0-7 (see notes below). The default is black. BACKGROUND-COLOR is only active for the Color Graphics Monitor Adapter.

Notes:

1. If you have the Color Graphics Monitor Adapter, the following colors are allowed for foreground (*integer-3*):

0	black	8	gray
1	blue	9	light blue
2	green	10	light green
3	cyan	11	light cyan
4	red	12	light red
5	magenta	13	light magenta
6	brown	14	yellow
7	white	15	high intensity white

   For background (*integer-4*), colors 0-7 are available, and values 8-15 cause colors 0-7 to blink.

   You could think of colors 8-15 as light or high-intensity versions of colors 0-7. If you use the same color for both foreground and background, the characters will be invisible. (Individual colors and intensity may vary, depending on your display device.)

2. If you have the IBM Monochrome Display and Printer Adapter and the IBM Monochrome Display, the use of foreground and background colors (other than white and black) will have no effect.

3. Additionally, you may check the current video mode (Black-and-white or Color) through an assembly language subroutine call. An example is provided in Appendix J. "Example Programs With Video Mode."

- VALUE IS *literal-1* specifies the character string which should be displayed on the screen when the screen item being defined is referenced by a DISPLAY *screen-name* statement. *Literal-1* must be bounded by quotes and cannot be a figurative constant. A screen item for which VALUE is specified is ignored by all ACCEPT statements. The PICTURE clause cannot be used with VALUE IS.

- PICTURE specifies the format in which data is to be presented on the screen. It is coded according to the rules for Working-Storage PICTURE clauses described under "PICTURE Clause" in this chapter.

  During a DISPLAY *screen-name* statement, the contents of a FROM or USING field are moved to an implicit temporary item with the specified PICTURE before they are displayed on the screen.

  During an ACCEPT *screen-name* statement, the displayed contents of the field being entered are punctuated so they conform with the given PICTURE format.

- FROM, TO, and USING describe relationships between a screen item and literals and/or fields in either the File, Working-Storage, or Linkage Sections. Identifiers can be qualified but not subscripted.

  When a screen item is displayed, a MOVE occurs from any FROM or USING literal or field to a temporary item defined by the screen item's PICTURE. The resulting contents of the temporary item are then displayed on the screen.

  When the screen item is accepted, the runtime system implicitly moves the accepted data to any TO or USING field specified for the item. A screen item with only a FROM clause has no effect on the operation of the ACCEPT *screen-name* statement.

- BLANK WHEN ZERO causes a screen item to be shown as spaces if its value is zero.
- JUSTIFIED, abbreviated JUST, specifies that operator-entered data or data from a FROM field, USING field, or literal will be aligned with the right boundary of the screen item when it is shown on the screen.
- AUTO specifies that when a field has been filled by operator input, the cursor automatically skips to the next input field, rather than waiting for a terminator character to be entered. If there are no more input fields remaining, the ACCEPT *screen-name* statement ends. AUTO has no effect on DISPLAY *screen-name*. It must be used with a PICTURE clause.
- SECURE suppresses the echoing of input characters. Instead, an asterisk is displayed for each data character accepted with the ACCEPT *screen-name* statement. SECURE has no effect on DISPLAY *screen-name*. It must be used with a PICTURE clause.
- REQUIRED specifies that some entry *must* be made before a field terminator character will be accepted.
- FULL specifies that any field terminator character will be ignored unless every data input position has been filled.

Example

```
SCREEN SECTION.
01 PASSWORD-SCREEN.
    02 BLANK SCREEN.
    02 LINE 5 COLUMN 18  PIC X(10) TO USER-PASSWORD
           SECURE BELL AUTO.
    02 LINE 10 COLUMN 10 VALUE
           'ENTER PASSWORD ABOVE'.
    02 LINE 15 COLUMN 18 PIC X(25) TO USER-NAME.
    02 LINE 20 COLUMN 10 VALUE
           'ENTER NAME ABOVE'.
PROCEDURE DIVISION.
SIGN-ON.
*GIVE THE PROMPTS
    DISPLAY PASSWORD-SCREEN.
*ACCEPT PASSWORD AND NAME
    ACCEPT PASSWORD-SCREEN.
```

Data Division Limitations

There is a limitation on the number of items in the Working-Storage, Linkage, and File Sections of the Data Division. The sum:

$$\frac{W}{4096} + F + L$$

must be less than or equal to 14, where

W The size of Working-Storage in bytes. (W/4K (4096) is rounded up.)

F The number of files described in the File Section.

L The number of level 01 or 77 entries in the Linkage Section.

Note: The maximum number of files which may be open in the same run unit (main program linked together with an arbitrary number of subprograms) is 14.

BLANK WHEN ZERO Clause

Purpose: 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.

Format:

```
BLANK WHEN ZERO
```

Remarks: When this clause is used with a numeric PICTURE, the field is considered a report field.

Example:

```
02  AMOUNT  PIC $$$.99  BLANK WHEN ZERO.
```

BLOCK Clause

Purpose: You use the BLOCK clause to specify characteristics of physical records in relation to the concept of logical records.

Format:

```
BLOCK CONTAINS integer-2 CHARACTERS
      integer-2 RECORDS
```

Remarks: The BLOCK clause has no effect in IBM COBOL, but it is examined for correct syntax. It is normally applicable to tape files, which are not supported by IBM COBOL.

Files assigned to the printer must not have a BLOCK clause in the associated FD entry.

If used as commentary, the size of a physical block 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.

CODE-SET Clause

Purpose: The CODE-SET clause is used to specify files not on a diskette.

Format:

```
CODE-SET IS alphabet-name
```

Remarks: The CODE-SET clause serves only the purpose of documentation in IBM COBOL, reflecting the fact that both internal and external data are represented in ASCII code. 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 USAGE IS DISPLAY.

DATA RECORD(S)
Clause

Purpose: The optional DATA RECORDS clause identifies the records in the file by name.

Format:

```
DATA RECORD IS|RECORDS ARE
    data-name-1 [data-name-2...]
```

Remarks: This clause is documentary only in IBM COBOL. 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, following the appropriate file description (FD) in the File Section.

When multiple data records occur, each reference to *data-name-1* implicitly redefines the area used by *data-name-1*.

Example:

```
DATA RECORDS ARE NEW-PATIENT, OLD-PATIENT.
```

FD Entry (Sequential I/O Only)

Purpose: 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).

Format:

```
FD filename LABEL-clause
    [VALUE-OF-FILE-ID-clause]
    [DATA-RECORD(S)-clause] [BLOCK-clause]
    [RECORD-clause] [CODE-SET-clause]
    [LINAGE clause].
```

Remarks: After FD *filename*, the order of the clauses does not matter.

Example:

```
FILE SECTION.
FD DISK-FILE
    LABEL RECORDS STANDARD
    VALUE OF FILE-ID "A:MEMBERS.DAT"
    LINAGE IS 5 LINES
             TOP 5 BOTTOM 5.
```

JUSTIFIED
Clause

Purpose: The JUSTIFIED clause signifies that values are stored in a right-to-left fashion. This results 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.

Format:
```
JUSTIFIED RIGHT
```

Remarks: This clause is only applicable to unedited alphanumeric (character string) items. The JUSTIFIED clause is effective only when the associated field is employed as the receiving field in a MOVE statement.

You may use the word JUST as an abbreviation of JUSTIFIED.

Example:
```
02  NAME    PIC X(20) JUST.
```

LABEL Clause

Purpose: This clause tells the system whether you have assigned labels to your files.

Format:

```
LABEL RECORD|RECORDS [IS|ARE]
      OMITTED|STANDARD
```

Remarks: The OMITTED option specifies that no labels exist for the file. You must specify OMITTED for files assigned to the printer.

The STANDARD option specifies that labels exist for the file and that the labels conform to system specifications. You must specify STANDARD for files assigned to a diskette.

Example:

```
LABEL RECORDS ARE OMITTED.
```

LINAGE
Clause

Purpose: For a file assigned to the printer, the LINAGE clause provides a means of specifying the size of the printable portion of a page, called the *page body*. The number of lines in the page body is specified along with, optionally, the size of the top and bottom margins and the line number within the page body at which a footing area begins.

Format:

```
LINAGE IS data-name-1|integer-1 LINES
    [WITH FOOTING AT data-name-2|integer-2]
    [LINES AT TOP data-name-3|integer-3]
    [LINES AT BOTTOM data-name-4|integer-4]
```

Remarks: All *data-names* must refer to unsigned numeric integer data items. *Integer-1* must be greater than zero, and *integer-2* must not be greater than *integer-1*.

The total page size is the sum of the values in each phrase except for FOOTING. If TOP or BOTTOM margins are not specified, their size is assumed to be zero. The *footing area* comprises that part of the page body between the line indicated by the FOOTING value, and the last line of the page body, inclusively.

The values in each phrase at the time the file is opened (by an OPEN OUTPUT statement) specify the number of lines that comprise each of the sections of the first logical page.

When a WRITE statement with the ADVANCING PAGE phrase is performed or a page overflow condition occurs (see the WRITE statement), the values in each phrase, at that time, specify the number of lines in each section of the next logical page.

A LINAGE-COUNTER is created by the presence of a LINAGE clause. The value in the LINAGE-COUNTER at any given time represents the line number at which the printer is positioned within the current page body. LINAGE-COUNTER may be referenced, but the counter may not be modified by Procedure Division statements. The counter is automatically modified by a WRITE statement, according to the following rules:

- When the ADVANCING PAGE phrase of the WRITE statement is specified or a page overflow condition occurs, the LINAGE COUNTER is reset to one.
- When the ADVANCING *identifier* or *integer* phrase is specified, LINAGE-COUNTER is incremented by the ADVANCING value.
- When the ADVANCING phrase is not specified, LINAGE-COUNTER is incremented by one.

See the description of the "WRITE Statement" in Chapter 8 for additional information about the effects of LINAGE specifications.

Example:

```
LINAGE IS 60 LINES TOP 5 BOTTOM 5.
```

OCCURS Clause

Purpose: You use the OCCURS clause to define related sets of repeated data, such as tables, lists, and arrays. You may specify the number of times, up to a maximum of 1023, that a data item with the same format is repeated.

Format:

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

Remarks: The OCCURS clause must not be used in any data description entry having a level number 01 or 77. 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.

Since the OCCURS clause can only be used at subordinate levels within a data record, the maximum size of a table is limited by the rules for the size of a group item. See "Group Item" in Chapter 2.

Subscripting lets you refer 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, the item must be subscripted or indexed whenever it is used. See Chapter 9, "Table Handling by the Indexing Method" for explanations on subscripting, 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. The subscript is enclosed in parentheses and appears after the delimiting space in the name of the element. A subscript must be a decimal or binary item. (We strongly recommend the latter, 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, the subscripts are written in the order of successively less inclusive dimensions of the data organization. Multiple subscripts are separated by commas; for example:

```
ITEM (I, J)
```

OCCURS
Clause

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

- A subscript
- The defining name of a data description entry
- Data-name-2 in a REDEFINES clause
- A qualifier

Example:

Example 1

```
01 ARRAY-1.
     03 ELEMENT, OCCURS 3, PICTURE 9(4).
```

In Example 1, storage is allocated as shown below.

ELEMENT (1) ELEMENT (2) ELEMENT (3)	ARRAY, consisting of twelve characters; each item has 4 digits.

Example 2

```
*initializing an array
 01  ARRAY-2.
     02  FILLER      PIC X   VALUE1.
     02  FILLER      PIC X   VALUE2.
     02  FILLER      PIC X   VALUE3.
 01  ARRAY-REFERENCED OCCURS 3 TIMES
         PIC X REDEFINES ARRAY-2.
```

Purpose: The PICTURE clause specifies a detailed description of an elementary level data item. It may include specification of special report editing.

Format: `PICTURE IS an-form|numeric-form|report-form`

Remarks: The reserved word PICTURE may be abbreviated PIC. There are three possible types of pictures:

An-Form: 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 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 following combinations:

- A and 9
- X and 9
- X and A

in any order is considered as if every **9**, **A**, or **X** character is an **X**. All strings containing an **X** or **A** are stored as if each character is an **X**.

The characters **B**, **0**, and / may be used to insert blanks, zeros, or slashes in the item, respectively. This is then called an *alphanumeric-edited item*. (Alphanumeric-edited items are discussed under the "Report-Form" descriptions.)

If the string has only **A**s and **B**s, it is considered *alphabetic*. If it has only **9**s, **P**s, **S**s, or a **V**, it is *numeric* (see below). Strings may be tested in a CLASS test in an IF statement to determine whether they are alphabetic or numeric.

Numeric-Form: 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 tells the compiler 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.

S The optional character **S** indicates that the item has an operational sign. It must be the first character of the PICTURE. See "SIGN Clause" in this chapter.

P The character **P** indicates an assumed decimal scaling position. It specifies 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 left-most 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 left-most PICTURE characters and to the right of the **P**s if the **P**s are right-most PICTURE characters), the assumed decimal point symbol **V** is redundant as either the left-most or right-most character within such a PICTURE description.

For example, if 4 is moved to "PICTURE 9PP", then the number 400 is in the field. If .4 is moved to "PICTURE VPP9", then .004 moves to the field.

Report-Form: 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 the period (.) is used.

Z,*

The characters **Z** and * are called *replacement characters*. Each one represents a digit position. While the program is running, leading zeros to be placed in positions defined by **Z** or * are suppressed, becoming blank or *, respectively.

Zero suppression ends at the first decimal point (.), **V**, or **9** character in the PICTURE, or at a nonzero 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.

If the value is nonzero, all **Z** or * characters to the right of the decimal point are treated as **9**s. If the display data is zero, a **Z** to the right of the decimal point also suppresses the decimal point.

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. They 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 appear instead. The **CR**, **DB**, +, and – are mutually exclusive (+ and – are described further on).

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 a continuous string of at least two characters of either $, +, or –, optionally interrupted by one or more insertion commas and/or decimal points. For example:

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

A floating string containing N + 1 occurrences of $, +, or – defines N digit positions. When you move 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 – immediately to the left of the most significant nonzero digit, in one of the positions indicated by $, +, or – in the PICTURE. (Moving a positive number into a field of – is an exception; see next page.) Blanks are placed in all character positions to the left of the $, +, 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 – in the right-most position of the floating string, and nonsignificant 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 are indicated by the PICTURE character **9** for the display of nonzero data.

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 (that is, as a fixed sign control character) or in a floating string. As a fixed sign control character, the + or – must appear as either the first or last symbol in the PICTURE.

The plus sign (+) indicates that the sign of the item (either a plus or minus) is placed in the character position, depending on the algebraic sign of the numeric value.

The minus sign (–) indicates that either a blank or a 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** (zero) 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:

- If you use one type of floating string, you cannot use any other floating string.
- You must have at least one digit position character.
- If you use a floating sign string or fixed plus or minus insertion character, you cannot use any other of the sign control insertion characters (+, –, **CR**, **DB**).

- 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:
  - Only one type of digit position character may appear. That means that the **Z**, *, and **9**, and the floating-string digit position characters $, +, and – are all mutually exclusive.
  - If one of the numeric character positions to the right of a decimal point is represented by +, –, $, or **Z**, then all the numeric character positions in the PICTURE must be represented by the same character.
- The PICTURE character **9** can never appear to the left of a floating string or replacement character.

Additional notes on the PICTURE clause:

- A PICTURE clause must only be used at the elementary level.
- An integer enclosed in parentheses and following an **X**, **9**, $, **Z**, **P**, *, **B**, –, or + indicates the number of consecutive times the PICTURE character occurs.
- Characters **V** and **P** are not counted in the space allocation of a data item. **CR** and **DB** occupy two character positions each.

- A maximum of 30 character positions is allowed in a PICTURE character string. For example, both PIC 99999 and PICTURE X(89) consist of five characters.

- A PICTURE must contain at least one of the characters **A**, **Z**, *, **X**, **9**, or at least two consecutive appearances of the +, –, or $ characters.

- The characters ., **S**, **V**, **CR**, and **DB** can appear only once in a PICTURE.

- When you specify DECIMAL-POINT IS COMMA, the explanations for period and comma apply to comma and period, respectively.

- The PICTURE clause cannot be used with the VALUE IS clause in the Screen Section (see "Screen Section" in this chapter).

Example: 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

Figure 4. Examples of Editing Data with PICTURE

Purpose: 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.

Format:

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

Remarks: *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 to the size of the smallest record. The sizes are given as character positions required to store the logical records.

REDEFINES
Clause

Purpose: 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.

Format:

```
data-name-1 REDEFINES data-name-2
```

Remarks: The optional 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 performed 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.

The following rules must be obeyed in order to establish a proper redefinition:

1. The level of the definition must equal the level of the redefinition and must not be level 88.

2. There can be no lower level numbers between the definition and the redefinition.

3. The length of the value in *data-name-1* multiplied by the number of times that *data-name-1* occurs cannot be greater than the length of the value in *data-name-2*, unless *data-name-1* is a level 01 data item. (This is permitted only outside the File Section.)

4. *Data-name-1*, and entries subordinate to *data-name-1*, must not contain any VALUE clauses, except in level 88.

In the File Section, all level 01 entries that are subordinate to a given FD entry implicitly represent redefinitions of the same area.

Example:

```
02  RECORD-ID
    03  NAME      PIC X(15).
    03  FILLER    PIC X.
    03  NUMBER    PIC 99.
02  NEW-RECORD REDEFINES RECORD-ID  PIC X(18).
```

SIGN Clause

Purpose: This clause allows you to specify the manner of representing an operational sign.

Format:
```
[SIGN IS] TRAILING|LEADING
    [SEPARATE CHARACTER]
```

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

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

SIGN Clause	Sign Representation
TRAILING (default)	Embedded in right-most byte
LEADING	Embedded in left-most byte
TRAILING SEPARATE	Stored in separate right-most byte
LEADING SEPARATE	Stored in separate left-most byte

Figure 5. Effects of SIGN Clause

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 one character. The entries to which the SIGN clause apply must be implicitly or explicitly described as USAGE IS DISPLAY.

> **Note:** When you specify the CODE-SET clause for a file, you must describe all signed numeric data for that file with the SIGN IS SEPARATE clause.

Any sign specification that does not specify SEPARATE (for example, PIC S9(3)) produces output with an alpha-character in the bit that contains *both the sign and the number*. Refer to Figure 6.

Example:

```
02  TOTAL  PIC 9(5) SIGN IS LEADING SEPARATE.
```

Number	Positive	Negative
1	A	J
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	G	P
8	H	Q
9	I	R
0	{	}

Figure 6. Alpha-characters in Signed Bit

SYNCHRONIZED Clause

Purpose: The SYNCHRONIZED clause allocates space for data in an efficient manner, with respect to the computer memory.

Format:

```
SYNC|SYNCHRONIZED [LEFT|RIGHT]
```

Remarks: The SYNCHRONIZED specification is treated as commentary only. In IBM COBOL, word alignment is always on an even-byte boundary, which allows for efficient use of the 8088 processor.

Example:

```
02  TOTAL  PIC 999 SYNC.
```

USAGE Clause

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

Format:

```
USAGE IS DISPLAY|COMPUTATIONAL|
    COMPUTATIONAL-0|COMPUTATIONAL-3|INDEX
```

Remarks: DISPLAY defines the representation of an external decimal (ASCII) data item, where 1 byte per character is used.

COMPUTATIONAL, which may be abbreviated COMP, is identical to DISPLAY.

COMPUTATIONAL-0, which may be abbreviated COMP-0, defines an integer binary field.

COMPUTATIONAL-3, which may be abbreviated COMP-3, defines a packed decimal (sometimes called an internal decimal) field.

INDEX is explained in Chapter 9, "Table Handling by the Indexing Method."

The USAGE clause may be written at any level. 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.

If USAGE is not specified, the item is assumed to be in DISPLAY mode.

Example:

```
01  AMT    PIC 99  USAGE IS COMPUTATIONAL-3.
01  SUM    PIC 999 COMP-0 VALUE 0.
```

VALUE Clause

Purpose: The VALUE clause specifies the initial value of working storage items.

Format: `VALUE IS literal`

Remarks: The VALUE clause must not be written in a data description entry that also has an OCCURS or REDEFINES clause, nor in an entry that is subordinate to an entry containing an OCCURS or REDEFINES clause. Furthermore, you cannot use the VALUE clause 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 the 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. For edited items, values must be specified as nonnumeric literals, and must be presented in edited form. The *literal* may be a figurative constant.

If you do not specify an initial value for an item, you should not assume that the item in Working-Storage is initialized.

VALUE Clause

The VALUE clause may be specified at the group level, in the form of a correctly sized nonnumeric literal or a figurative constant. In these cases, the VALUE clause cannot be stated at the subordinate levels within 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). For more information, see "Level 88 Condition Names" in this chapter.

Example:

```
01  AVERAGE-PAY  PIC 9(5)  VALUE 15000.
77  TOTAL        PIC 99    VALUE ZERO.
```

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. If the elementary item has such a value, the named condition is true; otherwise, it is false. The format of a level 88 item's value clause is:

```
VALUE IS literal-1 [literal-2 ...]
VALUES ARE literal-1 THRU literal-2
```

The type of *literal* in a condition name entry must be consistent with the data type of the conditional variable.

One or more level 88 entries must be immediately preceded by the elementary item (which may be FILLER) to which it pertains. 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.

In the following example, PAYROLL-PERIOD is the conditional variable. The PICTURE associated with it limits the value of the level 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 nonnumeric literals.

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

VALUE OF FILE-ID Clause

Purpose: The VALUE OF FILE-ID clause appears in any FD entry for a disk-assigned file, and specifies a filename as it is known to the Disk Operating System (DOS).

Format: `VALUE OF FILE-ID IS` *`data-name|literal`*

Remarks: *Data-name-1* and *literal* refer to a filename. They must be entered as a data name or as a nonnumeric literal of at most 14 characters. The filename is specified according to the rules for filenames for DOS. Remember that a DOS filename is in the form:

[1 character:] 1-8 characters[.1-3 characters]

It must not contain any embedded space characters.

If a *data-name* is specified, the filename it contains may be as many characters as desired, but it must end with a space character.

A reminder: if a file is assigned to the printer, it is unlabeled, and the VALUE clause must *not* be included in the associated FD. If you have entered ASSIGN TO DISK for a file, you must include both LABEL RECORDS STANDARD and VALUE OF FILE-ID clauses in the associated FD.

VALUE OF FILE-ID
Clause

Example:
```
VALUE OF FILE-ID "A:MASTER.ASM"
VALUE OF FILE-ID newname
```

In the second example, the data item *newname* must contain a name followed by a space. For example,

```
01 NEWNAME.
    03 DRIVE  PICTURE XX VALUE "A:".
    03 FILLER PICTURE X(12) VALUE "CTLFILE.DAT ".
```

CHAPTER 7. PROCEDURE DIVISION

Contents

Purpose

In the Procedure Division, you tell your IBM Personal Computer what you want it to do and how to do it. The information you supplied in the Identification, Environment, and Data Divisions is now used to process input data and produce output data.

