MICROSOFT™

COBOL-86

(Z-DOS™)

593-77
CONSISTS OF

MANUAL
595-3047
FLYSHEET
597-3284

Printed in the
United States of America

data
systems

HEATH

MICROSOFT™

COBOL-86

(Z-DOS™)

595-3047

Printed in the
United States of America

ZENITH data systems

HEATH

NOTICE
This software is licensed (not sold). It is licensed to sublicensees, including end-users, without either express or implied warranties of any kind on an "as is" basis.

The owner and distributors make no express or implied warranties to sublicensees, including end-users, with regard to this software, including merchantability, fitness for any purpose or non-infringement of patents, copyrights or other proprietary rights of others. Neither of them shall have any liability or responsibility to sublicensees, including end-users, for damages of any kind, including special, indirect or consequential damages, arising out of or resulting from any program, services or materials made available hereunder or the use or modification thereof.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Technical consultation is available for any problems you encounter in verifying the proper operation of this product. Sorry, but we are not able to evaluate or assist in the debugging of any programs you may develop. For technical assistance, write:

Zenith Data Systems Corporation
Software Consultation
Hilltop Road
St. Joseph, Michigan 49085

or call:

(616) 982-3884 Application Software/SoftStuff Products
(616) 982-3860 Operating Systems/Languages/Utilities

Consultation is available from 8:00 AM to 7:30 PM (Eastern Time Zone) on regular business days.

Microsoft is a registered trademark of Microsoft Corporation.
The Microsoft logo is a trademark of Microsoft Corporation.
Z-DOS is a trademark of Zenith Data Systems Corporation.

Copyright © 1983 by Microsoft Corporation.
Copyright © 1983 Zenith Data Systems Corporation.

ESSENTIAL REQUIREMENTS for using COBOL-86:

a. Distribution Media: Two 5.25-inch soft-sectored 48-tpi disks
b. Machine Configuration (minimum): Z-100, 128K memory, two disk drives, and CRT
c. Operating System: Z-DOS
d. Microcomputer Language: Not Applicable

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

ZENITH DATA SYSTEMS CORPORATION
ST. JOSEPH, MICHIGAN 49085

ACKNOWLEDGMENT

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

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

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

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

The authors and copyright holders of the copyrighted material used herein

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

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

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

CONTENTS

Introduction
How this Manual is Organized 1
Part I: User's Guide 1
Part II: Language Overview 1
Part III: COBOL-86 Reserved Words 2
Part IV: Appendices and Index 2
Syntax Notation 3
Converting COBOL Programs to COBOL-86 4
Learning More about COBOL 5

Part I
User's Guide

Chapter 1 **Getting Started**
Overview 1.1
The Compilation Process 1.1
Your COBOL-86 System 1.3
Disk Backup 1.4
Sample Program Development 1.4
Sample Session 1.5

Chapter 2 **Compiling COBOL Programs**
Overview 2.1
Operating the Compiler 2.1
Compiler Responses 2.2
Partial Command Strings 2.3
Using Compiler Switches 2.5
Switches 2.5
The Source Listing File 2.7
Compiling Large Programs 2.7

Chapter 3 **Linking and Executing COBOL Programs**
Overview 3.1
Using Link 3.1
Linking Programs That Use Overlays 3.6
Linking Program Modules 3.6
Linking Large Programs 3.7
Executing COBOL Programs 3.8

Chapter 4 **Batch Command Files**

Chapter 5 **Data Input and Output**
Overview 5.1
Using Disk Files 5.1
Types of Disk Files 5.2
Using Z-DOS and Non-disk Files 5.3

CONTENTS

Part II Language Overview

Chapter 6 **The Interactive Debug Facility**
Overview 6.1
The Debugging Procedure 6.1
Use of Debug Commands 6.2

Chapter 7 **Introduction**
Overview 7.1
The ANSI Standard 7.1
Coding Fundamentals 7.4
  Character Set 7.4
  Punctuation 7.5
  Word Formation 7.6
  Source Line Structure 7.6
  Program Structure 7.7

Chapter 8 **DATA DIVISION**
Overview 8.1
Data Items 8.1
  Data Item Structures 8.2
Level Numbers 8.4
Data Description Entry 8.6
  Group Item Syntax 8.7
  Elementary Item Syntax 8.8
Literals and Figurative Constants 8.9
  Non-Numeric Literals 8.9
  Numeric Literals 8.10
  Figurative Constants 8.11
Size Limitations 8.12

Chapter 9 **PROCEDURE DIVISION**
Overview 9.1
Statements 9.1
  Statement Types 9.1
  Statement Structures 9.2
Design of the PROCEDURE DIVISION 9.2
  Organization 9.2
  Division Header 9.3
  Declaratives and I/O Error Handling 9.4
  Segmentation 9.5

CONTENTS

Chapter 10 **Indexed Files**

Overview . . . 10.1
Syntax in ENVIRONMENT DIVISION . . . 10.1
  SELECT Clause . . . 10.1
  RECORD KEY Clause . . . 10.1
  FILE STATUS Reporting . . . 10.2
Syntax in PROCEDURE DIVISION . . . 10.3
  OPEN Statement . . . 10.3
  READ Statement . . . 10.4
  WRITE Statement . . . 10.5
  REWRITE Statement . . . 10.6
  DELETE Statement . . . 10.6
  START Statement . . . 10.7

Chapter 11 **Relative Files**

Overview . . . 11.1
Syntax in ENVIRONMENT DIVISION . . . 11.1
  SELECT Clause . . . 11.1
  RELATIVE KEY Clause . . . 11.2
Syntax in PROCEDURE DIVISION . . . 11.2
  READ Statement . . . 11.2
  WRITE Statement . . . 11.3
  REWRITE Statement . . . 11.4
  DELETE Statement . . . 11.4
  START Statement . . . 11.5

Part III COBOL-86 Reserved Words

Chapter 12 **Alphabetical Reserved Word List**

Introduction . . . 12.1
ACCEPT . . . 12.8
  Syntax in PROCEDURE DIVISION . . . 12.8
  Details . . . 12.9
  Application . . . 12.22
ADD . . . 12.25
  Syntax in PROCEDURE DIVISION . . . 12.25
  Details . . . 12.25
  Application . . . 12.26
ALTER . . . 12.27
  Syntax in PROCEDURE DIVISION . . . 12.27
  Details . . . 12.27
  Application . . . 12.27
BLANK . . . 12.28
  Syntax in DATA DIVISION . . . 12.28
  Details . . . 12.28
  Application . . . 12.28

CONTENTS

BLOCK 12.29
Syntax in DATA DIVISION 12.29
Details 12.29
Application 12.29
CALL 12.30
Syntax in PROCEDURE DIVISION 12.30
Details 12.30
Application 12.30
CHAIN 12.32
Syntax in PROCEDURE DIVISION 12.32
Details 12.32
Application 12.32
CLOSE 12.33
Syntax in PROCEDURE DIVISION 12.33
Details 12.33
Application 12.33
CODE-SET 12.34
Syntax in DATA DIVISION 12.34
Details 12.34
Application 12.34
COMPUTE 12.35
Syntax in PROCEDURE DIVISION 12.35
Details 12.35
Application 12.37
CONFIGURATION 12.38
Syntax in ENVIRONMENT DIVISION 12.38
Details 12.38
COPY 12.40
Syntax in PROCEDURE DIVISION 12.40
Details 12.40
Application 12.40
DATA (in DATA DIVISION Header) 12.41
Syntax in Division Header 12.41
Details 12.41
DATA (in DATA RECORD Clause) 12.42
Syntax in DATA DIVISION 12.42
Details 12.42
Application 12.42
DECLARATIVES 12.43
Syntax in PROCEDURE DIVISION 12.43
Details 12.43
DISPLAY 12.45
Syntax in PROCEDURE DIVISION 12.45
Details 12.45
Application 12.47
DIVIDE 12.48
Syntax in PROCEDURE DIVISION 12.48
Details 12.48
Application 12.49

CONTENTS

ENVIRONMENT 12.50
Syntax in Division Header 12.50
Details 12.50
EXHIBIT 12.51
Syntax in PROCEDURE DIVISION 12.51
Details 12.51
Application 12.51
EXIT 12.52
Syntax in PROCEDURE DIVISION 12.52
Details 12.52
Application 12.52
EXIT PROGRAM 12.53
Syntax in PROCEDURE DIVISION 12.53
Details 12.53
Application 12.53
FILE 12.54
Syntax in DATA DIVISION 12.54
Details 12.54
Application 12.55
GO TO 12.56
Syntax in PROCEDURE DIVISION 12.56
Details 12.56
Application 12.56
IDENTIFICATION 12.57
Syntax in Division Header 12.57
Details 12.57
IF 12.58
Syntax in PROCEDURE DIVISION 12.58
Details 12.58
Application 12.61
IN, OF 12.63
Syntax in PROCEDURE DIVISION 12.63
Details 12.63
Application 12.63
INPUT-OUTPUT 12.64
Syntax in ENVIRONMENT DIVISION 12.64
Details 12.64
INSPECT 12.67
Syntax in PROCEDURE DIVISION 12.67
Details 12.67
Application 12.69
JUSTIFIED 12.70
Syntax in DATA DIVISION 12.70
Details 12.70
Application 12.70
LINAGE 12.71
Syntax in DATA DIVISION 12.71
Details 12.71
Application 12.72