Format

The Procedure portion of a program must begin with the header:

PROCEDURE DIVISION.

The rest of the Procedure Division may be subdivided in three possible ways:

1. It consists only of paragraphs.

2. It consists of a number of sections (each section subdivided into one or more paragraphs).

3. It consists of a Declaratives portion and a series of sections (each section subdivided into one or more paragraphs).

The general format of the Procedure Division is as follows:

```
PROCEDURE DIVISION [USING [identifier-1]...].
  [DECLARATIVES.
    [section-name SECTION. USE sentence.
    [paragraph-name. [sentence]...]...]...
  END DECLARATIVES.]
    [[section-name SECTION. [segment number]]
    [paragraph-name. [sentence]...]...]...
```

The statement formats available within the Procedure Division may be divided into five categories:

- I/O statements
- Data movement statements
- Arithmetic statements
- Sequence control statements
- Miscellaneous statements

Remarks

The Procedure Division can be logically thought of as a series of modules, each with a specific task to perform. Examples could be:

- Initialization
- File processing
- Data manipulation
- Subroutines
- Error handling

These modules can be constructed by using either only paragraphs or sections and paragraphs, as necessary. Sections are used to divide large COBOL programs into segments that will fit in memory.

The declaratives portion of the Procedure Division is designed to handle I/O errors. It is discussed in more detail separately in this chapter. Refer to "Declaratives and the USE Sentence" in this chapter for a complete discussion.

Advanced topics (such as indexing of tables, file accessing, interprogram communication and declaratives) are discussed in subsequent chapters.

On the following pages, we present in alphabetic order the header and each of the paragraphs in the Procedure Division. We have provided examples of the headers, paragraphs, and clauses.

Example

The following is an example of a Procedure Division. This example shows the paragraphs in an order in which they are commonly entered.

```
 PROCEDURE DIVISION.
 DECLARATIVES.
*In case there is an error
 MISTAKE SECTION.
         USE AFTER EXCEPTION DISK-FILE.
 DISPLAY-ERROR.
     EXHIBIT STATUS-FLAG.
     DISPLAY "I-O ERROR OCCURRED".
     STOP RUN
 END DECLARATIVES.
*Open the files, initialize counter
 START-UP.
 OPEN INPUT DISK-FILE OUTPUT PRINT-FILE.
 MOVE ZERO TO COUNTER.
*Read data from and write data to file
 READ-FILE.
     READ DISK-FILE INTO PRINT-LINE
          AT END GO TO EOJ.
     ADD ONE TO COUNTER.
 PRINT-FILE.
     WRITE PRINT-LINE AFTER 2.
     GO TO READ-FILE.
*Write counter message and stop
 EOJ.
      MOVE COUNTER-LINE TO PRINT-LINE.
      WRITE PRINT-LINE AFTER 5.
      STOP RUN.
```

Declaratives and the USE Sentence

The *declaratives* region provides a method of including tasks that are processed not as part of the sequential coding written by the programmer, but rather when a condition occurs that cannot normally be tested by the program.

Although the system automatically creates and checks standard labels, and processes error recovery routines in the case of input/output errors, you may add other tasks to your COBOL program.

Since these tasks are processed only when 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 tasks 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.

Upon exiting the error procedure, control is passed back to the next sentence after the one that produced the error.

The key words DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a period. Segment numbers, if used, must be 0-49.

```
PROCEDURE DIVISION.
DECLARATIVES.
{section-name SECTION [segment-number]. USE sentence.
    [paragraph-name. [sentence]...] ...} ...
END DECLARATIVES.
```

The USE sentence defines how the associated section of coding is used. A USE sentence must immediately follow a section header in the Declaratives Section of the Procedure Division. It must be followed by a period, which is then 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 processed; rather, it defines the conditions necessary to process the USE procedure. The general format of the USE sentence is:

```
USE AFTER STANDARD EXCEPTION|ERROR PROCEDURE
    ON [filename... |
    INPUT|OUTPUT|I-O|EXTEND].
```

The words EXCEPTION and ERROR may be used interchangeably. The associated Declaratives Section is processed (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 filename may not be associated with more than one Declaratives Section.

Within a Declaratives Section, you cannot have a nondeclarative procedure. Conversely, in the nondeclarative section, you cannot have a reference to procedure names that appear in the Declaratives Section, except that PERFORM statements may refer to a USE statement and its procedures. In a range specification (see "PERFORM Statement"), if one procedure name is in a Declaratives Section, then the other must be in the same Declaratives Section.

The compiler inserts an exit from a Declaratives Section that comes after the last statement in the section. All logical program paths within the section must lead to that exit point.

Example:

```
PROCEDURE DIVISION.
DECLARATIVES.
ERRS-1-SECTION.
    USE AFTER ERROR PROCEDURE ON FILE1.
ERRS-1-PARAGRAPH.
    ADD 1 TO ERRS-1-STATUS.
ERRS-2-SECTION.
    USE AFTER ERROR PROCEDURE ON FILE2.
ERRS-2-PARAGRAPH.
    ADD 2 TO ERRS-2-STATUS.
END DECLARATIVES.
```

Segmentation

Program segmentation lets you run IBM Personal Computer COBOL programs that are larger than physical memory. When you use segmentation (that is, when any section header in the program contains a segment number), the entire Procedure Division must be written in sections. Each section is assigned a segment number by a section header of the form:

```
section-name SECTION [segment number].
```

where *segment number* must be an integer with a value in the range from 0 through 99. If *segment number* is omitted, it is assumed to be 0. Declaratives sections must have *segment numbers* less than 50. All sections which have the same *segment number* constitute a single program segment and must occur together in the source program. All segments with numbers less than 50 must occur together at the beginning of the Procedure Division.

Segments with numbers 0 through 49 are called *fixed segments* and are always resident in memory while the program is running. Segments with numbers greater than 49 are called *independent segments*. Each independent segment is treated as a program overlay; only one is in memory at any given time.

An independent segment is in its initial state when control is passed to it for the first time while the program is running. It is also in its initial state when control is passed to that section (implicitly or explicitly) from another segment with a different segment number. Specifically, an independent segment is reached implicitly and is in its initial state when it is reached by "falling through" the end of a fixed or different independent segment.

Segmentation causes the following restrictions on the use of the ALTER and PERFORM statements:

- A GO TO statement in an independent segment must not be referred to by an ALTER statement in any other segment.

- A PERFORM statement in a fixed segment may have within its range only:
  - Sections and/or paragraphs wholly contained within fixed segments
    or
  - Sections and/or paragraphs wholly contained in a single independent segment

- A PERFORM statement in an independent segment may have within its range only:
  - Sections and/or paragraphs wholly contained within fixed segments
    or
  - Sections and/or paragraphs wholly contained within the same independent segment as the PERFORM statement

ACCEPT Statement

Purpose: You use the ACCEPT statement in a program to obtain low-volume input when the program is running.

Format: Four formats are available:

Format 1:

```
ACCEPT identifier-1 FROM
    DATE|DAY|TIME|ESCAPE KEY
```

Format 2:

```
ACCEPT identifier-2
```

Format 3:

```
ACCEPT position-spec identifier-3
    [WITH
    {SPACE-FILL|ZERO-FILL|LEFT-JUSTIFY|
    RIGHT-JUSTIFY|TRAILING-SIGN|PROMPT
    UPDATE|LENGTH-CHECK|AUTO-SKIP|BEEP
    NO-ECHO|EMPTY-CHECK} ...]
```

Format 4:

```
ACCEPT screen name
    [ON ESCAPE imperative-statement]
```

Remarks: The function of each form of the ACCEPT statement is to acquire data from a source external to the program and place it in a specified receiving field or set of receiving fields. The forms differ primarily in the data source with which they are designed to interface.

ACCEPT Statement

Format 1 ACCEPT Statement

```
ACCEPT identifier-1 FROM
    DATE|DAY|TIME|ESCAPE KEY
```

Format 1 ACCEPT obtains date or time information from DOS.

Any of several standard values may be obtained by use of the Format 1 ACCEPT statement.

The formats of the standard values 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. Example: January 30, 1981, is 81030.

TIME: An eight-digit value of the form HHMMSSFF, where HH is the hour (from 00 to 23), MM is the minutes (from 00 to 59), SS is the seconds (from 00 to 59), and FF represents hundredths of a second (from 00 to 99). Example: 9:30:53.72 is 09305372.

ESCAPE KEY: A two-digit code generated by the key that ended the most recently processed Format 3 or Format 4 ACCEPT statement. *Identifier-1* can be examined to determine exactly which key was typed. You can stop inputting by pressing any of the following keys, which causes the ESCAPE KEY value to be set as shown:

Terminator Character	ESCAPE KEY Value
Backtab key (terminates only Format 3 ACCEPT statements)	99
Esc key	01
Field-terminator (Enter key, Tab key)	00
Function key (01-10)	02-11

Figure 7. ESCAPE KEY Values When ACCEPT Ends

If input is stopped as a result of using the AUTO-SKIP option (that is, no terminator key is struck), the ESCAPE KEY value is set to 00.

Identifier-1 should be an unsigned numeric integer whose length agrees with the content of the system-defined data item. If not, the standard rules for a MOVE govern storage of the source value in the receiving item (*identifier-1*).

Example:

```
ACCEPT TODAY FROM DATE.
ACCEPT STATUS-ITEM FROM ESCAPE KEY.
```

ACCEPT Statement

Format 2 ACCEPT Statement

```
ACCEPT identifier-2
```

Format 2 of the ACCEPT statement receives data typed in from the keyboard.

You use Format 2 of the ACCEPT statement to accept a string of input characters from the display in its normal scrolling mode. When the ACCEPT statement is used, input characters are read from the display until the Enter key is pressed. Then a carriage return/line feed pair is sent back to the display. The input data string is considered to consist of all characters typed prior to (but not including) the Enter key.

For a Format 2 ACCEPT with an alphanumeric receiving field, the input data string is transferred to the receiving field exactly as if it were being moved from an alphanumeric field of length equal to the number of characters in the string. (That is, left justification, space filling, and right truncation occur by default; right justification and left truncation occur if the receiving field within a Working-Storage PICTURE clause is described as JUSTIFIED RIGHT.) If the receiving field is alphanumeric-edited, it is treated as an alphanumeric field of equal length (as if each character in its PICTURE were X), so that no insertion editing occurs.

For a Format 2 ACCEPT with a numeric or numeric-edited receiving field, the input data string is subjected to a validity test which depends on the PICTURE of the receiving field. (If the receiving field is described as COMP-0, its PICTURE is treated as S9(5) for purposes of this discussion.) The digits 0 through 9 are considered valid anywhere in the input data string.

The decimal point character (period or comma, depending on the DECIMAL POINT IS clause of the Configuration Section) is considered valid if both of these conditions are true:

- It occurs only once in the input data string.
- The PICTURE of the receiving field contains a fractional digit position (that is, a **9**, **Z**, *, or floating insertion character which appears to the right of either an assumed decimal point (**V**) or an actual decimal point (.)).

The operational sign characters + and – are considered valid only as the first or last character of the input string and only if the PICTURE of the receiving field contains one of the sign indicators **S**, **+**, **–**, **CR**, or **DB**.

All other characters are considered invalid. If the input data string is invalid, the

```
INVALID NUMERIC INPUT -- PLEASE RETYPE
```

message is sent to the screen, and another input data string is read.

When a valid input data string has been obtained, data is transferred to the receiving field exactly as if the instruction being processed were a MOVE to the receiving field from a hypothetical source field. The source field has the following characteristics:

- A PICTURE of the form S9...9V9...9.
- USAGE DISPLAY.
- A total length equal to the number of digits in the input data string.

- As many digit positions to the right of the assumed decimal point as there are digits to the right of the explicit decimal point in the input data string (zero if there is no decimal point in the input data string).
- Current contents equal to the string of digits embedded in the input data string.
- A separate sign with a current negative status if the input data string contains the character –; a current positive status, otherwise.

Example:

```
ACCEPT DATA-1.
```

Format 3 ACCEPT Statement

```
ACCEPT position-spec identifier-3
    {[WITH
    [SPACE-FILL ZERO-FILL]|
    [LEFT-JUSTIFY|
    RIGHT-JUSTIFY]|
    TRAILING-SIGN|PROMPT|UPDATE|
    LENGTH-CHECK|EMPTY-CHECK|
    AUTO-SKIP|BEEP|NO-ECHO] ...}
```

Format 3 of the ACCEPT statement accepts data into a field from a specified position on the display screen. You do this by using the position spec.

The following syntax rules must be observed when the Format 3 ACCEPT is used:

- The SPACE-FILL and ZERO-FILL options may not both be specified in the same ACCEPT statement.
- The LEFT-JUSTIFY and RIGHT-JUSTIFY options may not both be specified within the same ACCEPT statement.
- If *identifier-3* is described as a numeric-edited item, the UPDATE option must not be specified.
- The TRAILING-SIGN option may be specified only if *identifier-3* is described as an elementary numeric data item. If *identifier-3* is described as unsigned, the TRAILING-SIGN option is ignored.
- For an alphanumeric or alphanumeric-edited *identifier-3*, the SPACE-FILL option is assumed if the ZERO-FILL option is not specified, and the LEFT-JUSTIFY option is assumed if the RIGHT-JUSTIFY option is not specified.

- For a numeric or numeric-edited *identifier-3*, the ZERO-FILL option is assumed if the SPACE-FILL option is not specified.

On the following pages, we discuss these additional topics relating to the Format 3 ACCEPT statement:

- Data input field
  - Location
  - Characteristics
- Data input and data transfer
  - Data characters
  - Editing characters
- The WITH phrase

Data Input Field

The *position-spec* and receiving field (*identifier-3*) specifications of the Format 3 ACCEPT statement are used to define the location and characteristics of a data input field on the display.

Location of the Data Input Field

The *position-spec* is of the form:

```
([LIN [+|- integer-1]| integer-2],
 [COL [+|- integer-3]| integer-4])
```

The opening and closing parentheses and the comma (a space must follow the comma) separating the two major bracketed groups are required. The position spec specifies the position on the screen at which the data input field begins. LIN and COL are COBOL special registers. Each behaves like a numeric data item with USAGE COMP, but they may be referenced by every COBOL program without being declared in the Data Division.

If LIN is specified, the data input field begins on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by *integer-1* if *+ integer-1* (or *– integer-1*) is specified. If *integer-2* is specified, the data input field begins on the row whose number is *integer-2*. If neither LIN nor *integer-2* is specified, the data input field begins on the screen row containing the current cursor position.

If COL is specified, the data input field begins in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by *integer-3* if *+ integer-3* (or *– integer-3*) is specified. If *integer-4* is specified, the data input field begins in the screen column whose number is *integer-4*. If neither COL nor *integer-4* is specified, the data input field begins in the screen column containing the current cursor position.

Characteristics of the Data Input Field

The characteristics (other than position) of the data input field on the display are determined by the receiving field's PICTURE specification (which is treated as S9(5) in the case of an item whose USAGE is COMP-0).

For alphanumeric or alphanumeric-edited *identifier-3*, the data input field is simply a string of data input character positions starting at the screen location specified by the position spec. The number of character positions in the data input field equals the number of character positions in the receiving field in memory.

For numeric or numeric-edited *identifier-3*, the data input field may contain any or all of the following:

- Integer digit positions
- Fractional digit positions
- Sign position
- Decimal point position

There is one digit position for each **9**, **Z**, *, **P**, or noninitial floating insertion symbol (a floating insertion symbol is a +, –, or $ which is not the left-most symbol in a PICTURE character string) in the PICTURE of *identifier-3*.

Each digit position in the data input field is a fractional digit position if the corresponding PICTURE character is to the right of an assumed decimal point (**V**) or actual decimal point (.) in the PICTURE of *identifier-3*. Otherwise, it is an integer digit position. There is one sign position if *identifier-3* is described as signed, and no sign position otherwise. There is one decimal point position if there is at least one fractional digit position, and no decimal point position otherwise.

The data input positions that are defined occupy successive character positions on the screen, beginning with the position specified by *position-spec*. If TRAILING-SIGN is specified in the ACCEPT statement, the data input positions must be in the following sequence:

1. Integer digit positions (if any)
2. Decimal point position (if any)
3. Fractional digit positions (if any)
4. Sign position (if any)

If TRAILING-SIGN is not specified, the data input positions must be in the following sequence:

1. Sign position (if any)
2. Integer digit positions (if any)
3. Decimal point position (if any)
4. Fractional digit positions (if any)

Data Input and Data Transfer

A character entered into the data input field by the operator may be treated either as:

- An editing character
- A terminator character
- A data character

When a terminator character is entered, the ACCEPT is ended, and the ESCAPE KEY value is set as described in Figure 7 in this chapter. This value can be interrogated by using the Format 1 ACCEPT statement, ACCEPT FROM ESCAPE KEY.

The editing characters are Ctrl-END (line-delete), Ctrl-HOME, backspace, cursor right, and cursor left. The action of the editing characters is described later in this section. The following describes the input of data characters only.

Data Characters–Alphanumeric

Consider first the Format 3 ACCEPT statement with an alphanumeric or alphanumeric-edited receiving field. An alphanumeric-edited receiving field is treated as an alphanumeric field of the same length (as if every character in its PICTURE is an **X**). Specifically, no insertion editing occurs.

The initial appearance of the data input field depends on the specifications in the WITH phrase of the ACCEPT statement. If UPDATE is specified, the current contents of *identifier-3* are displayed in the input field. In this case, all data input positions are treated as if they are entered by the operator.

If UPDATE is not specified, but PROMPT is specified, a period (.) is displayed in each input data position. If neither UPDATE nor PROMPT is specified, the data input field is not changed. The cursor is placed in the first data input position, and characters are accepted as they are entered by the operator until a terminator character (normally Enter) is encountered.

If AUTO-SKIP is specified in the ACCEPT statement, the ACCEPT is also stopped if the operator enters a character in the last (right-most) data input position.

As each input character is received, it is echoed to (shown on) the screen, except that nondisplayable characters (ASCII characters 0-32) are echoed as ?. (During input, ASCII characters 0-32 are echoed as ? but internally processed as the true character. During output, IBM COBOL can display all 0-255 ASCII characters in quoted literals.) If all positions of the data input field are filled, additional input is ignored until a terminator character or editing character (listed under "Data Input and Data Transfer" in this chapter) is encountered.

If RIGHT-JUSTIFY is specified in the ACCEPT statement, the operator-entered characters are shifted to the right-most positions of the data input field when the ACCEPT is ended. All unentered character positions are filled when the ACCEPT is ended: the fill character is either space (if SPACE-FILL is in effect) or zero (if ZERO-FILL is specified).

The contents of the receiving field are the same set of characters that appear in the input field. However, the operator-entered characters are controlled by the JUSTIFIED specification in the receiving field's data description, not by the RIGHT- or LEFT-JUSTIFY option of the ACCEPT. Excess positions of the receiving field are filled with spaces or zeros based on the SPACE- or ZERO-FILL specification in the ACCEPT statement.

Data Characters–Numeric

Next, consider the Format 3 ACCEPT statement with a numeric or numeric-edited receiving field. As described above, the data input field on the screen may contain integer digit positions, fractional digit positions, or both. First assume that both are present; the other cases will be treated as variations.

As with the alphanumeric ACCEPT, the data input field may be initialized in a way determined by the WITH options specified in the ACCEPT statement. If UPDATE is specified (not permitted for a numeric-edited receiving field), the integer and fractional parts of the data input field are set to the integer and fractional parts of the decimal representation of the initial value of the receiving field, with leading and trailing zeros included, if necessary, to fill all digit positions. Except for leading zeros, these initialization characters are treated as operator-entered data. If UPDATE is not specified, but PROMPT is specified, a zero is displayed in each input digit position. In either of these cases (UPDATE or PROMPT), a decimal point is displayed at the decimal point position.

If neither UPDATE nor PROMPT is specified, the input field on the screen is not initialized, except for the sign position. The sign position is always initialized as positive except when UPDATE is specified, in which case it is initialized according to the sign of the current contents of the receiving field. A positive sign position is shown as a space, and a negative sign position is shown as a minus sign.

The cursor is initially placed in the right-most *integer* digit position, and characters are accepted one at a time as they are entered by the operator.

A received character may be treated in one of several ways. If the incoming character is a digit, previously entered digits shift one position to the left in the input field, and the new digit appears in the right-most integer digit position. If all integer digit positions have not been filled, the cursor remains on the right-most digit position, and another character is accepted.

If the entire integer part of the input field is filled, and you specify AUTO-SKIP, the integer part ends, and the cursor moves to the left-most fractional digit position. If the integer part is filled, and you do not specify AUTO-SKIP, the cursor moves to the decimal point position. All following digits are ignored until the integer part ends with a decimal point.

If the character entered is one of the sign characters + or –, the sign position is changed to a positive or negative status, respectively. Cursor position is not affected.

If the character entered is a decimal point character, the integer part is ended, and the cursor is moved to the left-most fractional digit position.

If the character entered is a field terminator (normally the Enter key), the ACCEPT ends, and the cursor is turned off. Any other character is ignored.

When the *integer* part ends, the cursor is placed in the left-most *fractional* digit position, and operator-entered characters are again accepted. Digits are simply echoed to the display.

The sign characters + and – are treated exactly as they were while integer part digits were being entered. The field terminator character ends the ACCEPT. (If AUTO-SKIP is in effect, filling the entire fractional part also ends the ACCEPT.) Other characters are ignored. After all digit positions of the fractional part have been filled, further digits are also ignored.

If no fractional digit positions are present, the decimal point is ignored as an input character, and entry of integer part digits may be ended only by ending the entire ACCEPT. If no integer digit positions are present, the cursor is initially placed in the left-most fractional digit position and entry of the fractional part digits proceeds as described above.

When you end a Format 3 ACCEPT of a numeric or numeric-edited item, data is transferred to the receiving field. The exact form of the data in the receiving field (after the ACCEPT is processed) is described in the last paragraph of the discussion of the "Format 2 ACCEPT," where the role of the *input data string* mentioned in that paragraph is taken by the string of characters displayed in the data input field.

When the ACCEPT is processed, if SPACE-FILL is in effect, leading zeros in the integer part of the data input field (not in the receiving field) are replaced by spaces, and the leading operational sign, if present, is moved to the right-most space thus created.

Editing Characters

The editing characters Ctrl-END (line-delete), backspace, cursor right, and cursor left may be used to change data which has already been entered (or supplied by the COBOL runtime system as a result of a WITH UPDATE specification). Entering Ctrl-END restarts the ACCEPT, and all data entered by the operator or initially present in the receiving field is lost. The data input field on the display is reinitialized if PROMPT is in effect. Otherwise, the data input field is filled with spaces or zeros according to the SPACE-FILL or ZERO-FILL specification.

Typing the cursor right or cursor left moves the cursor forward or back one *data input position*.