CONTENTS

LINKAGE 12.73
  Syntax in DATA DIVISION 12.73
  Details 12.73
  Application 12.73
MOVE 12.74
  Syntax in PROCEDURE DIVISION 12.74
  Details 12.74
  Application 12.75
MULTIPLY 12.77
  Syntax in PROCEDURE DIVISION 12.77
  Details 12.77
  Application 12.78
OCCURS 12.79
  Syntax in DATA DIVISION 12.79
  Details 12.79
  Application 12.81
OPEN 12.83
  Syntax in PROCEDURE DIVISION 12.83
  Details 12.83
  Application 12.84
PERFORM 12.85
  Syntax in PROCEDURE DIVISION 12.85
  Details 12.86
  Application 12.87
PICTURE 12.89
  Syntax in DATA DIVISION 12.89
  Details 12.89
  Application 12.95
PROCEDURE 12.96
  Syntax in Division Header 12.96
  Details 12.96
READ (to Perform Sequential Input) 12.97
  Syntax in PROCEDURE DIVISION 12.97
  Details 12.97
  Application 12.97
RECORD 12.98
  Syntax in DATA DIVISION 12.98
  Details 12.98
  Application 12.98
REDEFINES 12.99
  Syntax in DATA DIVISION 12.99
  Details 12.99
  Application 12.100
REWRITE (to Perform Sequential I/O) 12.101
  Syntax in PROCEDURE DIVISION 12.101
  Details 12.101
  Application 12.101

CONTENTS

SCREEN ... 12.102
  Syntax in DATA DIVISION ... 12.102
  Details ... 12.103
SEARCH ... 12.107
  Syntax in PROCEDURE DIVISION ... 12.107
  Details ... 12.107
  Application ... 12.110
SET ... 12.111
  Syntax in PROCEDURE DIVISION ... 12.111
  Details ... 12.111
  Application ... 12.111
SIGN ... 12.112
  Syntax in DATA DIVISION ... 12.112
  Details ... 12.112
  Application ... 12.112
STOP ... 12.113
  Syntax in PROCEDURE DIVISION ... 12.113
  Details ... 12.113
  Application ... 12.113
STRING ... 12.114
  Syntax in PROCEDURE DIVISION ... 12.114
  Details ... 12.114
  Application ... 12.115
SUBTRACT ... 12.116
  Syntax in PROCEDURE DIVISION ... 12.116
  Details ... 12.116
  Application ... 12.117
SYNCHRONIZED ... 12.118
  Syntax in DATA DIVISION ... 12.118
  Details ... 12.118
  Application ... 12.118
TRACE ... 12.119
  Syntax in PROCEDURE DIVISION ... 12.119
  Details ... 12.119
  Application ... 12.119
UNSTRING ... 12.120
  Syntax in PROCEDURE DIVISION ... 12.120
  Details ... 12.120
  Application ... 12.122
USAGE ... 12.123
  Syntax in DATA DIVISION ... 12.123
  Details ... 12.123
  Application ... 12.124
VALUE (to Define Truth Set of Condition-name) ... 12.125
  Syntax in DATA DIVISION ... 12.125
  Details ... 12.125
  Application ... 12.126

CONTENTS

VALUE (to Initialize Data Value) . . . 12.127
Syntax in DATA DIVISION . . . 12.127
Details . . . 12.127
Application . . . 12.128
VALUE (to Specify a Disk Filename) . . . 12.129
Syntax in DATA DIVISION . . . 12.129
Details . . . 12.129
Application . . . 12.129
WORKING-STORAGE . . . 12.130
Syntax in DATA DIVISION . . . 12.130
Details . . . 12.130
WRITE (to Perform Sequential Output) . . . 12.131
Syntax in PROCEDURE DIVISION . . . 12.131
Details . . . 12.131
Application . . . 12.132

Part IV Appendices and Index

Appendix A **Interprogram Communication**
Overview . . . A.1
Calling COBOL Programs . . . A.2
Syntax . . . A.2
Purpose . . . A.2
Details . . . A.2
Sample Program Structure . . . A.3
Calling Assembly Language Subroutines . . . A.4
Sample Program Structure . . . A.5
Chaining COBOL Programs . . . A.6
Syntax . . . A.6
Purpose . . . A.7
Details . . . A.7
Sample Program Structure . . . A.7
Chaining Assembly Language Programs . . . A.8

Appendix B **Customizations**
Source Program Tab Stops . . . B.1
Compiler Listing Page Length . . . B.1

Appendix C **Compiler Phases**

Appendix D **Rebuild: Indexed File Recovery Utility**
Overview . . . D.1
Running Rebuild . . . D.2
Sample Rebuild Session . . . D.7

CONTENTS

Appendix E **COBOL-86 Error Messages**
Overview E.1
Command Input and Operating System I/O Errors E.2
Program Syntax Errors E.4
Runtime Errors E.15
Program Load Errors E.17