In no case, however, does the cursor right or cursor left key move the cursor outside the range of positions, including:

- The positions already entered by the operator (or filled when WITH UPDATE is specified)
- The right-most data input position that the cursor has occupied during the processing of this ACCEPT

If the cursor is moved to a position of this range other than the right-most, and a legal data character is entered, it is displayed at the current cursor position and the cursor is moved forward one data position.

Typing the backspace key effectively cancels the last data character entered. For alphanumeric and functional numeric fields, the cursor is moved back one data position, and a fill character (space or zero) is displayed under the cursor. When the cursor is to the left of the decimal point for a numeric ACCEPT, no fill character is displayed and the cursor is not moved; but the digit at the cursor position is deleted, and all digits to the left of the cursor are shifted one position to the right.

The backspace character has no effect unless the cursor is in position to accept a new data character; in other words, it has no effect if the cursor right (or left) has been used to move the cursor over already keyed positions.

WITH Phrase Summary

The following list summarizes the effects of the WITH phrase specifications for a Format 3 ACCEPT with an *alphanumeric* or *alphanumeric-edited* receiving field:

- SPACE-FILL causes unkeyed character positions of the data input field and the receiving field to be space-filled when the ACCEPT ends.
- ZERO-FILL causes unkeyed character positions of the data input field and the receiving field to be set to ASCII zeros when the ACCEPT ends.
- LEFT-JUSTIFY is treated as commentary.

- RIGHT-JUSTIFY causes operator-entered characters to occupy the right-most positions of the data input field after the ACCEPT ends. Note that the transferred data in the receiving field is controlled by the JUSTIFIED declaration or default of the receiving field's data description, not by the WITH RIGHT-JUSTIFY phrase.
- PROMPT causes the data input field on the screen to be set to all periods (.) before input characters are accepted.
- UPDATE causes the data input field to be initialized with the initial contents of the receiving field and the initial data to be treated as operator-entered data.
- LENGTH-CHECK causes a field terminator character to be ignored unless every data input position has been filled.
- EMPTY-CHECK causes terminator characters to not be accepted unless at least one entry has been made in the field.
- AUTO-SKIP forces the ACCEPT to end when all data input positions have been filled. A terminator character explicitly entered has its usual effect.
- BEEP causes the speaker to sound when the ACCEPT is initialized and the system is ready to accept operator input.
- NO-ECHO causes the field to be filled with asterisks (*) for each character entered.

The following list summarizes the effects of the WITH phrase specifications for the Format 3 ACCEPT with a *numeric* or *numeric-edited* receiving field:

- SPACE-FILL causes unkeyed digit positions of the data input field (not of the receiving field) to the left of the (possibly implied) decimal point to be space-filled when the ACCEPT ends. Any leading operational signs are displayed in the right-most space thus created.

- ZERO-FILL causes all unkeyed digit positions of the data input field to be set to zero when the ACCEPT ends.

- LEFT-JUSTIFY and RIGHT-JUSTIFY have no effect for a numeric or numeric-edited receiving field.

- TRAILING-SIGN causes the operational sign to appear as the right-most position of the data input field. Ordinarily, the sign is the left-most position of the field.

- PROMPT causes the data input field positions to be initialized as follows before input characters are accepted:
  - Digit positions as zeros
  - Decimal point position (if any) as the decimal point character
  - Sign position (if any) as a space

- UPDATE causes the data input field to be initialized to the current contents of the receiving field and causes this initial data to be treated like operator-entered data.

- LENGTH-CHECK causes a received decimal point character to be ignored unless all integer digit positions have been entered. A field terminator character is ignored unless all digit positions have been entered.
- EMPTY-CHECK causes terminator characters to not be accepted unless at least one entry has been made in the field.
- AUTO-SKIP causes the integer part of the ACCEPT to end when all integer digit positions have been entered and causes the entire ACCEPT to end when all digit positions have been entered.
- BEEP causes the speaker to sound when the ACCEPT is initialized and the system is ready to accept operator input.
- NO-ECHO causes the screen to be filled with asterisks (*) for each character entered.

Receiving Field: 05 RS-DISCOUNT PIC X(8). Initial Contents: ABCDEFGH ACCEPT Statement: ACCEPT (1, 1) RS-DISCOUNT WITH PROMPT.	Set up prior to running
At Start of ACCEPT: .... Operator Enters N: N....... Operator Enters ONE: NONE.... Operator Presses Enter key: NONEbbbb	(Cursor is at top left-hand corner) During the ACCEPT
Final Contents of Receiving Field: NONEbbbb	Result

Figure 8. Example 1 of Format 3 ACCEPT Statement

Receiving Field: 10 VEND-NAME PIC X(12). Initial Contents: ACMEbHAMMERS ACCEPT Statement: ACCEPT (1, 1) VEND-NAME WITH PROMPT UPDATE.	Set up prior to running
At Start of ACCEPT: ACMEbHAMMERS (If operator presses Enter key here, the receiving field will not be changed.) Operator Enters Line-delete: . Operator Enters XYZ: XYZ..... Operator Presses Enter key: XYZbbbbbbbbb	(Cursor is at top left-hand corner) During the ACCEPT
Final Contents of Receiving Field: XYZbbbbbbbbb	Result

Figure 9. Example 2 of Format 3 ACCEPT Statement

Receiving Field: 05 CREDIT PIC S9(4)V99. Initial Contents: + 111111 ACCEPT Statement: ACCEPT (LIN + 4, COL - 3) CREDIT-LINE WITH PROMPT TRAILING-SIGN.	Set up prior to running
At Start of ACCEPT: 0000.00b Operator Enters 8: 0008.00b Operator Enters 7: 0087.00b Operator Enters -: 0087.00- Operator Enters 6: 0876.00- Operator Enters N: 0876.00- Operator Enters .: 0876.00- Operator Enters 5: 0876.50- Operator Presses Enter key: 0876.50-	(Cursor position of left-hand character is 4 plus current LIN and 3 less than the current COL) During the ACCEPT
Final Contents of Receiving Field: 0876.50	Result

Figure 10. Example 3 of Format 3 ACCEPT Statement

Format 4 ACCEPT Statement

```
ACCEPT screen-name
    [ON ESCAPE imperative-statement]
```

Format 4 of the ACCEPT statement is used to *accept data* from an entire formatted screen. Screen items having only VALUE *literal* or FROM clause have no effect on the operation of the ACCEPT *screen-name* statement (Format 4). If you have formatted your comments under *screen-name* in the Screen Section and want to display your comments on the screen, then you should perform the DISPLAY *screen-name* statement prior to the ACCEPT *screen-name* statement.

Format 4 of the ACCEPT statement transfers information from the display to all TO and/or USING fields specified in the Screen Section definition of *screen-name* or to any screen item subordinate to *screen-name*. Each such transfer consists of an implicit Format 3 ACCEPT of a field defined by the appropriate screen item's PICTURE followed by an implicit MOVE to the associated TO or USING field. Fields are accepted in the order in which they are defined under the *screen-name* in the Screen Section. You can change this order by using the backtab key (as described below), but the position of the field on the screen does not affect the order.

If the Esc key is pressed during data input, the entire ACCEPT is ended without moving the current field to the associated TO or USING item, the ESCAPE KEY value is set to 01 (see "Format 1 ACCEPT Statement"), and the ON ESCAPE statement is processed. Items entered prior to the current item are updated before control is passed to the ON ESCAPE statement.

If a function key is typed, the appropriate ESCAPE KEY value is set, and the entire ACCEPT ends.

If a field-terminator key (Enter, tab) is typed, the ESCAPE KEY value is set to 00, and the cursor moves to the next input field defined under *screen-name*, if one exists. If the current field is the *last* field, the entire ACCEPT ends.

If the backtab key is typed, the current field is ended, and the cursor moves to the previous input field defined under *screen-name*. If the current field is the first field, the cursor does not move from that field. When a field is ended by a function key, field-terminator key, or backtab key, the contents of the current field are moved to the associated TO or USING item, except in the case where no data characters and no editing characters have been entered in that field. This allows the operator to tab forward or backward through the input fields without affecting the contents of the receiving items.

All the editing keys and validation features described for the Format 3 ACCEPT apply to the Format 4 ACCEPT as well. Several Screen Section specifications correspond to the Format 3 ACCEPT options:

AUTO corresponds to AUTO-SKIP
BELL corresponds to BEEP
FULL corresponds to LENGTH-CHECK
JUSTIFIED corresponds to RIGHT-JUSTIFY
SECURE corresponds to NO-ECHO
REQUIRED corresponds to EMPTY-CHECK

If these are coded in the Screen Section, then the effect is the same as if individual Format 3 ACCEPT statements are performed. Also, if an input field specifies the USING clause or both a FROM and TO clause, the ACCEPT is processed with the UPDATE option. Format 4 ACCEPT statements always use the PROMPT and TRAILING-SIGN options when performing the individual accepts.

If the screen item's PICTURE specifies a numeric-edited or alphanumeric-edited input field, the ACCEPT is performed as if the field is numeric or alphanumeric, respectively. When the field is ended, the data is edited according to the PICTURE and redisplayed in the specified screen position. In this case, the JUSTIFIED clause has no effect.

Moves from screen fields to receiving items follow the standard COBOL rules for MOVE statements, except that moves from numeric-edited fields are allowed. In this case, the data is input as if the field is numeric and the move uses only the sign, decimal point, and digit characters.

Again, the Format 4 ACCEPT does not cause the display of any text or prompting label information. This can be accomplished by first displaying *screen-name* (if VALUE clauses exist) and then accepting *screen-name* (for corresponding input data). See the discussions of "DISPLAY Statement" and "SCREEN Section."

Example:

```
ACCEPT SCREEN-1.
```

ADD Statement

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

Format:

```
ADD numeric-literal|data-name-1...
    TO|GIVING data-name-n
    [ROUNDED] [SIZE-ERROR-clause]
```

Remarks: Either the TO option or the GIVING option must be specified.

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*.

When you use the GIVING option, at least two *data-names* and/or *numeric-literals* must be coded between ADD and GIVING. The sum of the values of these *data-names* and *literals* (not including *data-name-n*) replaces the value of *data-name-n*.

The ROUNDED and SIZE ERROR options are discussed in Chapter 2, "Arithmetic Statements."

Example:

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

The first statement places the sum of INTEREST, DEPOSIT, and BALANCE in item BALANCE. The second statement places the sum of REGULAR-TIME and OVERTIME earnings in item GROSS-PAY.

ALTER Statement

Purpose: The ALTER statement modifies a simple GO TO statement elsewhere in the Procedure Division, thus changing the sequence in which program statements are processed.

Format:

```
ALTER paragraph TO
    [PROCEED TO] procedure-name
```

Remarks: *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.

Example:

```
SECTION MF-READ.
    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.
```

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

COMPUTE Statement

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

Format:

```
COMPUTE data-name-1 [ROUNDED]
    [data-name-n [rounded]] ... =
    data-name-2|numeric-literal|
    arithmetic-expression
    [SIZE-ERROR-clause]
```

Remarks: An *arithmetic expression* is a proper combination of numeric literals, data names, arithmetic operators, and parentheses. See "Arithmetic Expressions" in Chapter 2 for more information. Note also that COMPUTE may have multiple targets.

Example:

```
COMPUTE GROSS-PAY ROUNDED = BASE-SALARY *
    (1 + 1.5 * (HOURS - 40) / 40).
COMPUTE A, B(I) = -C - D(3).
```

DISPLAY Statement

Purpose: The DISPLAY statement outputs data to the screen or printer at runtime without the complexities of file definition.

Format:

```
DISPLAY {[position-spec]
    {identifier|literal|ERASE} ...
    [UPON mnemonic-name]|[screen-name]}
```

Remarks: The DISPLAY statement must be coded in accordance with the following rules:

- *Mnemonic-name* must be defined in the PRINTER IS clause of the SPECIAL-NAMES paragraph of the Configuration Section.
- *Screen-name* must be defined in the Screen Section of the Data Division.

The DISPLAY statement causes output to be sent to the screen, unless UPON *mnemonic-name* is specified, in which case output is sent to the printer. Each display item (that is, each occurrence of *identifier*, *literal*, or ERASE) is processed in turn as described in the paragraphs below. Then, if no *position-spec* is coded in the entire DISPLAY statement, a carriage return and a line feed are sent to the receiving device.

Position-spec

For each display-item, if *position-spec* is specified, the cursor is positioned prior to the transfer of data for this item.

Position-spec is of the form:

```
([LIN [+|- integer-1]|integer-2],
[COL [+|- integer-3]|integer-4])
```

The opening and closing parentheses and the comma separating the two major bracketed groups are required. The *position-spec* specifies the position on the screen at which the cursor is to be placed. LIN and COL are COBOL special registers. Each behaves like a numeric data item with USAGE COMP-0, but they may be referenced by every COBOL program without being declared in the Data Division.

To display the value in LIN or COL, the value must first be moved to a Working-Storage data item, and then that data item may be displayed.

If LIN is specified, the cursor is placed on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by *integer-1* if *+ integer-1* (or *– integer-1*) is specified. If *integer-2* is specified, the cursor is placed on the row whose number is *integer-2*. If neither LIN nor *integer-2* is specified, the cursor is placed on the screen row containing the current cursor position.

If COL is specified, the cursor is placed in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by *integer-3* if *+ integer-3* (or *– integer-3*) is specified. If *integer-4* is specified, the cursor is placed in the screen column whose number is *integer-4*. If neither COL nor *integer-4* is specified, the cursor is placed in the screen column that contains the current cursor position.

Identifier, Literal, and ERASE

If *identifier* or *literal* is specified for a given display item, the contents of *identifier* or the value of *literal* are sent to the receiving device. Since the data transfer occurs without conversion or reformatting, we recommend that numeric data be moved to numeric-edited fields for purposes of DISPLAY.

If ERASE is specified and if *position-spec* is coded for this or a previous display item, the screen is cleared from the current cursor position to the end of the screen. The initial cursor position for the next display item is the position specified by the *position-spec* coded in the ERASE display item, if present, or the position in which the cursor was left by the previous display item. If ERASE is specified and no *position-spec* has been encountered up to this point in the DISPLAY statement, no action is taken.

Screen-name

The DISPLAY *screen-name* statement transfers information from *screen-name* (or from each elementary screen item subordinate to *screen-name*) to the screen. For each such screen item having a VALUE, FROM, or USING specification, the specified literal or field is the source of the displayed data. For a field having only a TO clause, the effect is as if FROM ALL "." (period) had been specified.

The source data is moved implicitly to a temporary item defined by the appropriate screen item's PICTURE (or by the length of the data in the case of a VALUE *literal*). Then an implied identifier type DISPLAY of the constructed temporary is processed as modified by the positioning and control clause coded in the definition of the appropriate screen item.

Example:

```
DISPLAY DATA-1.
DISPLAY (10, 12) DATA-1.
DISPLAY "DATA-1 = " DATA-1.
DISPLAY SCREEN-1.
```

DIVIDE Statement

Purpose: The DIVIDE statement divides two numeric values and stores the quotient.

Format:

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

Remarks: The BY means 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 you do not use the GIVING option in this case, then the *first* operand must be a data name, in which the quotient is stored.

The INTO means that the first operand is the divisor and the second operand is the dividend. If you do not use the GIVING option 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.

The ROUNDED and SIZE ERROR options are discussed in Chapter 2, "Arithmetic Statements."

Example:

```
DIVIDE VALUE1 BY 100.
DIVIDE TOTAL-NUM INTO AMT GIVING AVERAGE.
```

EXHIBIT Statement

Purpose: The EXHIBIT statement prints or displays requested data names and data values at designated points for debugging purposes.

Format:

```
EXHIBIT NAMED {[position-spec]
    {identifier|literal|ERASE}} ...
    [UPON mnemonic-name]
```

Remarks: This statement produces a printout of values of the indicated *literal*, or data items in the format *data-name = value*. For more details concerning the syntax, see "DISPLAY Statement" or "SOURCE-COMPUTER Paragraph."

> **Note:** You may often want to include this statement on source lines that contain D in column 7, so that this statement is ignored by the compiler unless WITH DEBUGGING MODE is included in the SOURCE-COMPUTER paragraph.

Example:

```
    EXHIBIT DATA1
    EXHIBIT (10,12) DATA1.
D   EXHIBIT COUNTER.
```

EXIT Statement

Purpose: The EXIT statement is used where it is necessary to provide an end point for a procedure.

Format:

```
EXIT.
```

Remarks: 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 you decide to bypass some part of a paragraph or section.

Example:

```
END-POINT.
    EXIT.
```

GO TO Statement

Purpose: The GO TO statement transfers control from one portion of a program to another.

Format:

```
GO TO [procedure-name-1 [[procedure-name-2]...
      DEPENDING ON data-name]]
```

Remarks: 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 by an ALTER statement before control gets to it.

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 the program 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, the GO statement must be the last statement in that sequence.

Example:

```
GO TO HOURS-CALC.
GO TO DO-WEEKLY, DO-SEMI, DO-MONTHLY
    DEPENDING ON PAYROLL-PERIOD.
```

IF Statement

Purpose: The IF statement lets you specify a series of procedural statements to be processed in the event a stated condition is true. Optionally, you may specify an alternative series of statements to be processed if the condition is false.

Format:

```
IF condition
    statement(s)-1|NEXT SENTENCE
    [ELSE
    statement(s)-2|NEXT SENTENCE]
```

Remarks: The IF statement must be followed immediately by a period.

Example:

```
IF BALANCE = 0 GO TO NOT-FOUND.
IF T LESS THAN 5 NEXT SENTENCE
    ELSE GO TO T-1-4.
IF ACCOUNT-FIELD = SPACES OR
    NAME = SPACES ADD 1 TO
    SKIP-COUNT ELSE GO TO BYPASS.
```

The first series of statements is performed only if the designated condition is true. The second series of statements (ELSE part) is processed only if the designated condition is false. Refer to Appendix F for a discussion of nested IF statements.

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

IF Statement

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; for example, A < B OR C = D. Refer to Appendix E for further permissible forms involving parentheses, NOT, or abbreviations.

The simplest "simple relations" have three basic forms, expressed by the relational symbols equal to, less than, or greater than (that is, =, <, 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

Let's briefly discuss how relation conditions can be compounded. The reserved words AND or OR let you specify a series of relational tests, as follows:

- Individual relations connected by AND specify a compound condition that is met (true) only if all the individual relationships are met.

- 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 E for formal specification of evaluation rules.

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

In the above example, the program runs as follows, depending on various data values.

Data Value				Does Control 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

Figure 11. Effects of Conditions on Program Flow

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

Before we discuss class test, sign test, and condition name test conditions, we will discuss methods of performing comparisons.

Numeric Comparisons: The data operands are compared after alignment of their decimal positions. The results are 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 10) may appear in a comparison. Comparison of any two numeric operands is permitted regardless of length and of the formats specified in their respective USAGE clauses.

Character Comparisons: Unequal-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 (and a-z) are in an ascending sequence, and digits are less than letters. When compared, group items are treated simply as characters. Refer to Appendix G for all ASCII character representations. If one operand is numeric and the other is not, the numeric operand must be an integer and have an implicit or explicit USAGE IS DISPLAY.

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

A *class test condition* has the following syntactical format:

```
data-name IS [NOT] NUMERIC|ALPHABETIC
```

This condition specifies an examination of the data item content to determine whether it is numeric or alphabetic. When the test is for *numeric*, all characters must be proper digit representations (**0 . . . 9**) with an operational sign if SIGN IS SEPARATE is specified. When the test is for *alphabetic*, only *uppercase* alphabetic (**A . . . Z**) or blank space characters must be present. 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. (See "Level 88 Condition Names" in Chapter 6.)

INSPECT Statement

Purpose: The INSPECT statement lets you examine a character-string item. Options permit various combinations of the following actions:

- Counting appearances of a specified character
- Replacing a specified character with another
- Limiting the above actions by requiring the appearance of other specific characters

Format:

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

Where TALLYING-clause has the format:

```
TALLYING data-name-2 FOR
    {ALL|LEADING operand-3|CHARACTERS}
    [BEFORE|AFTER INITIAL operand-4]
```

and REPLACING-clause has the format:

```
REPLACING {ALL|LEADING|FIRST operand-5|
    CHARACTERS}
    BY operand-6
    [BEFORE|AFTER INITIAL operand-7]
```

Remarks: Because *data-name-1* is to be treated as a string of characters by INSPECT, it should be described (implicitly or explicitly) by USAGE IS DISPLAY (DISPLAY is assumed if no USAGE clause is specified). It must not be described by USAGE IS INDEX, COMP-0, 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 1, a figurative constant signifying a single character, or a data name of an item whose length is 1.

You must have either a TALLYING-clause or a REPLACING-clause, or both. If both are present, TALLYING-clause must be first.

The 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. If BEFORE INITIAL *operand-4* is specified, the counting process stops upon encountering a character in *data-name-1* which matches *operand-4*. If AFTER INITIAL *operand-4* is present, the counting process begins only after the system detects a character in *data-name-1* that matches *operand-4*.

Also going from left to right, the REPLACING-clause replaces characters under conditions specified by the REPLACING-clause. If BEFORE INITIAL *operand-7* is present, replacement stops after detection of a character in *data-name-1* that matches *operand-7*. If AFTER INITIAL *operand-7* is present, replacement starts after 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:

- CHARACTERS implies that every character in *data-name-1* is to be tallied or replaced.

- ALL *operand* means that all characters in *data-name-1* which match the *operand* character are to participate in TALLYING/REPLACING.

- LEADING *operand* specifies that only characters matching *operand* from the left-most portion of *data-name-1* which are contiguous (such as leading zeros) are to participate in TALLYING or REPLACING.

- 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 initially to zero elsewhere in the program.

Example:

```
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 is unsigned.

MOVE Statement

Purpose: You use the MOVE statement to move data from one area of memory to another and to perform conversions and/or editing on the data that is moved.

Format:

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

Remarks: 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*, *data-name-4*, etc.).

When a group item is a receiving field, characters are moved without regard to the level structure of the group involved and without being edited.

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 in 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)
```

The value *temp* is an intermediate result field assigned automatically by the compiler.

The following considerations pertain to moving items:

- Numeric (external or internal decimal, binary, numeric literal, or ZERO) or alphanumeric to numeric or report:
  - The items are aligned by decimal points, with generation of zeros or truncation on either end, as required. If the source is alphanumeric, it is treated as an unsigned integer and should not be longer than 31 characters.
  - 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 a USAGE IS DISPLAY clause.
  - 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.

- Do not move an item whose PICTURE declares it to be alphabetic or alphanumeric edited to a numeric or report item. Do not move a numeric item of any sort to an alphabetic item.

  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 you have specified SIGN IS SEPARATE.

- Nonnumeric source and destinations:
  - The characters are placed in the receiving area from left to right, unless JUSTIFIED RIGHT applies.
  - If the receiving field is not completely filled by the data being moved, the remaining positions are filled with spaces.
  - If the source field is longer than the receiving field, the move ends as soon as the receiving field is filled.
  - When overlapping fields are involved, results are not predictable.
  - An item having USAGE IS INDEX cannot appear as an operand of a MOVE statement. See "SET Statement" in this chapter.