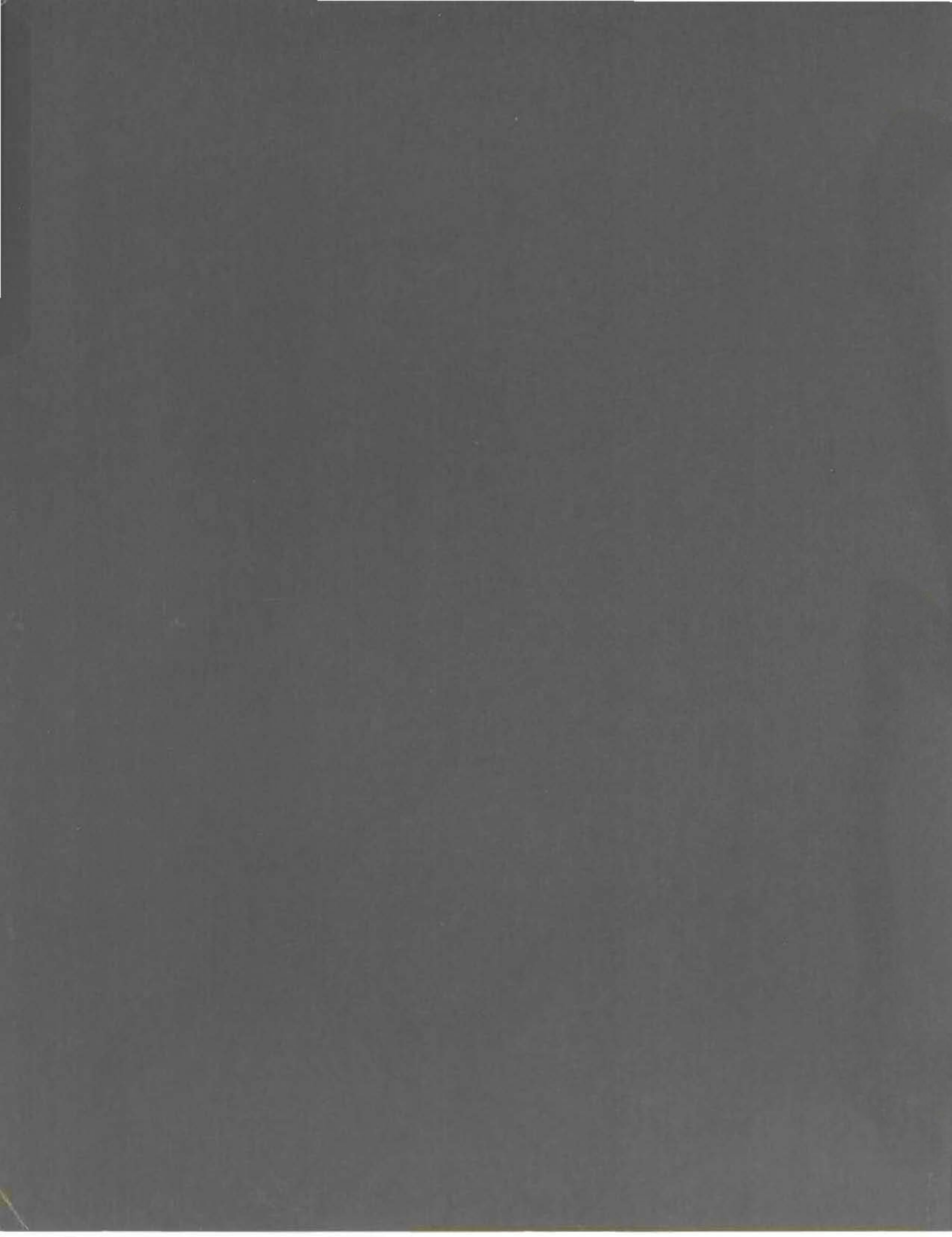
Appendix F **Demonstration Programs**
COBOL-86 Programs F.1
CRTEST F.1
CENTER F.1
COBOL-86 Demonstration System F.1

Appendix G **ASCII Character Set for ANSI-74 COBOL**

Appendix H **Additional ANSI Reserved Words**

Index

USER'S GUIDE



INTRODUCTION

The COBOL-86 Compiler is an extensive implementation of the COBOL language for microcomputers. This compiler has been certified with the Federal Compiler Testing Center at the Low Intermediate level of compliance with the ANSI X3.23-1974 standard. COBOL-86 has many features that are standard for higher levels of validation and includes extensions to the standard that are designed to optimize COBOL's usefulness in the microcomputer environment.

How this Manual is Organized

Part I: User's Guide

Chapters 1 through 3 provide the information you need to compile, link, load, and execute a COBOL-86 program.

Chapter 4 tells you how to set up a batch command file to "compile, link, and go."

Chapter 5 explains the four disk file organizations: sequential, line sequential, relative, and indexed. It also describes how to use disk input/output files and other types of files.

Chapter 6 tells you how to use the Interactive Debug Facility to identify program errors at runtime.