Example:

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

Figure 12. Examples of Data Movement

Notes:

1. In the above table, **b** represents a blank.

2. The number represented as the **Value** is not necessarily what would be seen if you displayed it with the DISPLAY statement (see "SIGN Clause").

MULTIPLY Statement

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

Format:

```
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]
```

Remarks: When the GIVING option is omitted, the second operand must be a data name; the product replaces the value of *data-name-2*.

> **Note:** Because this order might seem somewhat unnatural, we recommend that you use the GIVING option. For example, a new BALANCE value is computed by the statement MULTIPLY BALANCE BY 1.03 GIVING BALANCE. This is equivalent to MULTIPLY 1.03 BY BALANCE.

The ROUNDED and SIZE ERROR options are discussed in Chapter 2, "Arithmetic Statements."

Example:

```
MULTIPLY VALUE1 BY VALUE2.
MULTIPLY TAX-RATE BY GROSS GIVING TAX-AMT.
```

PERFORM Statement

Purpose: The PERFORM statement lets you process a separate body of program steps.

Format: Two formats of the PERFORM statement are available:

Option 1

```
PERFORM range [integer|data-name TIMES]
```

Option 2

```
PERFORM range
    [VARYING index-name|data-name
    FROM amount-1 BY amount-2
    UNTIL condition.
```

(A more extensive version of Option 2 is available for varying two or three items concurrently, as explained in Appendix I.)

Remarks: *Range* is:

```
procedure-name-1 [THRU|THROUGH
    procedure-name-2]
```

Procedure-name-1 may be a paragraph name or a section name.

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 processed to set them up; in other cases, control passes right through.

The generic operands *amount-1* and *amount-2* may be numeric literals, index names, or data names. In practice, these amount specifications are frequently integers, or data names that contain integers; and the specified *data-name* often 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, program control proceeds to the next statement following the PERFORM statement.

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 this time, control proceeds to the next statement after the PERFORM.

Note: The condition in an Option 2 PERFORM is evaluated prior to each attempted processing 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 processed at all.

At runtime, you cannot have concurrently active PERFORM *ranges* whose end points are the same.

Example:

```
PERFORM READ-DATA THRU WRITE-DATA.
PERFORM ARITHMETIC-SECTION 10 TIMES.
PERFORM CYCLE VARYING SUBSCRIPT
    FROM 1 BY 1 UNTIL > 10.
```

STOP Statement

Purpose: The STOP statement stops or delays the object program while it is running.

Format:

```
STOP RUN|literal
```

Remarks: STOP RUN ends a program, returning control to the Disk Operating System (DOS). 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 screen and suspends the program. The program is resumed only after operator intervention. Presumably, the operator performs a function suggested by the content of the *literal*, prior to resuming the program by pressing the Enter key.

Example:

```
STOP RUN.
STOP "CHECK DATA BEFORE CONTINUING".
```

STRING Statement

Purpose: The STRING statement allows concatenation of multiple sending data item values into a single receiving item.

Format:

```
STRING {operand-1...
    DELIMITED BY operand-2|SIZE} ...
    INTO identifier-1
    [WITH POINTER identifier-2]
    [ON OVERFLOW imperative-statement]
```

Remarks: In this format, *operand* means a nonnumeric 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 terminating 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 search pattern which, if found to match a contiguous sequence 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 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. The *imperative-statement* given in the OVERFLOW phrase (if any) is performed. If there is no OVERFLOW phrase, control is transferred to the next 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 is a POINTER phrase, the resultant value of *identifier-2* equals its original value plus the number of characters moved during the STRING statement.

STRING
Statement

Example:

```
STRING SOURCE-STRING DELIMITED BY SIZE
    INTO DESTINATION-STRING.
STRING STRING-1 DELIMITED BY "#"
    INTO STRING-2
    WITH POINTER S-POINTER.
STRING NAME DELIMITED BY SIZE
    INTO NEW-NAME
    ON OVERFLOW GO TO PROC-NAME.
```

SUBTRACT Statement

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

Format:

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

Remarks: The SUBTRACT statement adds the values of all the operands that precede FROM and subtracts 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*.

The ROUNDED and SIZE ERROR options are discussed in Chapter 2, "Arithmetic Statements".

Example:

```
SUBTRACT TAX, FICA, OTHER FROM GROSS-PAY.
SUBTRACT TAX, FICA, OTHER FROM GROSS-PAY
    GIVING NET-PAY.
```

TRACE Statement

Purpose: The TRACE mode displays program procedure names on your screen in the order in which you run them.

Format:

```
READY | RESET  TRACE
```

Remarks: When you run a READY TRACE statement, the TRACE mode causes every section and paragraph name to be printed each time it is entered. The RESET TRACE statement stops such printing.

A printed list of procedure names, in the order in which they are run, is valuable when you try to find a program error. It helps you find the point at which the actual program flow departed from the expected program flow.

For more information on debugging, see "SOURCE-COMPUTER Paragraph" in Chapter 5.

Example:

```
READY TRACE.
PERFORM PARA-A THRU PARA-Z.
  .
  .
  .
RESET TRACE.
```

UNSTRING Statement

Purpose: The UNSTRING statement causes data in a single sending field to be separated into subfields that are placed into multiple receiving fields.

Format:

```
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]
```

Remarks: The delimiters used to separate subfields are entered in the DELIMITED BY phrase. Each time a succession of characters matches one of the nonnumeric literals, one-character figurative constants, or data item values named by *operand-i*, the current collection of sending characters is ended 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:

- An unedited alphabetic item
- A character-string (alphanumeric) item
- A group item
- An external decimal item (numeric, usage DISPLAY) whose PICTURE does not contain a **P** character

When an 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 delimits the data moved into *identifier-2* is stored in *identifier-3*, which should be an alphanumeric item.

If a COUNT IN phrase is present, the number of characters moved into *identifier-2* are 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. If there isn't a POINTER phrase, then 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 is copied back into *identifier-5*.

When the value of the logical pointer is less than one or exceeds the size of *identifier-1*, *overflow* occurs. 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. When the character string is completed by recognition of a delimiter or by acquiring as many characters as the size of the current receiving field can hold, the character string then moves 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*, is 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 *operand-i* or *identifier-2*, *3*, or *4* is evaluated immediately before access to the data-item.

UNSTRING Statement

Example:

```
UNSTRING SOURCE-FIELD
         DELIMITED BY ALL SPACE OR "."
         INTO FIELD-1
              DELIMITER IN DELIM-1
              COUNT IN CNT-1
            FIELD-1
              DELIMITER IN DELIM-2
            FIELD-3
              COUNT IN CNT-3
            FIELD-4
            FIELD-5
         WITH POINTER S-POINTER
         TALLYING IN TALLY-CNT
         ON OVERFLOW DISPLAY "**OVERFLOW**".
```

CHAPTER 8. DATA INPUT AND OUTPUT

Contents

Introduction

IBM COBOL allows you to select specific formats when outputting data to files and/or the printer. IBM COBOL also offers a variety of methods for organizing and accessing the data in those files.

In this chapter, we will discuss the following methods with which you can input or output data in a COBOL program:

- Printer files
- Communication files
- Display/keyboard
- Diskette files

We will also discuss the three types of diskette file organization:

- Sequential
- Relative
- Indexed

How to Handle Printer Files

Printer files should be viewed as a stream of characters going to the printer. Records should be defined as the fields to appear on the printer. No extra characters are needed in the record for carriage control. 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.

To send a file to the printer, you use the SELECT *filename* ASSIGN TO PRINTER clause. Then under FD, you must specify the clause LABEL RECORD IS OMITTED, and you must not specify the VALUE OF FILE-ID clause.

You can also send files to the printer by using the DOS reserved words LPT1 and PRN. If you assign these to the VALUE OF FILE-ID clause, IBM COBOL treats the files as disk files. (That is, you assign the files to disk in the SELECT clause, but DOS sends the files to the printer.)

How to Handle Communication Files

If you have the asynchronous communications adapter, you can communicate with other IBM Personal Computers by assigning the DOS reserved words AUX or COM1 in the VALUE OF FILE-ID clause in your program. DOS recognizes these words and sends the file to the RS232 port. Any protocols must be handled by your programs.

If you have connected your IBM Personal Computer to a printer (other than the IBM Personal Computer Printer) through the RS232 port, then you should assign AUX in the VALUE OF FILE-ID clause.

How to Handle the Display/Keyboard

Display Output

Normally, output to the screen is done by the DISPLAY or EXHIBIT statement. Characters are sent one at a time to the screen. If no cursor positioning is specified for any of the displayed items, carriage return and line feed are sent following the last displayed item. Otherwise, no assumptions about carriage control are made.

You can also send output to the display by assigning the name CON or USER to the VALUE OF FILE-ID clause.

CON is a DOS reserved word for the display device. Output written to CON is buffered as a file and thus appears on the display in blocks of characters at a time.

USER is a special COBOL reserved word for the display. Output written to USER is not buffered and thus appears on the display on a character-by-character basis.

Keyboard Input

All input from the keyboard is done by Formats 2, 3, and 4 of the ACCEPT statement. One of two methods of input is used, depending on the type of ACCEPT being performed.

For a Format 2 ACCEPT, a full line of input is typed, using the DOS facilities for character echo and input editing, ending with the Enter key. The editing characters for Formats 3 and 4 have no effect.

For a Format 3 or 4 ACCEPT, each character typed is read directly by the runtime ACCEPT module by using a call to the Disk Operating System (DOS). The ACCEPT module performs all necessary character echo and input editing functions using the editing control characters, function keys, and terminator keys described with "ESCAPE KEY Values" under "Format 1 ACCEPT Statement" in Chapter 7.

How to Handle Diskette Files

File access is defined in the Environment Division, Input-Output Section under File-Control. The actual filename and layout are specified in the Data Division, in the File Section.

Diskette files must have LABEL RECORD IS STANDARD declared and have a VALUE OF FILE-ID clause. File ID formats are described under "How to Compile a COBOL Program" in Chapter 3. Block clauses are checked for syntax but have no effect on any type of file.

Three types of data files can be stored on diskettes and used by your programs. Files have three types of organizations:

- Sequential (regular sequential and line sequential)
- Relative
- Indexed

The types of organizations are discussed later in this chapter, accompanied by the Procedure Division statements used with each type.

What Is Sequential File Organization?

The format of regular *sequential* organization files is that of a 2-byte count of the record length followed by the actual record, for as many records as exist in the file. This type of file is normally created by a COBOL program.

The *line sequential* organization has each record followed by Enter and a line feed delimiter, for as many records as exist in the file. This type of file is normally created by using an editor.

Both organizations pad remaining space in the last physical block with Ctrl-Z characters, indicating end-of-file. To make maximum use of diskette space, records are packed together with no unnecessary bytes in between.

Syntax Considerations

In the Environment Division, the SELECT entry must specify ORGANIZATION IS SEQUENTIAL, or ORGANIZATION IS LINE SEQUENTIAL. The ORGANIZATION clause is optional for sequential files (*not* line sequential), so this clause may be omitted. The ACCESS clause is also optional, but if entered, it must specify ACCESS MODE IS SEQUENTIAL.

Procedure Division Statements for Sequential Files

The I/O statements for sequential files follow the discussions of relative and indexed files. Each statement discussion includes the format and an example.

Note: See Sequential File Status reporting on page 5-9.

What Is Relative File Organization?

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; therefore, none is provided. Deleted records are filled with hex value "00". Additionally, 6 bytes are reserved at the beginning of the file to contain system bookkeeping information.

Relative organization is restricted to diskette files. Records are differentiated on the basis of a *relative record number*, which may range from 1 to 32,767. 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 organization 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 at will.

Syntax Considerations

In the Environment Division, the ACCESS and ORGANIZATION clause formats are:

```
ACCESS MODE IS SEQUENTIAL|RANDOM|DYNAMIC
ORGANIZATION IS RELATIVE.
```

ASSIGN, RESERVE, and FILE STATUS clause formats are identical to those specified in Chapter 5, under "FILE-CONTROL Paragraph."

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

The first byte of the record area associated with a relative file must not be set to *binary zero* by using a COMP-0 or COMP-3 item, nor set to LOW-VALUE for an alphanumeric item.

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 integer item *not* contained within any record description of the file itself. Its value must be positive and nonzero.

FILE STATUS Reporting

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

File Status Left	File Status Right	Meaning
0	0	Successful completion
1	0	EOF
2	2	Attempt to write a duplicate key
2	3	No record found
2	4	Disk space full
3	0	Permanent error
9	1	File structure destroyed

Procedure Division Statements 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 statements in Figure 14 also apply to relative files.)

The OPEN and CLOSE statements described under sequential files are applicable to relative files, except for the EXTEND phrase.

What Is Indexed File Organization?

Each indexed file declared in a COBOL program generates two diskette files. The file specification in the VALUE OF FILE-ID clause specifies a file containing data only. The filename included in the file specification is concatenated with an extension KEY to form the file specification of the key file.

The *key file* contains keys, pointers to keys, and pointers to data. The format of this file is very complex, but it follows the guidelines for a prefix B-tree.

The *data file* consists of data records. Each data record is preceded by a 2-byte length field and a 1-byte *reference count* that indicates whether a record has been deleted. The data file is terminated by a control record that has a length field containing the number 2 followed by 2-bytes of high-values.

The key file is divided into 256-byte units, called *granules*. Five possible granule types exist. The following list shows the granule type indicator values, which are located in the first byte of each granule:

Value	Type Indicator
1	Data set control block
2	Key set control block
3	Node
4	Leaf
5	Deleted granule

Figure 13. Granule Type Indicators

Note: See Indexed File Recovery Utility (REBUILD) in Appendix K.

The key file contains only one data set control block in the first granule, one key set control block for the primary file key, and additional key set control blocks for alternate keys.

Damaged flags exist in the fourth byte of the data set control block and in the fourth byte of each key set control block. These flags are set to nonzero values when the file is opened for updating, and restored to zero when the file is closed.

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 *random access*, you control the order of access to records. Each record desired is accessed by placing the value of its key in a key data item prior to an access statement.

In *dynamic access*, the program logic may change from sequential access to random access, and vice versa, at will.

Syntax Considerations

In the Environment Division, the ACCESS and ORGANIZATION clause formats are:

```
ACCESS MODE IS SEQUENTIAL|RANDOM|DYNAMIC
ORGANIZATION IS INDEXED.
```

ASSIGN, RESERVE, and FILE STATUS clause formats are identical to those specified in Chapter 5, under "FILE-CONTROL Paragraph."

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

RECORD KEY Clause

The general format of this File Section 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. It is a group item or an elementary alphanumeric item. The maximum key length is 60 bytes and the key should never be made to contain all nulls.

If you specify random access mode, 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.

FILE STATUS Reporting

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

File Status Left	File Status Right	Meaning
0	0	Successful completion
1	0	EOF
2	1	Key not in sequence
2	2	Attempt to write a duplicate key
2	3	No record found
2	4	Disk space full
3	0	Permanent error
9	1	File structure destroyed

File Status "21" arises if ACCESS MODE is SEQUENTIAL, and

- You do not write to an indexed file in ascending sequence

  or

- A key is altered prior to processing a REWRITE statement

In an OPEN INPUT or OPEN I-O statement, a File Status of "30" means **File Not Found**.

File Status "91" occurs on an OPEN INPUT or OPEN I-O statement for a relative or indexed file whose structure has been destroyed (for example, by a system crash during output to the file). When this status is returned on an OPEN INPUT, the file is considered to be open, and READ statements may be processed. On an OPEN I-O, however, the file is not considered to be open, and all I/O operations fail. The other settings are self-explanatory.

Note that **Disk Space Full** occurs with Invalid Key (2) for indexed and relative file handling, whereas it occurs with **Permanent Error** (3) for sequential files.

If an error occurs when you run the program 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 is displayed on the screen, and the program ends.

Procedure Division Statements for Indexed Files

The syntax of the sequential file OPEN statement also applies to indexed files, except that EXTEND is not applicable.

The following table summarizes the available statement types and whether they are permissible with the ACCESS mode and the OPEN option in effect. Where X appears, the statement is permissible; otherwise, it is not valid under the associated ACCESS mode and OPEN option.

In addition to the following statements, CLOSE is permissible under all conditions; the same format shown for SEQUENTIAL files is used.

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

Figure 14. Procedure Statements for Indexed Files

CLOSE Statement

Purpose: The CLOSE statement causes the system to make the proper disposition of the file.

Format: `CLOSE` *filename* `[WITH LOCK]` ...

Remarks: You must use a CLOSE statement when the file has stopped processing. When a file is closed or has never been opened, you cannot READ from, REWRITE to, or WRITE to that file. Any of these three statements would cause a runtime error and make the program end.

If LOCK is used, the file cannot be reopened during the current job. If LOCK is not specified immediately after a *filename*, then that file may be reopened later in the program if the program logic requires it.

If you try to CLOSE a file that is not currently open, you get an error when you run the program, and the program stops abnormally.

Example:

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

DELETE Statement (Indexed I/O)

Purpose: The DELETE statement logically removes a record from an indexed file.

Format:

```
DELETE filename RECORD [INVALID KEY
    imperative-statement...]
```

Remarks: For a file in the sequential access mode, the last I/O statement performed for *filename* 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. Control passes to an applicable Declarative ERROR section if no INVALID KEY clause exists.

Example:

```
DELETE DISK-FILE RECORD.
```

DELETE Statement (Relative I/O)

Purpose: You use the DELETE statement to remove a record from a file.

Format: The format of the DELETE statement is the same for a relative file and an indexed file:

```
DELETE filename RECORD
    [INVALID KEY imperative-statement...]
```

Remarks: For a file in a sequential access mode, the immediately previous action would have been a successful READ statement; the record previously made available is logically removed from the file. If the previous READ was unsuccessful, an error occurs when you run the program, and the program ends. Therefore, an INVALID KEY phrase may not be specified for sequential access mode files.

For a file declared with dynamic or random access mode, 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.

Example:

```
DELETE DISK-FILE RECORD
    INVALID KEY
            DISPLAY "KEY ERROR"
            GO TO ERROR-RTN.
```

OPEN Statement

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

For an OUTPUT file, opening makes available a record area for development of one record, which is transmitted to the assigned output device when the WRITE statement is processed.

Format: `OPEN INPUT|I-O|OUTPUT|EXTEND` *`filename`*`... ...`

Remarks: The OPEN statement must be processed before you can perform input or output to a file.

The OPEN OUTPUT statement causes an existing file of the same name to be replaced by the file created with OPEN OUTPUT.

An OPEN I-O statement is valid only for a file assigned to a diskette. 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 diskette when the file is opened. It cannot be created by OPEN I-O.

When the EXTEND phrase is specified, the OPEN statement positions the file immediately after the last logical record of that file. The file must already exist on diskette. Subsequent WRITE statements referencing the file 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 or line sequential files.

Failure to precede (in terms of time sequence) file reading or writing by an OPEN statement is an error which causes the program to stop. 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 beginning of this chapter.

Example:

```
OPEN INPUT DISK-FILE, OUTPUT PRINT-FILE.
```

READ Statement (Indexed I/O)

Purpose: The READ statement accesses the data in your files.

Format: **Format 1 (Sequential Access):**

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

Format 2 (Random or Dynamic Access):

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

Remarks: Format 1 without NEXT must be used for all files having sequential access mode. Format 1 with the NEXT option is used for sequential reads of a file with dynamic access mode.

The AT END clause is performed 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 the action to be taken if the access key value does not refer to an existent key in the file. If you do not have an INVALID KEY clause, the appropriate Declaratives ERROR section, if supplied, is given control.

READ Statement (Indexed I/O)

The optional KEY IS clause must designate the record key item declared in the file's SELECT entry. This clause serves as documentation only. You must ensure that a valid key value is in the designated key field prior to performing a random access READ.

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

Example:

```
READ DISK-FILE NEXT RECORD
     AT END GO TO WRITE-REPORT.
READ DISK-FILE INTO WORK-AREA.
```

READ Statement (Relative I/O)

Purpose: You use the READ statement to access the data in your files.

Format: **Format 1:**

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

Format 2:

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

Remarks: 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 performed 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 the declared mode of access either random or dynamic.

If a Relative Key is defined (in the file's SELECT entry), 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.

READ Statement (Relative I/O)

For a Format 2 READ, the record that is retrieved is the one whose relative record number is prestored in the RELATIVE KEY item. If no such record exists, however, the INVALID KEY condition arises, and is handled by one of the following:

- The imperative statements given in the INVALID KEY portion of the READ
- An associated declaratives section

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

Example:

```
READ DISK-FILE NEXT RECORD
    AT END GO TO WRITE-REPORT.
READ DISK-FILE INTO STORAGE-PLACE
    INVALID KEY
            DISPLAY "INVALID KEY"
            GO TO ERROR-ROUTINE.
```

READ Statement (Sequential I/O)

Purpose: 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 is specified.

Format:

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

Remarks: Since at some time the end-of-file will always be encountered, you should always include the AT END clause. The reserved word END is followed by any number of imperative statements, all of which are performed 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 a declarative exists and no FILE STATUS item is specified for the file, a runtime I/O error is processed.

When a data record to be read exists, the program performs the sentence that follows the successful READ statement.

When more than one level 01 item is subordinate to a file definition, these records share the same storage area. Therefore, you 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 lets you specify that a copy of the data record is to be placed into a designated data field in addition to the file's record area. The *data-name* must not be defined in the File Section.

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*.

Diskette files occur as blocked input and output. For instance, a READ fills a physical buffer initially, and then additional READs may simply obtain the next logical record from the input buffer. The actual transmission of data from a diskette occurs as necessary.

If the actual record is shorter than the file record area, the file record area is padded on the right with spaces.

Example:

```
READ DISK-FILE INTO DATA-FIELD
    AT END GO TO WRITE-REPORT.
READ FILE-3.
```

REWRITE Statement (Indexed I/O)

Purpose: The REWRITE statement logically replaces an existing record.

Format:

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

Remarks: 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 record just previously read, then the invalid key condition exists and the imperative statements are processed, if they are present. Otherwise, an applicable Declaratives ERROR section is run, if one is 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.

Example:

```
REWRITE FILE-RECORD FROM HOLD-AREA.
```

REWRITE Statement (Relative I/O)

Purpose: You use the REWRITE statement to replace a record in a file.

Format: The format of the REWRITE statement is the same for a relative file and an indexed file:

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

Remarks: 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 the REWRITE. If the previous READ was unsuccessful, an error occurs when you run the program, and the program ends. Therefore, no INVALID KEY clause is allowed for sequential access.

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

Example:

```
REWRITE NAME-RECORD.
REWRITE NAME-RECORD FROM HOLD-RECORD
    INVALID KEY GO TO ERROR-RTN.
```

REWRITE Statement (Sequential I/O)

Purpose: The REWRITE statement replaces a logical record on a sequential DISK file.

Format:

```
REWRITE record-name [FROM data-name]
```

Remarks: *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.