Part II: Language Overview

Chapter 7 provides fundamental information that is pertinent to any use of the COBOL language.

Chapter 8 describes the purpose, structure, and limitations of the DATA DIVISION.

Chapter 9 covers the PROCEDURE DIVISION in a manner similar to that used in Chapter 8.

Chapters 10 and 11 treat indexed and relative files, respectively. Both general information and the syntax of specific reserved words are included.

INTRODUCTION

How this Manual is Organized

Part III: COBOL-86 Reserved Words

Chapter 12 is an alphabetical listing that includes a syntax diagram, an elaboration of details, and, where appropriate, examples of common applications for each reserved word used in COBOL-86. This chapter provides the principal programming aid for the experienced COBOL user.

Part IV: Appendices and Index

Appendix A explains interprogram communication with the CALL and CHAIN statements.

Appendix B shows you how to customize some of the COBOL-86 features.

Appendix C gives an overview of the five phases of the COBOL-86 Compiler. This appendix may be useful if your program results in a Compiler phase error.

Appendix D describes the REBUILD program, which allows you to recover or restore information in damaged indexed files.

Error messages are listed in Appendix E. They are arranged alphabetically within four sections: (1) command input and operating system I/O errors, (2) program syntax errors, (3) runtime errors, and (4) program load errors.

Appendix F gives you directions for compiling, linking, and running the demonstration programs.

Appendix G contains the ANSI 1974 character set and the ASCII equivalents in hexadecimal notation.

Appendix H is an alphabetical listing of words reserved in the ANSI 1974 syntax that are not used in COBOL-86. It is included to help you avoid duplicating reserved words in names you define (programmer- or user-defined names).

INTRODUCTION

Syntax Notation

- In this manual, required reserved words, COBOL-86 system commands, and Z-DOS™ commands are shown in underlined capital letters, nonreserved words and operating system variables are lowercase. Filenames and nonrequired reserved words are shown in capital letters.

- The inclusion of all underlined reserved words is required unless the portion of the syntax in which they occur is itself optional. The characters < > and = are not underlined but are required when you use such formats. Reserved words that are not underlined are optional and serve only to improve the readability of the source program.

- Punctuation shown in user entries is required. Terminal periods required on each COBOL source statement are omitted in the syntax diagrams unless the nature of the syntax necessitates that it end a statement (e.g., STOP RUN.). At least one space is required anywhere one or more spaces are shown. Parentheses and commas are required in syntax related to subscripts.

- Words printed in lowercase letters in syntax lines represent generic terms (e.g., data-names) for which you must insert a valid entry in the source program.

- Any part of a statement or data description entry that is enclosed in square brackets is optional.

- In order to facilitate reference to lowercase words in the explanatory text, some of them are followed by a hyphen and a digit or letter. This modification does not change the syntactical definition of the word.

- Alternate options are depicted by placing the mutually exclusive choices in braces and separating them with a vertical stroke, (or by listing them vertically), e.g.:

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

™Z-DOS is a trademark of Zenith Data Systems Corporation.

Syntax Notation

- The ellipsis (...) indicates that the immediately preceding unit may occur once or any number of times in succession. A *unit* means either a single lowercase word, or a group of lowercase words and one or more reserved words enclosed in brackets or braces. If a term is enclosed in brackets or braces, the entire unit of which it is part must be repeated when repetition is specified.

- Comments, restrictions, and clarification on the use and meaning of every syntax line are contained in narrative immediately following the syntax diagram.

- COBOL-86 syntax and words that the computer displays on the screen are listed in this type style:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