When this statement is processed, the file to which *record-name* belongs must be open in the I-O mode.

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

REWRITE replaces the record that was accessed by the most recent successfully completed READ statement. If the record which you are rewriting to the file is longer than the file's record, only as many bytes as will fit are actually rewritten. On the other hand, if the record which you are rewriting to the file is shorter than the file's record, unpredictable information will be written after the record until the beginning of the next record in the file.

Example:

```
REWRITE NAME-RECORD FROM WS-HOLD.
```

START Statement (Indexed I/O)

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

Format:

```
START filename [KEY IS GREATER THAN|
      NOT LESS THAN|EQUAL TO data-name]
      [INVALID KEY imperative-statement...]
```

Remarks: *Data-name* must be the declared record key, and the value to be matched by a record in the file must be prestored in the *data-name*. When this statement is processed, 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 performed, or control goes to an appropriate Declaratives ERROR section.

Example:

```
START DISK-FILE KEY GREATER THAN KEY-VALUE.
```

START Statement (Relative I/O)

Purpose: You use the START statement to specify the beginning position for file reading operation.

Format: The format of the START statement is the same for a relative file and an indexed file:

```
START filename [KEY IS GREATER THAN|
     NOT LESS THAN|EQUAL TO data-name]
     [INVALID KEY imperative-statement...]
```

Remarks: 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 performed. When performing 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 performed, or control goes to an appropriate Declaratives ERROR section.

Example:

```
START DISK-FILE KEY EQUAL REL-KEY.
```

WRITE Statement (Indexed I/O)

Purpose: The WRITE statement releases a logical record for an output or input-output file.

Format:

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

Remarks: Just before the WRITE statement is processed, a valid (unique) value must be in that portion of the *record-name* (or *data-name-1* if FROM appears in the statement) that serves as RECORD KEY.

In the event of an improper key value, the imperative statements are performed if the INVALID KEY clause appears in the statement. Otherwise, an appropriate Declaratives ERROR section is started, if applicable. The INVALID KEY condition arises if one of these conditions exists:

- For sequential access, key values are not ascending from one WRITE to the next WRITE
- The key value is not unique
- The allocated disk space is exceeded

Example:

```
WRITE FILE-RECORD FROM DATA-AREA
    INVALID KEY GO TO GET-VALID-KEY.
```

WRITE Statement (Relative I/O)

Purpose: You use the WRITE statement to perform output to a file.

Format: 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...]
```

Remarks: 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 you must preset 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.

Example:

```
WRITE DATA-RECORD FROM HOLD-RECORD
    INVALID GO TO ERROR-ROUTINE.
```

WRITE Statement (Sequential I/O)

Purpose: You use the WRITE statement for file output.

Format:

```
WRITE record-name [FROM data-name-1]
      [{AFTER|BEFORE} ADVANCING
      {operand LINE(S)|PAGE}]
      [AT END-OF-PAGE|EOP imperative-statement]
```

Remarks: Depending on the device assigned, "written" output may take the form of printed matter or magnetic recording on a diskette. We remind you also that you use READ with *filename*, but you use WRITE with *record-name*. The associated file must be open in the OUTPUT mode at the time when the WRITE statement is processed.

Record-name must be one of the level 01 records defined for an output file, and may be qualified by the filename. 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 lets you 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.

When you try to write beyond the externally defined boundaries of a sequential file, a declarative procedure is performed (if available), and the FILE STATUS (if available) indicates a boundary violation. If neither is available, an error occurs when the program runs.

The ADVANCING option is restricted to line printer output files, and lets you control the line spacing on the paper in the printer. *Operand* is either an unsigned integer literal or a data name; values from 0 to 120 are permitted:

Integer	Carriage Control Action
0	No spacing
1	Normal (single spacing)
2	Double spacing
3	Triple spacing
•	•
•	•
•	•

Single spacing (that is, AFTER ADVANCING 1 LINE) is assumed if there is no BEFORE or AFTER option in the WRITE statement.

Use of the key word AFTER implies that the carriage control action precedes printing a line, whereas use of BEFORE implies that writing precedes the carriage control action.

If PAGE is specified, the data is printed BEFORE or AFTER the printer is repositioned to the next physical page. However, if a LINAGE clause is associated with the file, the repositioning is to the first line that can be written on the next logical page as specified in the LINAGE clause.

If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file description entry for the associated file. EOP means END-OF-PAGE.

An *end-of-page* condition is reached when a WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the footing area of a page body. This occurs when such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the FOOTING value, if specified. In this case, after the WRITE statement is processed, the imperative statement in the END-OF-PAGE phrase is processed.

A *page overflow* condition is reached whenever a WRITE statement cannot be fully accommodated within the current page body. This occurs when a WRITE statement would cause the LINAGE-COUNTER to exceed the value specified as the size of the page body in the LINAGE clause. In this case, the record is printed before or after (depending on the phrase used) the printer is repositioned to the first line of the next logical page. The imperative statement in the END-OF-PAGE clause, if specified, is processed after the record is written and the printer has been repositioned.

Clearly, if no FOOTING value is specified in the LINAGE clause, or if the end-of-page and overflow conditions occur simultaneously, then only the overflow condition is effective.

Example:

```
WRITE PRINT-LINE FROM DATA-FIELD AFTER 2 LINES.
WRITE RECORD-1.
```

CHAPTER 9. TABLE HANDLING BY THE INDEXING METHOD

Contents

Index Names and Index Items

In addition to the capabilities of subscripting described in Chapter 3, IBM COBOL provides the indexing method of table handling. You define a table in the Data Division by using the OCCURS clause.

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* part of an OCCURS clause. An index name must be unique.

An *index data item* is an item defined by the USAGE IS INDEX phrase. An index data item must not have a PICTURE.

An index name or index data item may only be modified by:

- A SET statement
- A SEARCH statement
- A CALL statement's USING list with the corresponding PROCEDURE HEADER USING list in the subprogram

An index name or index data item may be used:

- In a relation condition
- As the variation item in a PERFORM VARYING statement
- In place of a subscript

In all cases, the process is equivalent to dealing with a binary word integer subscript. You must initialize (by using SET, SEARCH, or PERFORM) *index-name* to some value before you use it.

When you refer to an item in a table controlled by an OCCURS clause, the reference 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)
```

The subscripts can be:

- Integer decimal items
- Integer constants
- Binary integer (COMPUTATIONAL-0 or INDEX) items
- Index names

Subscripts may be qualified, but not subscripted themselves. A subscript may be signed; but if it is, 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 (up to a maximum of 1023) of the item as specified in its OCCURS clause.

Relative Indexing

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. For example:

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

SEARCH Statement–Format 1

Purpose: You use the SEARCH statement to perform a linear search of a table.

Format:

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

Remarks: *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.

The four possible uses of VARYING are:

- NO VARYING phrase: the first-listed *index-name* for the table is varied.
- 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.
- VARYING *index-name* defined for table: this specific *index-name* is the only one varied.
- VARYING *integer-data-item-name*: both this *data-item* and the first-listed *index-name* for table are varied, simultaneously.

Interpretation of the term VARYING follows these steps:

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 ends at once; and if an AT END phrase exists, the associated *imperative-statement-1* is performed.

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 performed and the SEARCH operation ends. 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 the four uses of VARYING stated above.

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-dimensional or three-dimensional table, a SEARCH must be performed several times with the other *index-names* set appropriately each time, probably with a PERFORM, VARYING statement.

Example:

```
SEARCH DATA-TABLE
    AT END GO TO WRITE-REPORT
    WHEN ID-NUMBER = ID-CODE (INDEX-VALUE)
    GO TO PROCESS-DATA.
```

SEARCH Statement–Format 2

Purpose: Format 2 SEARCH statements deal with tables of *ordered* data.

Format:

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

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

1. Only a simple relational *condition* or *condition-name* may be used, 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).

   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 a part of 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, subscripts, and indexes.)

If you do not follow these rules, you may get unpredictable results. Unpredictable results also occur if the table data is not ordered in conformance with 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 sentence if there is no AT END clause.

SEARCH Statement–Format 2

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.

Example:

```
SEARCH ALL PRODUCT-TABLE
    AT END GO TO WRITE-REPORT
    WHEN PRODUCT-NO = PRODUCT-CODE (INDEX-1)
        GO TO PROCESS-DATA.
```

Purpose: The SET statement lets you change index names, index items, or binary subscripts for table handling purposes.

Format: The two formats are:

Format 1:

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

Format 2:

```
SET index-name-3 ... UP BY|DOWN BY
    data-name-4|integer-4
```

Remarks: Format 1 is equivalent to moving the TO value (that is, *integer-2*) to multiple receiving fields written immediately after the verb SET.

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

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

Example:

```
SET INDEX-1 TO 1.
SET INDEX-2 UP BY 1.
```

CHAPTER 10. INTERPROGRAM COMMUNICATION

Contents

How Communication Is Handled

Separately compiled COBOL program modules may be combined into one single program. Interprogram 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 program chaining facility allows a COBOL program to transfer control to another program and, optionally, to pass data items as parameters to the chained program.

Assembler Subroutines

It is also possible for an IBM COBOL program to call assembler subroutines. (Refer to *IBM Personal Computer MACRO Assembler* for instructions on writing assembly language programs.) The IBM COBOL runtime system transfers execution to a subroutine by means of a machine language FAR CALL instruction. Execution should return via the MACRO Assembler RET instruction.

Parameters are passed by reference (that is, by passing the address of the parameter). Parameter addresses are passed on the stack.

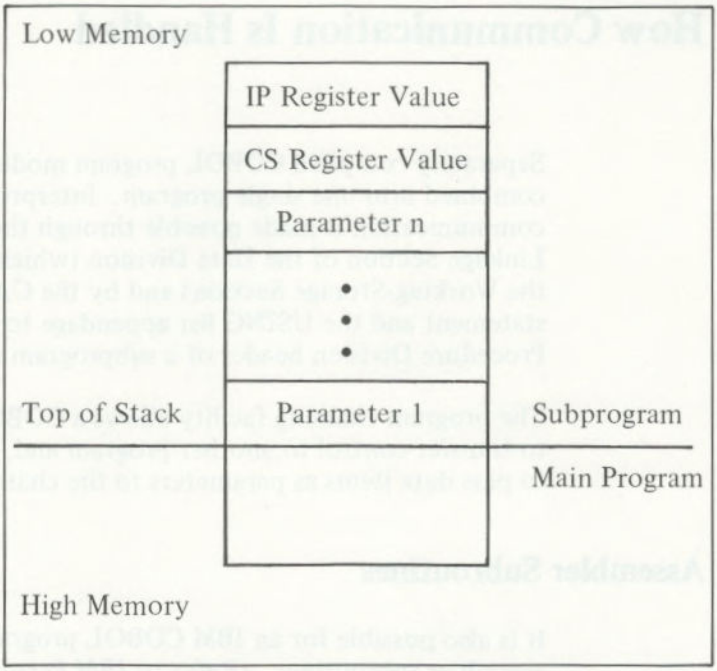


Figure 15. Contents of Stack at Entry to a Routine

The called routine must preserve the BP register contents and remove the parameter addresses from the stack before returning.

The subroutine can expect only as many parameters as are passed, and the calling program is responsible for passing the correct number of parameters. Neither the compiler nor the runtime system checks for the correct number of parameters. It is entirely up to you to determine that the type and length of arguments passed by the calling program are acceptable to the called subroutine. Numeric values must be passed as binary (COMP-0).

The stack space used by an IBM COBOL program is contained within the program boundaries, so assembler programs that use the stack must not overflow or underflow the stack.

The most certain way to assure safety is to save the COBOL stack pointer upon entering the routine, and to set the stack pointer to another stack area. The assembler routine must then restore the saved COBOL stack pointer before returning to the main program.

To call a subprogram, use the name of the subprogram in the COBOL CALL statement. The name of an assembler subprogram is defined by a PUBLIC pseudo-op and is declared as PROC FAR. (The name of a COBOL subprogram is the name entered in the PROGRAM-ID paragraph.) Then link the subprogram to the main program using the IBM Personal Computer Linker, as described in Chapter 3 and in Appendix C, "The Linker (LINK) Program."

Example

COBOL Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.EXAMPLE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  PARM1          PIC 99 COMP-0 VALUE 45.
77  PARM2          PIC 99 COMP-0 VALUE 50.
77  PARM3          PIC 99 COMP-0 VALUE 0.
77  PAR1           PIC 99.
77  PAR2           PIC 99.
77  PAR3           PIC 99.
PROCEDURE DIVISION.
THIS-IS-IT.
    CALL 'ADDIT' USING PARM1, PARM2, PARM3.
    MOVE PARM1 TO PAR1.
    MOVE PARM2 TO PAR2.
    MOVE PARM3 TO PAR3.
    DISPLAY PAR1 ' + ' PAR2 ' = ' PAR3.
    STOP RUN
```

```
        assume  cs:codeseg
parm    struc                   ;stack definition
savebp  dw      ?               ;saved caller's bp
        dw      ?               ;caller's ip reg
        dw      ?               ;caller's cs reg
parm3   dw      ?               ;addr 3rd parameter
parm2   dw      ?               ;addr 2nd parameter
parm1   dw      ?               ;addr 1st parameter
        parm    ends

codeseg segment para
        public  addit           ;entry point
addit   proc    far             ;long call
        push    bp              ;save bp of caller
        mov     bp,sp           ;set up stack frame
        mov     bx,[bp].parm1   ;get addr of parm1
        mov     ax,[bx]         ;put value in ax
        mov     bx,[bp].parm2   ;get addr of parm2
        add     ax,[bx]         ;add values
        mov     di,[bp].parm3   ;get addr of parm3
        mov     [di],ax         ;put result into parm3
        pop     bp              ;restore caller's bp
        ret     6               ;restore stack
addit   endp
codeseg ends
        end
```

Chain Parameters

The parameters that are passed between the programs with a CHAIN USING statement are stored at the highest available memory address. The memory layout follows, starting at the highest available address and proceeding toward location zero. (See Figure 16.)

1. 32 bytes are reserved for stack space.

2. The length of the first parameter in the USING list is stored in two bytes, high-order byte first.

3. The parameter is stored as a string of bytes in the same order as they were stored in the Data Division, beginning at the address of the length minus the length itself.

4. Each parameter in the USING list follows, in order, each preceded by its length.

The chained program must expect the same number and format of parameters as were passed, as the compiler and runtime system cannot check the number or format.

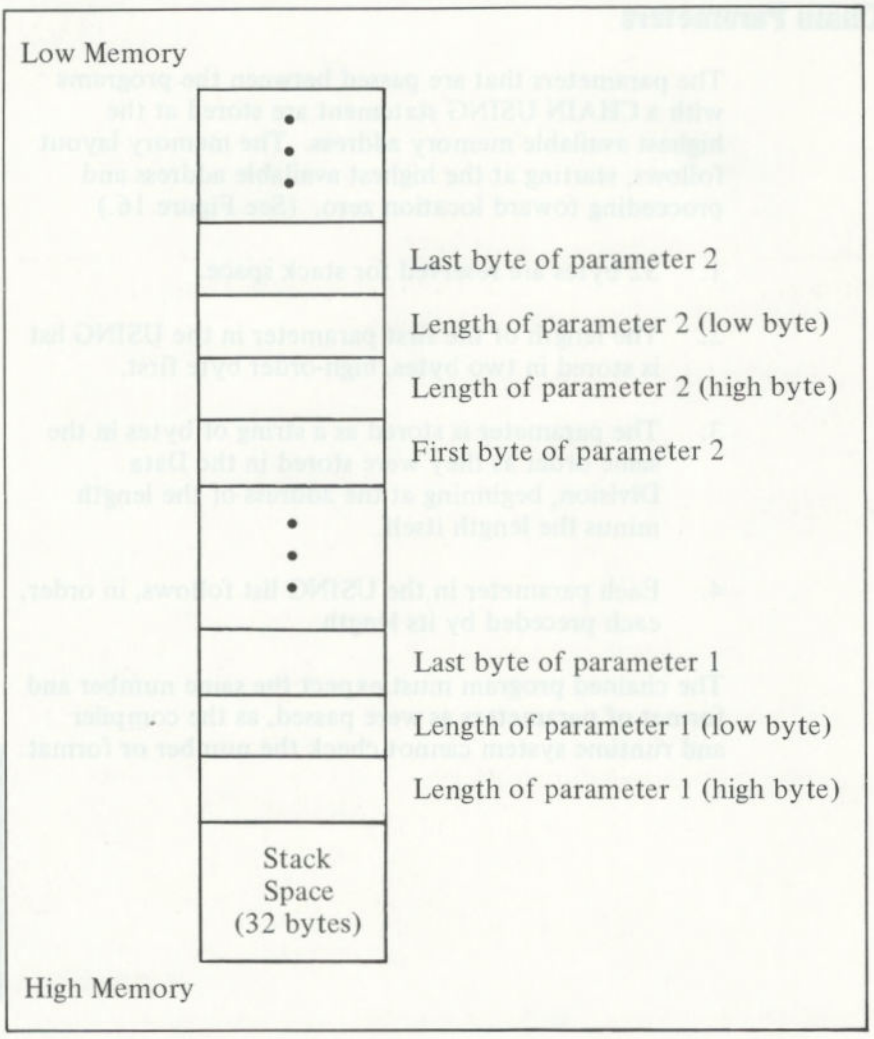


Figure 16. Memory Layout When Chaining Programs

CALL Statement

Purpose: The CALL statement allows a COBOL program to transfer control to a subprogram.

Format:

```
CALL literal [USING data-name ...]
```

Remarks: *Literal* is a subprogram name defined as the PROGRAM-ID of a separately compiled program; *literal* is nonnumeric.

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 lists is by position, not by identical spelling of names.

Example:

```
CALL "SUBPRG" USING PARM1, PARM2, PARM3.
CALL "COBSUB".
```

CHAIN Statement

Purpose: The CHAIN statement allows a COBOL program to transfer control to any other program and, optionally, to pass data items as parameters to the chained program.

Format:

```
CHAIN literal|identifier-1
    [USING identifier-2...]
```

Remarks: *Literal* and *identifier-1* must be alphanumeric, and *identifier-1* must contain a terminating space. Each occurrence of *identifier-2* must be defined in the Working-Storage or Linkage Section or in the record area of a file open at the time the CHAIN statement is processed.

When the CHAIN statement is processed, the value of *literal* or *identifier-1*, up to but not including the first space encountered (or the end of the *literal*), is interpreted as the name of a DOS format program file. The named program is loaded into memory and run. All program and data structures of the chaining program are lost, except that the USING clause may be used to transfer parameters to the chained program.

The chained program need not be a COBOL program. If it is, it must be a main program.

Example:

```
CHAIN "NEXTFILE.COB".
CHAIN MENU USING EGGS, HAM, O-J.
```

EXIT PROGRAM Statement

Purpose: The EXIT PROGRAM statement, appearing in a called subprogram, causes control to be returned to the next statement after CALL in the calling program.

Format:

```
EXIT PROGRAM.
```

Remarks: This statement must be a paragraph by itself. If it appears in a main program, it causes no action.

Example:

```
EXIT PROGRAM.
```

LINKAGE Section

Purpose: The Linkage Section describes data made available in memory from another program module.

Format: 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.

Remarks: 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 the calling program. The Linkage Section occurs in the called program where the CALL statement appears in the calling program.

Example:

```
LINKAGE SECTION.
01 PARAMETER-1    PIC 999.
01 PARAMETER-2    PIC 999.
01 PARAMETER-3    PIC 999.
```

PROCEDURE DIVISION Header with CALL and CHAIN

Purpose: This header describes the linkage and parameter initialization requirements of a called or chained program.

Format: The header of a chained main program is coded as:

```
PROCEDURE DIVISION [CHAINING data-name-1...].
```

Data-name-1 must be in the Working-Storage Section of your program.

The header of a subprogram is coded as:

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

Data-name-2 must be in the Linkage Section of your program.

Remarks: A *main program* must be linked by itself or with any number of subprograms. It may then be run independently or started by a CHAIN statement in another program.

A *subprogram* must be linked with exactly one main program and, optionally, any number of other subprograms. You can run a subprogram only by using the CALL statement. For a description of the linking process, see Appendix C, "The Linker (LINK) Program."

PROCEDURE DIVISION Header with CALL and CHAIN

A chained or called program should have a chaining list or USING list if and *only* if the invoking CHAIN or CALL statement has a USING list. The USING list must contain at least one item. Also, the numbers of entries in the lists should be equal, and entries with corresponding positions in the two lists should reference data items of the same size and USAGE. Failure to conform to these rules cannot be diagnosed by the compiler and can cause unpredictable results when the program runs.

The values of the data items named in the Procedure Division header are established when the program is initialized. They are established by using the contents of corresponding data items named in the invoking CALL or CHAIN statement. In the case of CALL, the identification is made by passing pointers. Therefore, if the value of a data item named in a Procedure Division USING clause is changed while the subprogram is running, the corresponding data item in the calling program reflects the change after control is returned from the subprogram.

APPENDIXES

Contents

APPENDIX A. COBOL ERROR MESSAGES

This appendix lists all of the error messages which you may encounter while you are compiling and running an IBM COBOL program. Each message is accompanied by a brief explanation of what caused the error.

The messages are organized in two sections:

- Compile time errors
- Runtime errors

Compile Time Errors

Two different types of errors can be detected by your IBM Personal Computer during compilation:

- Command input errors and errors caused when a DOS-dependent I/O operation encounters a problem.
- Syntax errors in the COBOL program.

Command Input and DOS-dependent I/O Errors

The following messages may be displayed whenever the error occurs during the compile. Each of the seven messages is followed here by an additional explanation. When you get one of these messages, you should correct the problem and retry the compile.

Message	Explanation
?Bad filename	A filename is not constructed according to the rules of DOS.
?File not found	You have specified a filename for input that does not exist.
?Bad switch:/X	You have entered a slash parameter ("/") that the compiler does not recognize.
?Command error:'X'	You have an invalid entry (X) in the command line.
?Can't create file	An output file cannot be opened.
?Disk X full	The diskette in the specified drive is full. If X is blank, it refers to the default drive.
?Overlay n not found	One of the COBOL compiler overlay files (COBOLn.OVR) is not present on the diskette.
?Memory Full	See explanation following.
?Compiler Error	See explanation following.

Two error messages that occur infrequently and are also displayed on the console must be noted. One is **?Memory Full**. This occurs when there is insufficient memory for all the symbols and other information the compiler obtains from your source program. It indicates that the program is too large and must be decreased in size or split into separately compiled modules.

The symbol table of data names and procedure names is usually the largest user of space during compilation. All names require as many bytes as there are characters in the name, and there is an overhead requirement of about 10 bytes per data-name and 2 bytes per procedure-name. On the average, each line in the Data Division requires about 14 bytes of memory during compilation, and each line in the Procedure Division requires about 3-1/4 bytes.

The other error message is the following: **?Compiler Error in Phase n at address**. It occurs when the compiler becomes confused. It is usually caused by one of four problems:

- The source program is incorrect. You can sometimes determine the cause by compiling increasingly larger chunks of your program, starting with only a few lines, until the error recurs.
- The source program diskette is damaged.
- The compiler or one of the overlay files has been damaged. In this case, you should try your backup copy.
- A stack overflow may cause this error. You can try the /P option to correct this type of error.

With both of these error messages, compilation stops immediately.

Diagnostic messages are listed at the bottom of a compiled program listing and also on the screen. They consist of two parts:

1. The associated source line number or file occurrence – four digits, followed by a colon (:).

2. An English explanation of the error detected by the compiler. If this text begins with an /F/ or a /W/, then it is only a warning; if not, it is an error sufficiently severe to prevent you from linking and running an object program.

Regardless of whether a listing has been requested, the errors/warnings (if any) are always listed on the screen at the end of compilation. Also, a message displaying the total number of errors/warnings is displayed. This allows you to make a simple change to a COBOL program, recompile it without a listing and still know whether the compiler encountered any questionable statements in the program.

The following diagnostic messages are issued by the compiler. The messages are listed in alphabetical order, with the /F/ and /W/ warnings alphabetized at the end of the rest of the compilation messages. When a /F/ message appears, the line number associated with it represents the order of files as found in the File Section.

A FILE-ID NAME IS UNDEFINED.

A data name specified in a VALUE OF FILE-ID clause is not defined.

A PARAGRAPH DECLARATION IS REQUIRED HERE.

An EXIT statement is not followed by a section or paragraph header.

AREA A NOT BLANK IN CONTINUATION LINE.

A character was encountered in Area A.

AREA-A VIOLATION; RESUMPTION AT NEXT PARAGRAPH/SECTION/DIVISION/VERB.

The entry starting in one of columns 8-12 cannot be interpreted as a division header, section name, paragraph name, file description indicator, or 01 or 77 level number.

CLAUSES OTHER THAN VALUE DELETED.

The data description of a level 88 item includes a descriptive clause other than VALUE IS.

ELEMENT LENGTH ERROR.

The length of the quoted literal is over 120 characters, or the numeric literal is over 18 digits, or the identifier/name is over 30 characters.

ERRONEOUS FILENAME IS IGNORED.

An entry which has not been declared as a filename appears where a filename is required.

ERRONEOUS QUALIFICATION; LAST DECLARATION USED.

The qualifiers used with a data name are incorrect or not unique.

ERRONEOUS RERUN-ENTRY IS IGNORED.

A RERUN clause of the I-O-CONTROL paragraph contains a syntax error.

ERRONEOUS SUBSCRIPTING; STATEMENT DELETED.

Too few or too many subscripts are provided for a data name.

EXCESSIVE LITERAL POOL OR DISPLAY STRING LENGTH.

The total length of the literals contained within a single paragraph is greater than 4096 bytes.

EXCESSIVE NUMBER OF FILES/4KB WORKING-STORAGE BLOCKS.

The sum of (number of files declared) + (size of WORKING-STORAGE divided by 4KB and rounded up) + (number of level 01 and level 77 entries in the LINKAGE SECTION) is greater than 14.

EXCESSIVE OCCURS NESTING IS IGNORED.

OCCURS clauses are nested more than three deep.

EXCESSIVE SEGMENT NUMBER.

A section header contains a section number greater than 99.

EXCESSIVE SEGMENT NUMBER IN DECLARATIVES.

A section header in the DECLARATIVES region contains a section number greater than 49.

FILE NOT SELECTED; ENTRY BYPASSED.

An FD is given for a filename which does not appear in any SELECT sentence.

FILL CHARACTER CONFLICT.

In a Format 3 ACCEPT statement, SPACE-FILL and ZERO-FILL are both specified.

FRACTIONAL EXPONENT OR NEGATIVE SCALED BASE (99P).

In a COMPUTE statement, an exponent is a numeric literal with a decimal point or a numeric data item described with a digit to the right of an assumed decimal point, or the PICTURE of an exponentiation base (entry preceding **) contains the character **P** as the rightmost digit.

GROUP ITEM, THEREFORE PIC/JUST/BLANK/ SYNC IS IGNORED.

A phrase which is only allowed for elementary data items is used in the description of an item which is followed immediately by an item of a higher level number.

GROUP SIZE GREATER THAN 4095; LENGTH SET TO 1.

The size of an item at a level other than 01 is declared to be greater than 4095 bytes.

ILLEGAL CHARACTER.

An invalid character has been encountered.

ILLEGAL COPY FILENAME.

The filename for the copy file is in error.

ILLEGAL MOVE OR COMPARISON IS DELETED.

The operands of a MOVE statement or relational condition are of incompatible class.

IMPERATIVE STATEMENT REQUIRED. STATEMENT DELETED.

A conditional statement is contained within a conditional statement other than IF.

IMPROPER CHARACTER IN COLUMN 7.

An invalid character in column 7 has been encountered.

IMPROPER PICTURE. PIC X ASSUMED.

An invalid PICTURE clause has been encountered.

IMPROPER PUNCTUATION.

Incorrect punctuation has been encountered. For instance, a comma or period must be followed by a space.

IMPROPER REDEFINITION IGNORED.

The data name specified in a REDEFINES clause is not at the same level as the current data name, or it is separated from it by an item with a lower level number.

IMPROPERLY FORMED ELEMENT.

Incorrect syntax for an item has been encountered. For instance, you could have used multiple decimal points in a numeric literal.

INCOMPLETE (OR TOO LONG) STATEMENT DELETED.

A verb immediately follows a partial statement form, or an otherwise acceptable statement is too large for the compiler to read.

INVALID KEY SPECIFICATION.

The key item for a relative or indexed file should not be subscripted, or it is inconsistent with the file organization in class or USAGE.

INVALID QUOTED LITERAL.

A literal of zero length, improper construction, or missing end quotes has occurred.

INVALID RECORD SIZE(S) IGNORED.

The RECORD clause of an FD contains an error.

INVALID SELECT-SENTENCE.

The syntax of a SELECT sentence in the FILE-CONTROL paragraph is incorrect.

INVALID VALUE IGNORED.

The value specified in a VALUE IS phrase is not a properly formed literal.

JUSTIFICATION CONFLICT.

In a Format 3 ACCEPT statement, LEFT-JUSTIFY and RIGHT-JUSTIFY are both specified.

KEY DECLARATION OF THIS FILE IS NOT CORRECT.

The RELATIVE KEY clause is missing for a relative file, or the RECORD KEY clause is missing for an indexed file.

KEYS MAY ONLY APPLY TO AN INDEXED/ RELATIVE FILE.

A RECORD KEY or RELATIVE KEY clause was specified for a file with sequential or line sequential organization.

LITERAL TRUNCATED TO SIZE OF ITEM.

The literal specified in a VALUE IS phrase is larger than the data item being declared.

MISORDERED/REDUNDANT SECTION PROCESSED AS IS.

A section in the Identification, Environment, or Data Division is out of order or repeated.

NAME OMITTED; ENTRY BYPASSED.

The data name is missing in a data description entry.

NON-CONTIGUOUS SEGMENT DISALLOWED.

Two sections with the same number, larger than 49, are separated by one or more sections with a different number.

NO PICTURE; ELEMENTARY ITEM ASSUMED TO BE BINARY.

No PICTURE is given for an elementary data item.

OCCURS DISALLOWED AT LEVEL 01/77, OR COUNT TOO HIGH.

An OCCURS clause appears in a data description entry at level 01 or 77, or the number of occurrences specified is greater than 1023.

OMITTED WORD 'SECTION' IS ASSUMED HERE.

The required word SECTION is missing from the header of a section in the Data Division.

PROCEDURE-NAME IS UNRESOLVABLE.

A reference appears to a section name or procedure name which is not sufficiently qualified or is not unique.

PROCEDURE RANGE NOT IN CURRENT SEGMENT.

A PERFORM statement in a section with a number greater than 49 refers to a procedure in a section with a different number greater than 49.

PROCEDURE RANGE SPANS SEGMENTS.

A procedure range (procedure-name-1 THRU procedure-name-2) mentioned in a PERFORM statement contains paragraphs in sections with different section numbers greater than 49, or in sections numbered both less than or equal to 49 and greater than 49.

REDUNDANT FD PROCESSED AS IS.

The same filename appears in more than one file description.

REWRITE VALID ONLY FOR A DISK FILE.

The filename entry in a REWRITE statement is a file assigned to PRINTER.

SEMANTICAL ERROR IN SCREEN DESCRIPTION.

This message can be caused in five different ways:

- The SCREEN SECTION does not begin with a level 01 screen item description.
- A level 01 screen item description does not include a screen name.
- A group screen item is described with a clause which is allowed only for elementary items.
- An elementary screen item description is missing FROM, TO, USING, or VALUE.
- A screen item description contains inconsistent clauses (such as USING and VALUE).

SIGN CLAUSE IGNORED FOR UNSIGNED ITEM.

The PICTURE of a numeric item with USAGE IS DISPLAY describes it as unsigned, but a SIGN IS clause is present.

SINGLE-SPACING ASSUMED DUE TO IMPROPER ADVANCING COUNT.

The operand of the BEFORE or AFTER phrase of a WRITE statement is greater than 120.

SOURCE BYPASSED UNTIL NEXT FD/SECTION.

An error in a file description prevents further analysis.

STATEMENT DELETED BECAUSE INTEGRAL ITEM IS REQUIRED.

- A numeric data item whose PICTURE specifies digits to the right of the decimal point is used where an integer is required.

STATEMENT DELETED BECAUSE OPERAND IS NOT A FILENAME.

A name appearing where a filename is required is not declared as a filename.

STATEMENT DELETED DUE TO ERRONEOUS SYNTAX.

A syntax error is present, to which no more specific message applies.

STATEMENT DELETED DUE TO NON-NUMERIC OPERAND.

An alphanumeric or alphanumeric-edited item is used as an operand of an arithmetic statement, a numeric-edited item is used as an operand other than the result, or a number is greater than 18 digits long.

SUBSCRIPT 0 OR OVER MAX. NO. OCCURRENCES; 1 USED.

A literal used as a subscript is inconsistent with the range defined by the associated OCCURS clause.

SUBSCRIPT OR INDEX-NAME IS NOT UNIQUE.

A name which requires qualification is used as a subscript.

SYNTAX ERROR IN SCREEN DESCRIPTION.

A screen item description contains a clause which is unrecognizable, improperly constructed, or redundant.

UNRECOGNIZABLE ELEMENT IS IGNORED.

A required keyword is missing, or a data name or procedure name is unidentified.

USING-LIST ITEM LEVEL MUST BE 01/77.

A name used in the PROCEDURE DIVISION header USING list is not declared at level 01 or level 77.

VALUE DISALLOWED–OCCURS/REDEFINES/ TYPE/SIZE CONFLICT.

The VALUE IS clause is specified for a data item described with (or included within an item described with) an OCCURS or REDEFINES clause; or, the literal given in a VALUE IS clause is not compatible with the PICTURE of the declared item.

VALUE OF FILE-ID REQUIRED.

The VALUE OF FILE-ID clause is not specified in the file description of a file assigned to DISK.

VARYING ITEM MAY NOT BE SUBSCRIPTED.

The data item controlled by the VARYING phrase of a PERFORM statement is subscripted.

/F/ FILE NEVER CLOSED.

No CLOSE statement is present for the file.

/F/ FILE NEVER OPENED.

No OPEN statement is present for the file.

/F/ INCONSISTENT READ USAGE.

An OPEN INPUT statement is present for a file, but no READ statement, or vice versa.

/F/ INCONSISTENT WRITE USAGE.

An OPEN OUTPUT statement is present for a file, but no WRITE statement, or vice versa.

/W/ BLANK WHEN ZERO IS DISALLOWED.

The BLANK WHEN ZERO phrase appears in the description of an alphanumeric or alphanumeric-edited item.

/W/ DATA DIVISION ASSUMED HERE.

The DATA DIVISION header is missing.

/W/ DATA RECORDS CLAUSE WAS INACCURATE.

The record name(s) given in a DATA RECORDS clause are not consistent with the record descriptions following the file description.

/W/ FD-VALUE IGNORED SINCE LABELS ARE OMITTED.

The VALUE OF FILE-ID clause is used in the description of a file which is assigned to PRINTER.

/W/ FILE SECTION ASSUMED HERE.

The FILE SECTION header is missing.

/W/ INVALID BLOCKING IS IGNORED.

The BLOCK clause of an FD contains an error.

/W/ 'LABEL RECORD STANDARD' REQUIRED.

The LABEL RECORD(S) STANDARD phrase is not present in the FD of a file assigned to DISK.

/W/ LABEL RECORDS OMITTED ASSUMED FOR PRINTER FILE.

The LABEL RECORDS OMITTED clause is missing in the file description of a file assigned to PRINTER.

/W/ LEVEL 01 ASSUMED.

A record description begins with a level number other than 01.

/W/ PERIOD ASSUMED AFTER PROCEDURE-NAME DEFINITION.

A section or paragraph header does not end with a period.

/W/ PICTURE IGNORED FOR INDEX ITEM.

A data item described with USAGE IS INDEX phrase also has a PICTURE phrase.

/W/ PROCEDURE DIVISION ASSUMED HERE.

The PROCEDURE DIVISION header is missing.

/W/ RECORD MAX DISAGREES WITH RECORD CONTAINS; LATTER SIZES PREVAIL.

The record size specified in the RECORD CONTAINS clause of an FD is inconsistent with the sizes of the associated record descriptions.

/W/ REDUNDANT CLAUSE IGNORED.

The same clause is specified more than once.

/W/ RIGHT PARENTHESIS REQUIRED AFTER SUBSCRIPTS.

The closing parenthesis for a subscript is missing.

/W/ TERMINAL PERIOD ASSUMED ABOVE.

A data description entry or paragraph does not end with a period.

/W/ WORKING-STORAGE ASSUMED HERE.

The WORKING-STORAGE header is missing.

Runtime Errors

Some programming errors cannot be detected by the compiler but cause the program to end prematurely when you are running it. Each of these error messages is accompanied by the name (PROGRAM-ID) of the module and, if object line numbers are present (compile time option), the compilation listing number of the statement that was being processed when the program stopped. This information is displayed in the following format:

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

The possible reasons for the program to end, with additional explanation, are listed below.

Message	Explanation
REDUNDANT OPEN	Attempt to open a file that is already open.
DATA UNAVAILABLE	Attempt to reference data in a record of a file that is not open or has reached the AT END condition.

Message	Explanation
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, FILE STATUS item or Declaratives Section.
NON-NUMERIC DATA	Whenever the content of a numeric item does not conform to the given PICTURE, this condition may arise. You 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 PERFORMs, as for example, when paragraph A is performed, and prior to exiting from it another PERFORM A is initiated.
ILLEGAL READ	Attempt to READ a file that is not open in the INPUT or I-O mode.

Message	Explanation
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 mode file when the last operation was not a successful READ.
OBJ. CODE ERROR	An undefined object program instruction has been encountered. This should occur only if the absolute version of the program has been damaged in memory or on the disk file.
GO TO (NOT SET)	An attempt is made to execute a null GO statement which has never been altered to refer to a destination.
FILE LOCKED	Attempt to OPEN after earlier CLOSE WITH LOCK.
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.

Message	Explanation
ILLEGAL START	File not opened for INPUT or I-O.
SEG nn LOAD ERR	An error occurred while attempting to load overlay segment 49+dd, where **dd** is the decimal equivalent of **nn** interpreted as a hexadecimal number.
NEED MORE MEMORY	The indexed file manager has ended abnormally because of insufficient dynamically allocatable memory.

The following error messages start with ****COBOL**: to distinguish them from similar DOS error messages.

Message	Explanation
****COBOL: FILE 'filename' NOT FOUND. ENTER NEW DRIVE LETTER:**	The chained file, segment file, or common runtime file could not be found.
****COBOL: PROGRAM TOO BIG TO FIT IN MEMORY.**	There is insufficient memory available to load chained program or common runtime file.
****COBOL: ERROR IN EXE FILE.**	Error in loading chained or common runtime EXE file.

APPENDIX B. RESERVED WORDS

A plus sign (+) indicates additional words required by IBM Personal Computer COBOL for interactive screens, debug extensions, and packed decimal format.

A solid box (■) indicates Standard COBOL reserved words which are not reserved in IBM COBOL. For compatibility, you should avoid using these words as filenames, data names, and variables in your programs.

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
■ALSO
ALTER
■ALTERNATE
AND
ARE
AREA(S)
ASCENDING
+ASCII
ASSIGN
AT
AUTHOR
AUTO
+AUTO-SKIP

+BACKGROUND-COLOR
+BEEP
BEFORE
BELL
BLANK
BLINK
BLOCK
BOTTOM
BY

CALL
■CANCEL
■CD
■CF
■CH
CHAIN
CHAINING
CHARACTER(S)
■CLOCK-UNITS
CLOSE
■COBOL
■CODE
CODE-SET
+COL
COLLATING
COLUMN
COMMA
■COMMUNICATION
COMP
COMPUTATIONAL
COMPUTATIONAL-0
+COMPUTATIONAL-3
COMPUTE
COMP-0
+COMP-3

CONFIGURATION
CONTAINS
■CONTROL(S)
COPY
■CORR(ESPONDING)
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
■DE(TAIL)
DEBUGGING
■DEBUG-CONTENTS
■DEBUG-ITEM
■DEBUG-NAME
■DEBUG-SUB-1
■DEBUG-SUB-2
■DEBUG-SUB-3
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
■DESTINATION
■DISABLE
+DISK
DISPLAY
DIVIDE
DIVISION
DOWN
■DUPLICATES
DYNAMIC

■EGI
ELSE
■EMI
+EMPTY-CHECK
■ENABLE
END
END-OF-PAGE
■ENTER
ENVIRONMENT
EOP
EQUAL
+ERASE
ERROR
ESCAPE
■ESI
■EVERY
EXCEPTION
+EXHIBIT
EXIT
EXTEND

FD
FILE
FILE-CONTROL
+FILE-ID
FILLER
■FINAL
FIRST
FOOTING
FOR
+FOREGROUND-COLOR
FROM
+FULL

■GENERATE
GIVING
GO
GREATER
■GROUP

■HEADING
+HIGHLIGHT
HIGH-VALUE(S)

IDENTIFICATION
IF
IN
INDEX
INDEXED
■INDICATE
INITIAL
■INITIATE
INPUT
INPUT-OUTPUT

INSPECT
INSTALLATION
INTO
INVALID
IS
I-O
I-O-CONTROL

JUST(IFIED)

KEY

LABEL
■LAST
LEADING
LEFT
+LEFT-JUSTIFY
■LENGTH
+LENGTH-CHECK
LESS
■LIMIT(S)
+LIN
LINAGE
LINAGE-COUNTER
LINE(S)
■LINE-COUNTER
LINKAGE
LOCK
LOW-VALUE(S)

MEMORY
MERGE
■MESSAGE
MODE
MODULES
MOVE
■MULTIPLE
MULTIPLY

+NAMES
NATIVE
NEGATIVE
NEXT
■NO
+NO-ECHO
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
PROCEDURE(S)
PROCEED
PROGRAM
PROGRAM-ID
+PROMPT

■QUEUE
QUOTE

RANDOM
■RD
READ
+READY
■RECEIVE
RECORD(S)
REDEFINES
■REEL

■REFERENCES
RELATIVE
RELEASE
■REMAINDER
REMOVAL
■RENAMES
REPLACING
■REPORT(S)
■REPORTING
+REQUIRED
RERUN
RESERVE
RESET
RETURN
■REVERSED
+REVERSE-VIDEO
■REWIND
REWRITE
■RF
■RH
RIGHT
+RIGHT-JUSTIFY
ROUNDED
RUN

SAME
SCREEN
■SD
SEARCH
SECTION
SECURE
SECURITY
■SEGMENT
■SEGMENT-LIMIT
SELECT
■SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
■SOURCE
SOURCE-COMPUTER
SPACE(S)
+SPACE-FILL
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
STRING
■SUB-QUEUE-1,2,3
SUBTRACT
■SUM
■SUPPRESS
+SWITCH-1
+SWITCH-2
+SWITCH-3
+SWITCH-4
+SWITCH-5
+SWITCH-6
+SWITCH-7
+SWITCH-8
■SYMBOLIC
SYNC(HRONIZED)

■TABLE
TALLYING
■TAPE
■TERMINAL
■TERMINATE
■TEXT
THAN
THROUGH
THRU
TIME
TIMES
TO
TOP
+TRACE
TRAILING
+TRAILING-SIGN
■TYPE

+UNDERLINE
■UNIT
UNSTRING

UNTIL
UP
+UPDATE
UPON
USAGE
USE
+USER
USING

VALUE(S)
VARYING

WHEN
WITH
WORDS

WORKING-STORAGE
WRITE

ZERO((E)S)
+ZERO-FILL

+
-
*
/
**
>
<
=

APPENDIX C. THE LINKER (LINK) PROGRAM

Introduction

The Linker (LINK) program is a program that:

- Combines separately produced object modules.
- Searches library files for definitions of unresolved external references.
- Resolves external cross-references.
- Produces a printable listing that shows the resolution of external references and error messages.
- Produces a relocatable load module.

In this appendix, we show you how to start LINK. You should read all of this appendix before you start LINK.

Files

The linker processes the following input, output, and temporary files:

Input Files

Type	Default .ext	Override .ext	Produced by
Object	.OBJ	Yes	Compiler[2] or MACRO Assembler
Library	.LIB	Yes	Compiler User
Automatic Response	(None)	N/A*	

Figure 17. Input Files Used by the Linker

*N/A–Not applicable.

Output Files

Type	Default .ext	Override .ext	Used by
Listing	.MAP	Yes	User
Run	.EXE	No	Relocatable loader (COMMAND.COM)

Figure 18. Output Files Used by the Linker

[2] One of the optional compiler packages available for use with the IBM Personal Computer DOS.

VM.TMP (Temporary File)

LINK uses as much memory as is available to hold the data that defines the load module being created. If the module is too large to be processed with the available amount of memory, the linker may need additional memory space. If this happens, a temporary diskette file called VM.TMP is created on the DOS default drive.

A message is displayed to indicate when the overflow to diskette has begun. Once this temporary file is created, you should not remove the diskette until LINK ends. When LINK ends, the VM.TMP file is deleted.

If the DOS default drive already has a file by the name of VM.TMP, it will be deleted by LINK and a new file will be allocated. The contents of the previous file are destroyed; therefore, you should avoid using VM.TMP as one of your own filenames.

Definitions

Segment, *group*, and *class* are terms that appear in this chapter and in some of the messages at the end of this appendix. These terms describe the underlying function of LINK. An understanding of the concepts that define these terms provides a basic understanding of the way LINK works.

Segment

A *segment* is a contiguous area of memory up to 64K bytes in length. A segment may be located anywhere in memory on a *paragraph* (16-byte) boundary. Each of the four segment registers defines a segment. The segments can overlap. Each 16-bit address is an offset from the beginning of a segment. The contents of a segment are addressed by a segment register/offset pair.

The contents of various portions of the segment are determined when machine language is generated.

Neither size nor location is necessarily fixed by the machine language generator because this portion of the segment may be combined at linker time with other portions forming a single segment.

A program's ultimate location in memory is determined at load time by the relocation loader facility provided in COMMAND.COM, based on your response to the Load Low parameter. The Load Low parameter is discussed later in this appendix.

Group

A *group* is a collection of segments that fit together within a 64K-byte segment of memory. The segments are named to the group by the assembler or compiler. A program may consist of one or more groups.

The group is used for addressing segments in memory. The various portions of segments within the group are addressed by a segment base pointer plus an offset. The linker checks that the object modules of a group meet the 64K-byte constraint.

Class

A *class* is a collection of segments. The naming of segments to a class affects the order and relative placement of segments in memory. The class name is specified by the assembler or compiler. All portions assigned to the same class name are loaded into memory contiguously.

The segments are ordered within a class in the order that the linker encounters the segments in the object files. One class precedes another in memory only if a segment for the first class precedes all segments for the second class in the input to LINK. Classes are not restricted in size. The classes are divided into groups for addressing.

Command Prompts

After you start the linker session, you receive a series of four prompts. You can respond to these prompts from the keyboard, respond to these prompts on the command line, or use a special diskette file that is called an *automatic response file* to respond to the prompts. An example of an automatic response file is provided in this appendix. Refer to the section called "How to Start the Linker Program" in this appendix for information on how to start the Linker session.

LINK prompts you for the names of the object, run, list, and library files. When the session is finished, LINK returns to DOS. The DOS prompt is displayed when LINK has finished. If the LINK is unsuccessful, LINK displays a message.

The prompts are described in their order of appearance on the screen. The default is shown in square brackets ([ ]), in the response column. Prompts that are not followed by a default require a response from you.

PROMPT	RESPONSES
Object Modules [.OBJ]:	*filespec*[*+filespec2. . .*]
Run File [filename1.EXE]:	*filespec* [/P]
List File [NUL.MAP]:	[*filespec*]
Libraries [.LIB]:	[*filespec*[*+filespec. . .*]]

Figure 19. Command Prompts for the Linker

Notes:

1. If you enter a filespec without specifying the drive, the default drive is assumed. The libraries prompt is an exception.

2. You can end the linker session prior to its normal end by pressing Ctrl-Break.

Detailed Descriptions of the Command Prompts

The following detailed descriptions contain information about the responses that you can enter to the prompts.

Object Modules [.OBJ]:

Enter one or more filespecs for the object modules to be linked. If the extension is omitted, LINK assumes the filename extension .OBJ. If an object module has another filename extension, the extension must also be specified. Object filenames may not begin with the @ symbol. (@ is reserved for using an automatic response file.)

Filespecs must be separated by single plus (+) signs or blanks.

LINK loads segments into classes in the order encountered.

If you specify an object module, but LINK cannot locate the file, a prompt requests you to insert the diskette containing the specific module. This permits .OBJ files from several diskettes to be included. On a single-drive system, diskette exchanging can be done safely *only* if VM.TMP has *not* been opened. A message will indicate if VM.TMP has been opened. The VM.TMP file is discussed earlier in this appendix.

IMPORTANT: If a VM.TMP file has been opened, you should *not* remove the diskette containing the VM.TMP file.

If a VM.TMP file has been opened and the linker is unable to locate an object module on the same drive on which VM.TMP has been allocated, the linker session ends.

Run File [filename1.EXE]:

The filespec you enter is created to store the Run (executable) file that results from the LINK session. All Run files receive the filename extension .EXE, even if you specify another extension. If you specify another extension, your specified extension is ignored.

The default filename for the Run file prompt is the first filename specified on the object module prompt.

List File [NUL.MAP]:

The List file is not created unless you specifically request it. You can request it by overriding the default with a filespec or a drive ID. If the linker is unable to locate an object module on the same drive on which the list file has been allocated, the linker session ends.

The List file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in the Run file.

The DOS reserved filename NUL with the default extension .MAP is used if you do not enter a filespec.

> **Note:** If the List file is allocated to a diskette, it must not be removed until the LINK has ended.

To avoid generating the .MAP file on a diskette, you can specify the display as the List file device. For example:

```
List File [NUL.MAP]: CON
```

If you direct the output to your display, you can also print a copy of the output by pressing the Ctrl-PrtSc keys.

Libraries [.LIB]:

The valid responses are either listing the library filespecs, or pressing the Enter key. If you just press the Enter key, LINK defaults to the library provided as part of the Compiler package. The Compiler package also provides the location of the library. For linking objects from just the MACRO Assembler, there is no automatic default library search.

When LINK attempts to reference a library file and cannot find it, a prompt requests you to enter the drive identifier containing the library.

If you answer the library prompt, you may specify a list of drive IDs and filespecs separated by plus (+) signs or spaces. A drive ID tells the linker where to look for all subsequent libraries on the library prompt. The automatically searched library filespecs are conceptually placed at the end of the response to the library prompt.

When linking an object module produced by the IBM Personal Computer COBOL Compiler which looks for the libraries COBOL1.LIB and COBOL2.LIB on drive A, the following library prompt responses may be used:

```
Libraries [.LIB]:B:
```

Look for COBOL1.LIB and COBOL2.LIB on drive B.

```
Libraries [.LIB]:B:USERLIB
```

Look for USERLIB.LIB on drive B and COBOL1.LIB and COBOL2.LIB on drive A.

```
Libraries [.LIB]:A:+USERLIB1+USERLIB2+B:+USERLIB3+A:
```

Look for USERLIB1.LIB and USERLIB2.LIB on drive A, USERLIB3.LIB on drive B, and COBOL1.LIB and COBOL2.LIB on drive A.

You can enter from 1-8 library filespecs. The filespecs must be separated by plus signs or spaces.

LINK searches the library files in the order in which they are listed to resolve external references. When LINK finds the module that defines the external symbol, the module is processed as another object module.

If two or more libraries have the same filename, regardless of the location, only the first library in the search order is searched.

Parameters

At the end of any of the four linker prompts, you may specify one or more parameters that instruct the linker to do something differently. Only the / and first letter of any parameter are required.

/DSALLOCATION

The /DSALLOCATION parameter directs LINK to load all data defined to be in DGROUP at the *high-end* of the group. If the /HIGH parameter is specified, (module loaded high), this allows any available storage below the specifically allocated area within DGROUP to be allocated dynamically by your application and still be addressable by the same data space pointer.

> **Note:** The maximum amount of storage which can be dynamically allocated by the application is 64K (or the amount actually available) minus the allocated portion of DGROUP.

If the /DSALLOCATION parameter is not specified, LINK loads all data defined to be in the group whose group name is DGROUP, at the *low-end* of the group, beginning at an offset of 0. The only storage thus referenced by the data space pointer should be that specifically defined as residing in the group.

All other segments of any type in any group other than DGROUP are loaded at the low-end of their respective groups, as if the /DSALLOCATION parameter were not specified.

For certain compiler packages, DSALLOCATION is automatically used.

/HIGH

The /HIGH parameter causes the loader to place the Run image as high as possible in storage. If you specify the /HIGH parameter, you tell the linker to cause the loader to place the Run file as high as possible without overlaying the transient portion of COMMAND.COM, which occupies the highest area of storage when loaded. If you do not specify the /HIGH parameter, the linker directs the loader to place the Run file as low in memory as possible.

The /HIGH parameter is used with the /DSALLOCATION parameter.

/LINE

For certain IBM Personal Computer language processors, the /LINE parameter directs LINK to include the line numbers and addresses of the source statements in the input modules in the List file.

/MAP

The /MAP parameter directs LINK to list all public (global) symbols defined in the input modules. For each symbol, LINK lists its value and segment-offset location in the Run file. The symbols are listed at the end of the List file.

/PAUSE

The /PAUSE parameter tells LINK to display a message to you. This message requests you to insert the diskette that is to receive the Run file.

/STACK:size

The size entry is any positive decimal value up to 65536 bytes. If you do not use the /STACK, you specify that the original stack size provided by the assembler or compiler is to be used.

If you specify a value greater than 0 but less than 512, the value 512 is used. This value is used to override the size of the stack that the assembler or compiler has provided for the load module being created.

If the size of the stack is too small, the results of executing the resulting load module are unpredictable.

At least one input (object) module must contain a stack allocation statement. This is automatically provided by compilers. For the assembler, the source must contain a SEGMENT command that has the combine type of STACK. If a stack allocation statement was not provided, LINK returns the following message: **Warning: No Stack statement.**

How to Start the Linker Program

Before You Begin

- Make sure the files you will be using for the LINK are on the appropriate diskettes.
- Make sure you have enough free space on your diskettes to contain your files and any generated data.
- Make sure that the DOS default drive is correct. If the default is drive B, you will need to add **A:** to the following commands.

You can start the Linker program by using one of three options:

Option 1—Console Responses

From your keyboard, enter:

```
LINK
```

The linker is loaded into memory and displays a series of four prompts, one at a time, to which you must enter the requested responses. (Detailed descriptions of the responses that you can make to the prompts are discussed in this appendix in the section called "Command Prompts.")

If you enter an erroneous response, such as the wrong filespec or an incorrectly spelled filespec, you must press Ctrl-Break to exit LINK, then you must restart LINK. If the response in error has been typed but not entered, you may delete the erroneous characters, for that line only.

An example of a Linker session, using the console response option, is provided in this appendix in the section called "Example Linker Session."

As soon as you have entered the last filename, the linker begins to run. If the linker finds any errors, it displays the errors on the screen as well as in the listing file.

> **Note**: After any of these responses, before pressing Enter, you may continue the response with a comma and the answer to what would be the next prompt, without having to wait for that prompt. If you end any with the semicolon (;), the remaining responses are all assumed to be the default. Processing begins immediately with no further prompting.

Option 2–Command Line

From your keyboard, enter:

```
LINK objlist,runfile,mapfile,liblist/parms;
```

Your linker is loaded and immediately performs the tasks indicated by the command field as shown in the above example.

When you use this command line, the prompts described in Option 1 are not displayed if you specified an entry for all four files or if the command line ends with a semicolon.

If an incomplete list is given and no semicolon is used, the linker prompts for the remaining unspecified files. The */parms* are never prompted for, but may be added to the end of the command line or to any file specification given in response to a prompt. Each prompt displays its default, which may be accepted by pressing the Enter key, or overridden with an explicit filename or device name. However, if an incomplete list is given and the command line is terminated with a final semicolon, the unspecified files default without further prompting.

Certain variations of this command line are permitted.

Examples:

```
1)  LINK module
```

Object Module is *module*.OBJ. A prompt is given, showing the default of *module*.EXE. After the response is entered, a prompt is given showing the default of NUL.MAP. After the response is given, a prompt is displayed showing the default of .LIB.

```
2)  LINK module;
```

If the semicolon is added, no further prompts are displayed. The object module of *module*.OBJ is linked, the runfile is put into *module*.EXE, and no listfile is produced.

```
3)  LINK module,,;
```

This is similar to the above example, except the listfile is produced in *module*.MAP.

```
4)  LINK module,,
```

Using the same example, but without the semicolon, *module*.OBJ is linked, and the runfile is produced in *module*.EXE, but a prompt is given with the default of *module*.MAP.

```
5)  LINK module,,NUL;
```

No listfile is produced. The runfile is in *module*.EXE. No further prompts are displayed.

Option 3—Automatic Responses

From your keyboard, enter:

```
LINK @filespec
```

It is often convenient to save responses to the linker for use at a later time. This is especially useful when long lists of object modules need to be specified.

For this option, you enter a filespec preceded by an @ symbol in place of a prompt response or part of a prompt response. The prompt is answered by the contents of the diskette file. The filespec may not be a reserved DOS filename.

Before using this option, you must create the automatic response file. It contains several lines of text, each of which is the response to a linker prompt. These responses must be in the same order as the linker prompts that were discussed earlier in this chapter. If desired, a long response to the object module or libraries prompt may be contained across several lines by using a plus sign (+) to continue the same response onto the next line.

Use of the filename extension is optional and may be any name. There is no default extension.

Use of this option permits the command that starts LINK to be entered from the keyboard or within a batch file without requiring any response from you.

Example

Automatic Response File–Resp1

```
MODA+MODB+MODC
MODD+MODE+MODF
```

Automatic Response File–Resp2

```
Runfile/P
Printout
```

Command line

```
LINK @Resp1+mymod,@Resp2:
```

Notes:

1. In this example, the use of the plus sign causes the modules listed in the first two lines and any module entered by the operator in response to the object module prompts to be considered as the input object modules.

2. Each of the above lines ends when you press the Enter key.

Example Linker Session

This example shows you the type of information that is displayed during a linker session.

Once you enter:

```
B>a:link
```

the system responds with the following messages:

```
IBM Personal Computer Linker
Version 1.10 (C) Copyright IBM Corp 1982
Object Modules[OBJ]: example
Run File[EXAMPLE.EXE]: example/MAP
List File [NUL.MAP]:prn/line
Libraries [.LIB]:
```

Notes:

1. By responding **prn** to the List file prompt, we send our output to the printer.

2. By just pressing Enter in response to the Libraries prompt, an automatic library search is performed.

3. By specifying the /MAP parameter, we get both an alphabetic listing and a chronological listing of public symbols.

4. By specifying the /LINE parameter, LINK gives us a listing of all line numbers for all modules. The /LINE parameter can generate a large amount of output. (The /LINE parameter is not functional for IBM COBOL.)

If LINK cannot locate a library on the specified drive, the following message is displayed:

```
Cannot find library A:COBOL1.LIB
Enter new drive letter:
```

The drive that the indicated library is located on must be entered.

Once LINK locates all libraries, the linker MAP displays a list of segments in the relative order of their appearance within the load module. The list looks like this:

```
Start   Stop    Length  Name            Class

00000H 00028H 0029H   MAINQQ          CODE
00030H 000F6H 00C7H   ENTXQQ          CODE
00100H 00100H 0000H   INIXQQ          CODE
00100H 038D3H 37D4H   FILVQQ_CODE     CODE
038D4H 04921H 104EH   FILUQQ_CODE     CODE
  •
  •
  •
074A0H 074A0H 0000H   HEAP            MEMORY
074A0H 074A0H 0000H   MEMORY          MEMORY
074A0H 0759FH 0100H   STACK           STACK
075A0H 07925H 0386H   DATA            DATA
07930H 082A9H 097AH   CONST           CONST
```

The information on the **Start** and **Stop** columns shows a 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module. The addresses displayed are not the absolute addresses of where these segments are loaded. To find the absolute address of where a segment is actually loaded, you must determine where the segment listed as being at relative zero is actually loaded; then add the absolute address to the relative address shown in the .MAP listing. The procedure you use to determine where relative zero is actually located is discussed in this appendix, in the section called "How to Determine the Absolute Address of a Segment."

Now, because we specified the /MAP parameter, the public symbols are displayed by name and by value. For example:

```
  Address         Publics by Name

  0492:0003H      ABSNQQ
  06CD:029FH      ABSRQQ
  0492:00A3H      ADDNQQ
  06CD:0087H      ADDRQQ
  0602:000FH      ALLHQQ
    .
    .
    .
  0010:1BCEH      WT4VQQ
  0010:1D7EH      WTFVQQ
  0010:1887H      WTIVQQ
  0010:19E2H      WTNVQQ
  0010:11B2H      WTRVQQ

  Address         Publics by Value

  0000:0001H      MAIN
  0000:0010H      ENTGQQ
  0000:0010H      MAINQQ
  0003:0000H      BEGXQQ
  0003:0095H      ENDXQQ
    .
    .
    .
  F82B:F31CH      CRCXQQ
  F82B:F31EH      CRDXQQ
  F82B:F322H      CESXQQ
  F82B:F5B8H      FNSUQQ
  F82B:F5E0H      OUTUQQ
```

The addresses of the public symbols are also in the *segment:offset* format, showing the location relative to zero as the beginning of the load module. In some cases, an entry may look like this:

```
F8CC:EBE2H
```

This entry appears to be the address of a load module that is almost one megabyte in size. Actually, the area being referenced is relative to a segment base that is pointing to a segment below the relative zero beginning of the load module. This condition produces a pointer that has effectively gone negative. The chart on the following page is provided to illustrate this point.

When LINK has completed, the following message is displayed:

```
Program entry point at 0003:0000
```

Load Module Memory Map

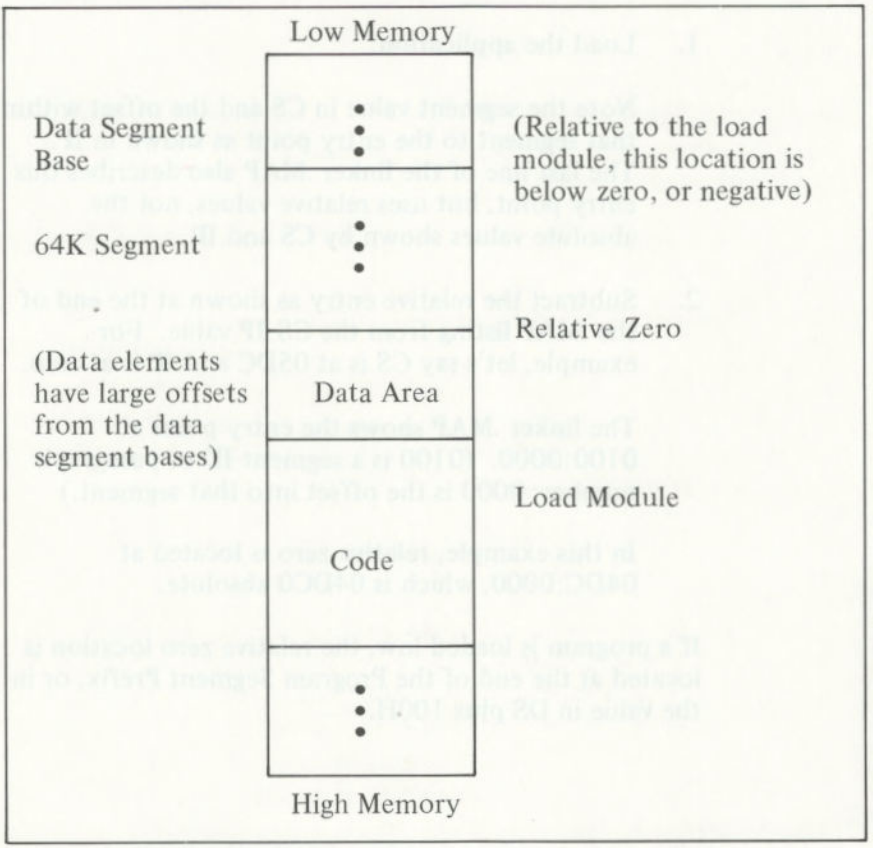


Figure 20. Load Module Memory Map

How to Determine the Absolute Address of a Segment

The linker .MAP displays a list of segments in the relative order of their appearance within the load module. The information displayed shows a 20-bit hex address of each segment relative to location zero. The addresses that are displayed are not the absolute addresses of where these segments are actually located. To determine where relative zero is actually located,

1. Load the application.

   Note the segment value in CS and the offset within that segment to the entry point as shown in IP. The last line of the linker .MAP also describes this entry point, but uses relative values, not the absolute values shown by CS and IP.

2. Subtract the relative entry as shown at the end of the .MAP listing from the CS:IP value. For example, let's say CS is at 05DC and IP is at zero.

   The linker .MAP shows the entry point at 0100:0000. (0100 is a segment ID or paragraph number; 0000 is the offset into that segment.)

   In this example, relative zero is located at 04DC:0000, which is 04DC0 absolute.

If a program is loaded low, the relative zero location is located at the end of the Program Segment Prefix, or in the value in DS plus 100H.

All messages, except for the warning messages, cause the LINK session to end. Therefore, after you locate and correct a problem, you must rerun LINK.

Messages appear both in the listfile and on the display unless you direct the listfile to CON, in which case the display messages are suppressed.

A complete list of linker messages follows:

About to generate .EXE file

Change diskettes and press any key.

An internal failure has occurred

Report this problem to your authorized IBM Personal Computer Dealer.

Attempt to access data outside of segment bounds

The object module is probably bad.

Bad Numeric Parameter

An invalid number was found on the /STACK parameter.

Cannot find file filename

Change diskettes and press any key. This error is unrecoverable if either VM.TMP or the listfile has been opened to a diskette where the object cannot be located.

Cannot find library libraryname

Enter new drive letter.

Cannot open overlay

Cannot open temporary file

The directory is full.

DUP record too complex

A problem exists in an object module created from an assembler source program. A single DUP requires 1024 bytes before expansion.

Fixup offset exceeds field width

A machine language processor instruction refers to an address with a NEAR attribute instead of a FAR attribute.

Invalid format file

A library is in error.

Invalid object module

Object module(s) incorrectly formed or incomplete (as when the language processor is stopped in the middle).

Invalid Switch

The linker found an invalid parameter on the command line or on a prompt.

Out of space on list file

Out of space on run file

Out of space on VM.TMP

No more diskette space remains to expand the VM.TMP file.

Program size exceeds capacity of linker

The load module is too big for processing.

Segment size exceeds 64K

Attempted to combine identically named segments, which resulted in segment requirement of greater than 64K. 64K-bytes is the addressing limit.

Stack Size Exceeds 64K

A number greater than 65536 was found on the /STACK parameter.

Symbol defined more than once

The linker found two or more modules that define a single symbol name.

Symbol table capacity exceeded

The limit is about 30K. Use shorter and/or fewer names.

There was/were number errors detected

Too many libraries specified

The limit is 8 libraries.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is 10, including DGROUP.

Too many public symbols in one module

The limit is 1024 public symbols.

Too many segments or classes

The limit is 256 (segments and classes taken together).

Too many overlays

The limit is 64.

Unexpected end-of-file on library

Unexpected end-of-file on VM.TMP

The diskette containing VM.TMP has been removed.

Unresolved external reference

A call statement reference could not be found.

VM.TMP is an illegal file name and has been ignored

VM.TMP *cannot* be used for object filename.