1 2 3 4 5 6 7 8 9 0
```

Converting COBOL Programs to COBOL-86

The COBOL-86 version of the COBOL Compiler is different from previous releases of the compiler in two respects.

1. COMPUTATIONAL USAGE (COMP) has been changed to COMPUTATIONAL-0 (COMP-0). If you intend to use a COBOL program written for a different release of the compiler, all references to COMPUTATIONAL or COMP must be changed to COMPUTATIONAL-0 or COMP-0.

2. The default TAB table for COBOL-86 contains tab stops at the following columns: 8, 12, 20, 28, 36, 44, 52, 60, 68, and 73. Either you must change a program that was written using different tab stops, or you must modify the tab stops in the COBOL-86 TAB table. See Appendix B, "Customizations," for an explanation of how to modify the tab stops.

Learning More about COBOL

If you are new to COBOL programming, you will probably want to learn more about writing programs before using the COBOL-86 Compiler. The following texts are all COBOL tutorials, written for the novice programmer:

Abel, Peter. *COBOL Programming: A Structured Approach.* Reston, Virginia: Reston Publishing Company, 1980.

McCracken, Daniel D. *A Simplified Guide to Structured COBOL Programming.* New York: John Wiley & Sons, Inc., 1976.

Parkin, Andrew. *COBOL for Students.* Beaverton, Oregon: Edward Arnold, Ltd., 1978.

Part I
User's Guide

LANGUAGE OVERVIEW

CHAPTER 1 GETTING STARTED

Overview

The purpose of Part I, "User's Guide," is to help you get a COBOL program up and running on your computer. The steps necessary for using COBOL-86—compilation, linking, and execution—are described in the following chapters. This chapter provides an overview of the compilation process, lists the contents of your COBOL-86 disks, tells you how to perform disk backup; and presents a sample program development session.

The Compilation Process

Steps in the Compilation Process

The three major steps in compiling and executing a COBOL-86 program (see Figure 1.1) are:

compiling
linking
loading and executing.

1. The COBOL-86 Compiler consists of the main software module (COBOL.COM) and four phases or overlays (COBOL1.OVR through COBOL4.OVR). The routines contained in the compiler analyze your COBOL program and produce an object code file. This file will have a filename extension of .OBJ.

   Compilation is performed in two passes. The first pass creates an intermediate version of the program, which is stored in a binary work file called COBIBF.TMP; and the second pass creates the final version of the object code.

2. The object code produced by the compiler is not executable machine code. The Object Linker is responsible for producing the machine executable code, which will be placed in a file with a .EXE extension.

GETTING STARTED

The Compilation Process

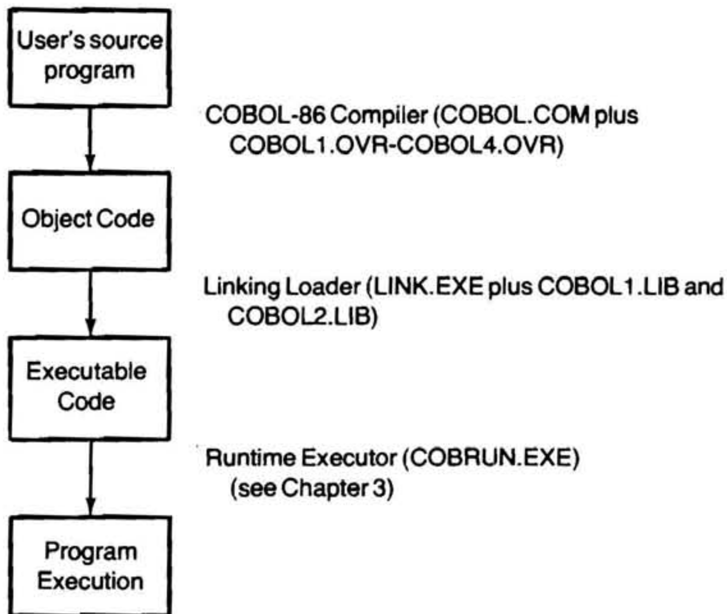


Figure 1.1. Major Steps in Compiling and Executing a COBOL Program

The Object Linker (linker) performs the following tasks:

- combines separately-produced object files
- searches library files for definitions of unresolved external references
- resolves external cross-references
- produces a printable listing of symbols
- produces the executable program.

3. The runtime system (COBRUN.EXE) "runs" the executable program.

GETTING STARTED

Your COBOL-86 System

The COBOL-86 system consists of parts entitled the "User's Guide," the "Language Overview," and "COBOL-86 Reserved Words," as well as appendices and an index contained in a single binder, and two distribution disks. The disks are organized in the following manner:

Disk I files contain the COBOL-86 Compiler and the runtime system.

COBOL.COM	the main compiler program
COBOL1.OVR	overlay 1
COBOL2.OVR	overlay 2
COBOL3.OVR	overlay 3
COBOL4.OVR	overlay 4
COBOL1.LIB	the runtime library of optional routines
COBOL2.LIB	the runtime library containing the routines necessary for loading COBRUN.EXE
COBRUN.EXE	the common runtime executor
COBDBG.OBJ	the Interactive Debug Facility
REBUILD.EXE	the utility for recovering damaged indexed files.

Disk II files are test and demonstration programs.

CTEST.COB	a test program for the color display system
CTEST.EXE	an executable version of CTEST.COB
CRTEST.COB	a test program for the terminal interface module
CRTEST.EXE	an executable version of CRTEST.COB
CENTER.COB	a test program for the COBOL-86 Compiler and runtime system