```
B>REM ---- The following sample session illustrates the use

B>REM ---- of the IBM Personal Computer COBOL Compiler to compile, debug and

B>REM ---- execute an application program.  The intermixed REMs and

B>REM ---- PAUSEs are for documentation purposes only.
B>
B>REM ---- The DOS diskette is in drive A

B>REM ---- Make drive B the default drive

B>B:

B>PAUSE -- Insert into drive B an unformatted diskette (the scratch diskette)
Strike a key when ready . . . x
B>REM ---- Format the new diskette

B>a:format
Insert new diskette for drive B:
and strike any key when ready/
```

```
Formatting...Format complete

Format another (Y/N)?n


B>REM ---- Make a copy of Edlin on the scratch diskette

B>copy a:edlin.com b:
        1 File(s) copied
B>REM ---- Invoke Edlin to input the program
B>edlin payroll.cob
New file
*i
     1:       IDENTIFICATION DIVISION.
     2:       PROGRAM-ID.        YOUR NAME.
     3:       INSTALLATION.      YOUR ADDRESS.
     4:       DATE-WRITTEN.      11/17/81.
     5:       SECURITY.          NONE.
     6:      *REMARKS.           ENTER DATA AS REQUESTED ON SCREEN.
     7:      *
     8:      *
```

```
 9:       ENVIRONMENT DIVISION.
10:       FILE-CONTROL.
11:               SELECT OUT-FILE ASSIGN TO PRINTER.
12:      *
13:      *
14:       DATA DIVISION.
15:       FILE SECTION.
16:       FD        OUT-FILE
17:               LABEL RECORDS ARE OMITTED.
18:       01        PRINT-LINE.
19:               02  PRINT-REC   PIC X(80).
20:      *
21:       WORKING-STORAGE SECTION.
22:       77        OVERTIME       PIC 999V99      VALUE ZEROS.
23:       77        WEEKS-PAY      PIC 9(4)V99     VALUE SPACES.
24:       77        TOTAL-PAY      PIC 9(5)V99     VALUE ZEROS.
25:      *
26:       01        HEADER-1.
27:               02  FILLER      PIC X(20)       VALUE SPACES.
28:               02  FILLER      PIC X(20)       VALUE 'EMPLOYEE NAME
   '.
29:               02  FILLER      PIC X(20)       VALUE '          PAY CH
ECK'.
30:               02  FILLER      PIC X(20)       VALUE SPACES.
```

```
31:      *
32:       01       HEADER-2.
33:                02  FILLER      PIC X(20)       VALUE SPACES.
34:                02  FILLER      PIC X(40)       VALUE ALL '-'.
35:                02  FILLER      PIC X(20)       VALUE SPACES.
36:      *
37:       01       REPORT-LINE.
38:                02  FILLER      PIC X(20)       VALUE SPACES.
39:                02  EMPLOYEE    PIC X(20).
40:                02  FILLER      PIC X(11)       VALUE SPACES.
41:                02  PAY-CHECK   PIC $$$9.99     VALUE ZEROS.
42:                02  FILLER      PIC X(23)       VALUE SPACES.
43:      *
44:       01       TOTAL-LINE.
45:                02  FILLER      PIC X(40)       VALUE SPACES.
46:                02  FILLER      PIC X(9)        VALUE 'TOTAL = '.
47:                02  TOTAL-AMT   PIC $$$$$9.99   VALUE ZEROS.
48:                02  FILLER      PIC X(23)       VALUE SPACES.
49:      *
50:       01       ENTRY-DATA.
51:                02  HOURLY-RATE PIC 99V99.
52:                02  HRS-WORKED  PIC 999.
53:      *
54:      *
55:      *
```

```
56:     PROCEDURE DIVISION.
57:     MAIN-PARAGRAPH.
58:             OPEN OUTPUT OUT-FILE.
59:             WRITE PRINT-LINE FROM HEADER-1 AFTER ADVANCING PAGE.
60:             WRITE PRINT-LINE FROM HEADER-2 AFTER ADVANCING 1.
61:             MOVE 'MARK FRANKLYN       ' TO EMPLOYEE.
62:             PERFORM IN-OUT-PARAGRAPH.
63:             MOVE 'MICHAEL LAWRENCE    ' TO EMPLOYEE.
64:             PERFORM IN-OUT-PARAGRAPH.
65:             MOVE 'JESSICA LYNN        ' TO EMPLOYEE.
66:             PERFORM IN-OUT-PARAGRAPH.
67:             MOVE TOTAL-PAY TO TOTAL-AMT.
68:             WRITE PRINT-LINE FROM TOTAL-LINE AFTER ADVANCING 4.
69:             STOP RUN.
70:     *
71:      IN-OUT-PARAGRAPH.
72:             DISPLAY (1, 1) ERASE  'ENTER HOURS WORKED BY ' EMPLOYEE.
73:             ACCEPT  (1, 45) HRS-WORKED.
74:             DISPLAY (4,1) 'ENTER RATE OF PAY'.
75:             ACCEPT  (4, 22)  HOURLY-RATE.
76:             MOVE ZEROS TO OVERTIME.
```

```
    77:                 IF HRS-WORKED > 40
    78:                     COMPUTE OVERTIME = HRS-WORKED - 40
    79:                     COMPUTE OVERTIME = OVERTIME * HOURLY-RATE * .5.
    80:                 COMPUTE WEEKS-PAY = HRS-WORKED * HOURLY-RATE + OVERTIME.
    81:                 ADD WEEKS-PAY TO TOTAL-PAY.
    82:                 MOVE WEEKS-PAY TO PAY-CHECK.
    83:                 WRITE PRINT-LINE FROM REPORT-LINE AFTER 2.
    84:                 MOVE ZEROS TO PAY-CHECK.
    85:

*e

B>REM ---- Look at the directory

B>dir
EDLIN    COM        2392  08-04-81
PAYROLL  COB        2894  11-18-81

B>REM ---- Run COBOL sending the listing to the printer

B>PAUSE -- Insert the COBOL diskette in drive A
Strike a key when ready . . . z
```

```
B>a:COBOL

IBM Personal Computer COBOL Compiler
Version 1.00 (C)Copyright IBM Corp 1982
(C)Copyright Microsoft, Inc. 1982

Source filename [.COB]: payroll
Object filename [payroll.OBJ]:
Source listing  [nul.LST]:

0023:  VALUE DISALLOWED--OCCURS/REDEFINES/TYPE/SIZE CONFLICT.
0074:  IMPROPER PUNCTUATION.
0001:  /F/ FILE NEVER CLOSED.

  3 Errors or Warnings
```

```
B>REM ---- Got three errors, go back to Edlin to fix them

B>edlin payroll.cob
End of input file
*23
    23:*       77       WEEKS-PAY       PIC 9(4)V99     VALUE SPACES.
    23:*       77       WEEKS-PAY       PIC 9(4)V99     VALUE ZEROS.
*69I
    69:*                CLOSE OUT-FILE.
    70:*^\

*75
    75:*                DISPLAY (4,1) 'ENTER RATE OF PAY'.
    75:*                DISPLAY (4, 1) 'ENTER RATE OF PAY'.
*E

B>REM ---- Rerun COBOL
```

```
B>a:COBOL

IBM Personal Computer COBOL Compiler
Version 1.00 (C)Copyright IBM Corp. 1982
(C)Copyright Microsoft, Inc. 1982

Source filename [.COB]: payroll
Object filename [payroll.OBJ]:
Source listing  [nul.LST]: prn

  No Errors or Warnings

B>REM ---- Ok, that worked.

B>REM ---- Look at the directory

B>dir
EDLIN    COM       2392  08-04-81
PAYROLL  BAK       2894  11-18-81
PAYROLL  OBJ       2970  11-18-81
PAYROLL  COB       2913  11-18-81

B>REM ---- Now its Link time

B>PAUSE -- Insert the LIBRARY diskette in drive A
Strike a key when ready . . .
```

```
B>a:link

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982

Object Modules [.OBJ] : payroll
Run File [PAYROLL.EXE] :
List File [NUL.MAP] :
Libraries [.LIB] :

B>REM ---- Look at the directory

B>dir
EDLIN    COM      2392  08-04-81
PAYROLL  BAK      2894  11-18-81
PAYROLL  OBJ      2970  11-18-81
PAYROLL  COB      2913  11-18-81
PAYROLL  EXE     30848  11-18-81

B>REM ---- It compiled, now run the program

B>payroll
```

```
--------------------------------------------------------------------------------

ENTER HOURS WORKED BY MARK FRANKLYN          52

ENTER RATE OF PAY     5.75

--------------------------------------------------------------------------------


ENTER HOURS WORKED BY MICHAEL LAWRENCE       43

ENTER RATE OF PAY     4.33

--------------------------------------------------------------------------------


ENTER HOURS WORKED BY JESSICA LYNN           18

ENTER RATE OF PAY     3.65
```

EMPLOYEE NAME	PAY CHECK
MARK FRANKLYN	$333.50
MICHAEL LAWRENCE	$192.68
JESSICA LYNN	$65.70
TOTAL =	$591.88

APPENDIX E. ADVANCED FORMS OF CONDITIONS

Evaluation Rules for Compound Conditions

1. Individual simple conditions (relation, class, condition name, and sign test) are evaluated first.

2. AND-connected simple conditions are evaluated next as a single result.

3. OR and its adjacent conditions (or previously evaluated results) are evaluated last.

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
```

   After expanding level 88 condition name WEEKLY, the evaluation is equivalent to (PAY-CODE = 'W') AND (HOURS <> 0) and is true only if both the simple conditions are true.

3.
```
A=1 AND B=2 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:

- A = 1
- B = 2
- 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 processed if A = 5 OR A = 1, while at the same time A = B. In this manner, compound conditions may be formed that contain, via the use of parentheses, other compound conditions (not just simple conditions).

Abbreviated Conditions

For the sake of brevity, you 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 that if the item immediately following NOT is a relational operator, then the NOT participates as part of the relational operator. Otherwise, the beginning of a new, completely separate condition must follow NOT, and 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 (for example, 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 also precede a level 88 condition name.

APPENDIX F. NESTING OF IF STATEMENTS

A "nested IF" exists when the verb IF appears more than once in a single sentence.

Example:

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

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

```
                IF1     X = Y

True Action1: IF2     A = B
          True Action2: MOVE "*" TO SWITCH

                ELSE2
         False Action2: MOVE "A" TO SWITCH

                ELSE1
False Action1:MOVE SPACE TO SWITCH.
```

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

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

Examples:

```
IF M = 1
   IF K = 0
      GO TO M1-KO
   ELSE
      GO TO M1-KNOTO

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

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

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

APPENDIX G. ASCII CHARACTER CODES

The following table lists all the ASCII codes (in decimal) and their associated characters. The column headed "Control Character" lists the standard interpretations of ASCII codes 0 to 31 (usually used for control functions or communications).

ASCII value	Character	Control character	ASCII value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	• (beep)	BEL	039	'
008	◘ (backspace)	BS	040	(
009	○ (tab)	HT	041	)
010	◙ (line feed)	LF	042	*
011	♂ (home)	VT	043	+
012	♀ (form feed)	FF	044	,
013	♪ (carriage return)	CR	045	-
014	♫	SO	046	.
015	☼	SI	047	/
016	►	DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	!!	DC3	051	3
020	¶	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	∟ (cursor right)	FS	060	<
029	↔ (cursor left)	GS	061	=
030	▲ (cursor up)	RS	062	>
031	▼ (cursor down)	US	063	?

ASCII value	Character	ASCII value	Character
064	@	095	_
065	A	096	'
066	B	097	a
067	C	098	b
068	D	099	c
069	E	100	d
070	F	101	e
071	G	102	f
072	H	103	g
073	I	104	h
074	J	105	i
075	K	106	j
076	L	107	k
077	M	108	l
078	N	109	m
079	O	110	n
080	P	111	o
081	Q	112	p
082	R	113	q
083	S	114	r
084	T	115	s
085	U	116	t
086	V	117	u
087	W	118	v
088	X	119	w
089	Y	120	x
090	Z	121	y
091	[	122	z
092	\	123	{
093	]	124	¦
094	^	125	}

ASCII value	Character
126	~
127	⌂
128	Ç
129	ü
130	é
131	â
132	ä
133	à
134	å
135	ç
136	ê
137	ë
138	è
139	ï
140	î
141	ì
142	Ä
143	Å
144	É
145	æ
146	Æ
147	ô
148	ö
149	ò
150	û
151	ù
152	ÿ
153	Ö
154	Ü
155	¢
156	£
157	¥
158	Pt

ASCII value	Character
159	ƒ
160	á
161	í
162	ó
163	ú
164	ñ
165	Ñ
166	ª
167	º
168	¿
169	⌐
170	¬
171	½
172	¼
173	¡
174	«
175	»
176	▒
177	▒
178	▓
179	|
180	┤
181	╡
182	╢
183	╖
184	╕
185	╣
186	║
187	╗
188	╝
189	╜
190	╛
191	┐

ASCII value	Character
192	└
193	┴
194	┬
195	├
196	─
197	┼
198	╞
199	╟
200	╚
201	╔
202	╩
203	╦
204	╠
205	═
206	╬
207	╧
208	╨
209	╤
210	╥
211	╙
212	╘
213	╒
214	╓
215	╫
216	╪
217	┘
218	┌
219	■
220	▄
221	▌
222	▐
223	▀
224	α

ASCII value	Character
225	β
226	Γ
227	π
228	Σ
229	σ
230	μ
231	τ
232	Φ
233	Θ
234	Ω
235	δ
236	∞
237	∅
238	∈
239	∩
240	≡
241	±
242	≥
243	≤
244	⌠
245	⌡
246	÷
247	≈
248	°
249	∙
250	·
251	√
252	^{n}
253	^{2}
254	■
255	(blank 'FF')

APPENDIX H. TABLE OF PERMISSIBLE MOVE OPERANDS

The table on the following page shows the permissible operands for the MOVE statement.

Source Operand	Numeric Integer	Numeric Noninteger	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)

Figure 21. Receiving Operands in MOVE Statement

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.

(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; you should not move numeric items to alphabetic items and vice versa.

APPENDIX I. 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. The example varies 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 be a SET (TO form) instead, and the associated ADD would be a SET (UP form).

APPENDIX J. EXAMPLE PROGRAMS WITH VIDEO MODE

The following video modes exist on your IBM Personal Computer:

Mode Number	Meaning		
0	BW	40X25	Alphanumeric
1	COLOR	40X25	Alphanumeric
2	BW	80X25	Alphanumeric
3	COLOR	80X25	Alphanumeric
4	BW	320X200	Graphic
5	COLOR	320X200	Graphic
6	BW	640X200	Graphic
7	BW	80X25	Alphanumeric

Figure 22. Video Modes

The following example programs show how to check the current video mode on your computer.

Example COBOL Program

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. VMODE.
*    DISPLAY VIDEO MODE.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  VIDEO-MODE PIC 9.
 PROCEDURE DIVISION.
 MAIN.
     CALL "GVMODE" USING VIDEO-MODE.
     DISPLAY "VIDEO-MODE IS " VIDEO-MODE.
     STOP RUN.
```

```
;*************************************************************************
;
;       GVMODE places the current video mode, a single ASCII character,
;       at the address pointed to by the stack pointer.
;
;*************************************************************************
PARAM1  EQU     6               ; offset of param on stack
VIDINT  EQU     10H             ; interrupt # for video
GVIDMO  EQU     15              ; video interrupt function # to get mode

        PUBLIC  GVMODE
DGROUP  GROUP   DATA,SSEG
        ASSUME  CS: EXAMPL, DS: DGROUP, SS: SSEG

DATA    SEGMENT
DATA    ENDS

SSEG    SEGMENT STACK
SSEG    ENDS
```

```
EXAMPL  SEGMENT PARA 'CODE'
GVMODE  PROC    FAR
        PUSH    BP              ; preserve BP
        MOV     BP,SP           ; set up local stack frame
        MOV     AH,GIVIDMO      ; set function number
        INT     VIDINT          ; do video interrupt
                                ; AL = mode
                                ; AH = character columns on screen
                                ; BH = current display page
        ADD     AL,"0"          ; convert mode to ASCII decimal
        MOV     BX,PARAM1[BP]   ; get return cell address from stack
        MOV     [BX],AL         ; store mode in return call
        POP     BP              ; restore BP
        RET     2               ; remove 2 bytes (1 param) from stack
GVMODE  ENDP
EXAMPL  ENDS
        END
```

APPENDIX K. INDEXED FILE RECOVERY UTILITY (REBUILD)

Introduction

You can use the Indexed File Recovery Utility (REBUILD) to recover or restore information contained in indexed files created by a program compiled under IBM Personal Computer COBOL (Version 1.00).

How the Utility Works

REBUILD reads the data file portion of an indexed file and generates new key and data files for that indexed file. The new file has the same structure as the old one. You must use an existing indexed data file of non-zero length in order to run this utility.

When to Use REBUILD

Diskette Full

Sometimes the WRITE operation requires more space than is available on the diskette containing the indexed file. The WRITE operation produces a boundary error (file status "24" in indexed files), indicating that the diskette is full. Using REBUILD restores the damaged file structure that occurs (in an indexed file) during the diskette-full condition.

As soon as this happens, close the file to write as much information as possible to the diskette. The CLOSE operation may return with a boundary error. If this happens, as in the case of a system failure during record addition, the last 256 bytes of information do not appear on the data file and are, therefore, not recoverable by the Indexed File Recovery Utility.

Abnormal Termination

Use REBUILD to recover indexed files that are damaged when a power failure interrupts computer processing or if the operating system is restarted with a system reset while an indexed file is open in I/O or output mode.

Because the system uses diskette file buffering in memory, a system failure can leave the data file with partially written data records. Sometimes REBUILD fails to *completely* recover an indexed file because:

- If a system failure occurs during a file update process, the file can contain records with both original and new information because some of the new information may not have been written to the file. REBUILD cannot determine which part of the record the system wrote during the aborted task, and cannot exclude the new, incomplete data from the rebuilt file. If you add a current date field to data records, you can discriminate between original and new data.

- If a system failure occurs when the system adds records to the indexed file, the system cannot write the last 256 bytes of data to the diskette. REBUILD detects that information is missing from the end of the file but cannot add it to the new file being built.

Unusable Space

You can periodically use REBUILD to recover unusable space in the data file portion of an indexed file when you need more space. The unusable space occurs as a result of numerous record DELETE and REWRITE operations, especially when records in the file have varying lengths.

Using REBUILD

REBUILD asks a series of questions about the file to be recovered. Your answers provide the information necessary to rebuild a new indexed file from the original data file. In response to the operating system (DOS) prompt, insert your COBOL diskette in drive B and enter:

```
B:REBUILD
```

The system responds as follows:

```
IBM Personal Computer Indexed File
  Recovery Utility
Version 1.00 (C) Copyright IBM Corp 1982
(C) Copyright Microsoft Corporation 1982
```

Insert your diskette containing the indexed data file to be rebuilt in drive A. If you need extra space, remove the COBOL diskette from drive B, insert a scratch diskette, and send the target file to drive B (by using a drive specifier on your target filename). Answer each prompt and press the Enter key. You can return to the first prompt (Input Key Length) at any time by pressing the Enter key without typing the requested information. To terminate REBUILD, answer "Input Key Length" by pressing the Enter key without typing a response.

```
Input Key Length
```

Enter the length of the key (in bytes) or press the Enter key to terminate the session. Enter the key length as a positive integer that is the number of bytes contained in the item specified by the RECORD KEY clause of the IBM Personal Computer COBOL program. If you enter an incorrect key length, REBUILD continues, but programs cannot access the newly generated indexed file (see "Sample REBUILD Session" on page K-8). After you enter the key length, REBUILD continues with the next prompt:

```
Input Key Position
```

Enter the byte position of the key field, starting at 1, as a positive integer representing the position within the record of the data item specified by the RECORD KEY clause of the IBM Personal Computer COBOL program. REBUILD does not check the response. Therefore, if you enter an incorrect key position, programs cannot access the newly generated indexed file (see "Sample REBUILD Session" on page K-8). After you enter the key position, REBUILD continues with the next prompt:

```
Input Source Filename
```

Enter the filename of the source file. The filename should be the name used in the VALUE OF FILE-ID clause in the IBM Personal Computer COBOL programs that refer to the indexed file. Use the name of the data file and not the name of the key file (which has the same name followed by .KEY). The source filename can contain a drive specifier (see "Sample REBUILD Session" on page K-8).

After you enter the filename, REBUILD checks for the presence of the file. If it is not present, REBUILD displays the following message:

```
***Source file not found
Input Source Filename
```

When you enter a correct name, REBUILD continues with the next prompt:

```
Input Target Filename
```

The target filename should be unique within a directory. If you want to enter a target filename identical to the source filename, send the target file to a different diskette by including a drive specifier in the filename. REBUILD can generate the target file on the same diskette as the source file, but you must use a different name. When the recovery operation is complete, you can rename the target filename to the source filename (see "Sample REBUILD Session" on page K-8).

If REBUILD cannot successfully create a new indexed file because the diskette directory is full or because of insufficient space on the diskette, the program displays the following message:

```
***No space for target file
Input Target Filename
```

If this happens, use another diskette with more space.

After you correctly enter the target filename, REBUILD displays:

```
Now reading source-file
and creating target-file
```

> **Note**: The names you supplied for the source and target files appear for *source-file* and *target-file*.

REBUILD begins building the new indexed file from the old data file. When finished, REBUILD displays the following message:

```
Conversion successfully completed.
Source records read:    xxx,xxx
Target records written: xxx,xxx
```

The record counts should match. If they do not, an input/output error occurred during the recovery operation.

Regardless of whether the record counts match, REBUILD displays the first prompt:

```
Input Key Length
```

You can begin another file recovery operation, *redo the one with an input/output error*, or terminate the program.

> **Note**: Remember that you can terminate the program at any time by pressing the Enter key without responding to a prompt. This brings you back to the first prompt. You can then change the information you gave to the previous session, or you can terminate the program by pressing the Enter key again.

Sample REBUILD Session

The following sample program fragment accesses the IXFILE.DAT indexed file.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL
        SELECT IX-FILE
               ASSIGN TO DISK
               ORGANIZATION INDEXED
               ACCESS DYNAMIC
               RECORD KEY IX-KEY
               FILE STATUS IX-STAT.

DATA DIVISION.
FILE SECTION.
FD IX-FILE
   LABEL RECORD STANDARD
   VALUE OF FILE-ID "IXFILE.DAT"
   RECORD CONTAINS 75 CHARACTERS
   DATA RECORD IX-REC.
01 IX-REC.
   05 IX-DATE      PIC X(6).
   05 IX-TIME      PIC X(6).
   05 IX-KEY.
      10 IX-STATE     PIC XX.
      10 IX-CITY      PIC X(20).
      10 IX-STREET    PIC X(30).
   05 IX-ZIP       PIC X(5).
   05 IX-ZONE      PIC X(6).
```

The responses for this program fragment are:

Prompt	Response
Input Key Length	52
Input Key Position	13
Input Source Filename	A:IXFILE.DAT
Input Target Filename	B:NEWIX.DAT

These responses generate a new indexed file with the key filename NEWIX.KEY and the data filename NEWIX.DAT.

INDEX

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

0

7

8

9

The Personal Computer
Computer Language Series

Product Comment Form

COBOL Compiler
by Microsoft 6172258

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

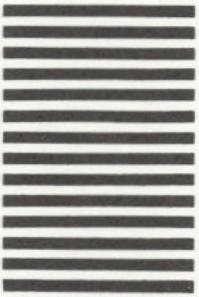
Name _____

Address _____

City _____ State _____

Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

The Personal Computer
Computer Language Series

Product Comment Form

COBOL Compiler
by Microsoft 6172258

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____

Tape Please do not staple Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Continued from inside front cover

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328–W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6172258

Printed in United States of America