CENTER.EXE	an executable version of CENTER.COB.
DEMO.COB	a program to demonstrate the COBOL-86 screen section, to call the subprogram BUILD, and chain to program UPDATE
DEMO__01.OVL	an overlay file generated by linking DEMO
DEMO.EXE	an executable version of DEMO already linked with BUILD
BUILD.COB	a subprogram to create an indexed (ISAM) file of names, addresses, and telephone numbers
UPDATE.COB	a program to list or update the ISAM file created by BUILD
UPDATE.EXE	an executable version of UPDATE already linked.
README.DOC	contains current release information

GETTING STARTED

Disk Backup

Using FORMAT and COPY for Disk Backup

The first thing you should do when you receive your disks is make copies to work with and save the original disks for masters. This may be done by using the FORMAT, CONFIGUR, and COPY utilities supplied on your Z-DOS disk. The copies you make should include the operating system files and the Object Linker, LINK.EXE. To do so, follow these steps:

1. Place Z-DOS disk I in drive A and a blank disk in drive B.
2. Enter `FORMAT B:/S/V`
3. Enter `COPY LINK.EXE B:`
4. Replace the Z-DOS disk (in A) with COBOL-86 distribution disk I.
5. Enter `COPY *.* B:`
6. When the function is complete the disk in drive B will contain a bootable COBOL-86 system. Before this system is used with your printer, it must be properly configured with the CONFIGUR utility on Z-DOS disk I.

You can create a bootable copy of the test and demo suite by skipping step 3 and substituting COBOL-86 distribution disk II in step 4. If you wish to use EDLIN for development of your source programs, copy EDLIN.COM from Z-DOS disk I to a separate, formatted system disk in the same manner.

Using the Test Program

Having made backup copies, you should verify your copy of the compiler and runtime system by compiling, linking, and executing the test program CENTER.COB using your working copy. To do this, refer to the Sample Program Development session.

Sample Program Development

The compilation, linking, and loading/execution of a COBOL-86 program are described in detail in Chapters 2 and 3 of this guide. To give you an overview of the COBOL-86 system, however, the following sample session is provided. We recommend that you work through the sample session and then read Chapters 2 and 3 of this "User's Guide" before beginning to compile your own programs.

Sample Session

1. Organize your disks.
   Organize the files on your disks to minimize disk swapping and `Disk full` errors during program development. Usually COBOL-86 program development will require two working disks—one for your text editor and your source and object programs, one for the COBOL-86 Compiler, runtime executor, runtime libraries, the Object Linker, and any other necessary utilities. For example, if you used the backup procedure included in this chapter your two working disks will contain the following files:

PROGRAM DISK	COBOL-86 SYSTEM DISK	
Your COBOL-86 program	COBOL.COM	COBDBG.OBJ
EDLIN.COM (Source, object,	COBOL1.OVR	LINK.EXE
and executable files will	COBOL2.OVR	COBOL1.LIB
be placed on this disk)	COBOL3.OVR	COBOL2.LIB
	COBOL4.OVR	COBRUN.EXE
		REBUILD.EXE

Drives for Program and Compiler Disks

During development, the program disk will be kept in drive B, and the COBOL-86 system disk in drive A.

Drive B should be selected as the default drive (the one where new files are placed unless specified otherwise in the command). This arrangement simplifies access to the program files by placing them all on the same disk. Use of the programs on the COBOL-86 system disk will then require an explicit drive specification (e.g., `A:COBOL` or `A:LINK`).

2. Create the source program.
   In this sample session, we'll use CORN.COB as the filename for the source program. Of course, you may use any filename you wish for the COBOL-86 program you enter. It is up to you to supply the COBOL-86 source program. If you don't want to enter a program right now, skip this step and transfer the sample COBOL-86 program CENTER.COB from the COBOL-86 distribution disk to your program disk.

GETTING STARTED

Sample Program Development

a. After booting the system as usual, place your COBOL-86 system disk in drive A. Then place the program disk in drive B and select B as the default drive by typing:

```
B:
```

b. Type the command

```
EDLIN CORN.COB
```

to run the EDLIN editor program so you can write your COBOL-86 program.

c. When you have finished writing the program, use the EDLIN E command to place the file CORN.COB on your program disk and then exit to the operating system.

3. Check program syntax with trial compilation.
Before you go on, you can check your program for syntax errors with a "quick" compilation. This is done by compiling the program and displaying the error listing on the screen. No object or listing files are created, so compilation is faster than usual.

To compile CORN.COB and display a list of errors on the terminal, use the following command:

```
A:COBOL CORN,NUL;
```

(See Invoking the Compiler in Chapter 2, "Compiling COBOL Programs.")

If you get errors during the trial compilation (see Appendix E for a list of error messages), go back to step 2 and correct the source file (with the COBOL-86 system disk in drive A). When the trial compilation is completed without errors, you are ready to proceed to step 4.

Sample Program Development

4. Compile the source program.
   Now the program is ready to be compiled, which produces the object file. First, make sure the COBOL-86 system disk is in drive A and you are logged on to drive B. The compiler looks for the overlay phases (COBOL1.OVR-COBOL4.OVR) first on the disk in the default drive (drive B in this example) and then in drive A. With the disks arranged as in our example, the overlay phases will be found on drive A.

   To compile the program so that an object file (named CORN.OBJ) is produced, enter one of the following commands:

A:COBOL CORN;	produces just the object file
A:COBOL CORN,,PRN	produces an object file and printed listing
A:COBOL CORN,,CORN	produces an object file and a list file (named CORN.LST).

   When compilation is successfully completed, the message `No Errors or Warnings` is displayed, and the compiler exits to the operating system.

5. LINK the executable program.
   NOTE: The linker expects to find the COBOL-86 common runtime libraries (COBOL1.LIB and COBOL2.LIB) on the disk in the default drive (drive B in this case). If the libraries are not there, you will be prompted to specify the drive containing the disk on which they are located, unless you instruct the linker to look elsewhere. In this example, we will do just that by specifying drive A in the following link command. Now enter the command:

   ```
   A:LINK CORN,,,A:;
   ```

   and press RETURN.

   This command links the object file with the runtime system, producing the executable file. The `A:` at the end of the command line tells the linker to look on drive A for the COBOL libraries. (See Chapter 3 for a discussion of the linker commands.) The executable file (called CORN.EXE) is saved on the disk in drive B.

GETTING STARTED

Sample Program Development

Your program disk now contains the following files: CORN.COB, CORN.OBJ, CORN.EXE, and, if you requested a list file, CORN.LST.

6. LOAD and execute the program.
To run a program, you need the executable file (CORN.EXE) and the common runtime executor (COBRUN.EXE). COBRUN.EXE may be in either drive. The system will search for it first on the disk in the default drive, then in drive A.

In this example, CORN.EXE is on the program disk and COBRUN is on the COBOL-86 system disk in drive A. Since we are keeping the program disk in drive B, and drive B is selected as the default drive, type just the name of the executable file (the .EXE is not required).

Even though you've been very careful to remove all compiletime errors, you may still get runtime errors when the program is run. Error messages are described in Appendix E of this manual. If you get runtime errors, return to step 2 and edit the program to correct the errors.

CHAPTER 2 COMPILING COBOL PROGRAMS

Overview

As in Chapter 1, the sample commands in this chapter assume that: the COBOL-86 system disk is in drive A, your program disk is in drive B, and drive B has been selected as the default drive.

The COBOL-86 Compiler may be operated in one of the two ways listed in this chapter. Note that the discussions in Compiler Responses and Partial Command Strings of this chapter apply to both of these methods, and you should therefore read these descriptions before you begin to compile your own programs.

Operating the Compiler

1. You may operate the compiler by entering the command

   A:COBOL

   and pressing RETURN.

   (The drive specification is necessary because the compiler is not in the default drive.) Then reply to the following prompts. Filenames are discussed in Computer Responses of this chapter, on the following pages.

   a. `Source filename [.COB]:`

      Name of your source program. A filename must be specified. If no extension is specified, .COB will be appended by default.

   b. `Object filename [source filename.OBJ]:`

      Name of the object file to be created. The source filename is the default filename. The extension .OBJ is the default extension.

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

      Name of the file to which the program listing is to be written.

Operating the Compiler

If you enter a filename, its default extension is .LST. If you do not enter a filename, the default is NUL (no list). See The Source Listing File of this chapter for further discussion of the list file.

For example: The following series of responses compiles the source file CORN.COB, producing the object file CORN.OBJ and a listing file CORN.LST on the default drive:

```
A:COBOL
Source filename [.COB]:          CORN
Object filename [CORN.OBJ]:      press the RETURN key
Source listing [NUL.LST]:        CORN
```

2. The compiler can also be operated by entering

```
A:COBOL command string
```

where the command string contains

```
source filename, object filename, source listing
```

as explained for the first operation method and in Compiler Responses.

The separator character is the comma (,). No spaces are allowed.

Error Displays

When compilation has finished, you will be notified of any errors. If errors exist, you must locate and correct them in the source program and recompile before linking. If the compiler detected no errors, you will be told

```
No Errors or Warnings
```

and you may proceed with linking (Chapter 3).

Compiler Responses

Entering the Filename

When you use either of the above methods to operate the compiler, each of your responses can be the name of a disk file and/or system device. The format is:

Operating the Compiler

```
device filename extension
```

where: `device` is the name of a system device. This can be a disk drive, terminal, line printer, or other device supported by the operating system. If the device is a disk drive, the filename must also be given, unless a default filename is available (see final example in Partial Command Strings of this chapter). If the device is not a disk drive, only the device name is required. The device may be followed by a colon (:) for readability (it is required for disk drives). COBOL-86 recognizes the following device names:

NUL	Do not create
CON	Display on terminal
A: or B: ...	Disk drive (colon required)
PRN	Printer
AUX	RS-232.

`filename` is the name of the file on disk. If filename is specified without a device, the default disk drive is assumed as the device. Maximum length of the filename is 8 characters.

`extension` is a period (.) followed by a three-character suffix to the filename. If you do not specify an extension, the following defaults are assumed:

.COB for the source program file
.OBJ for the object file
.LST for the list file.

Partial Command Strings

Entering a Command String

You may also enter a partial command string when operating the compiler. Note that the default object filename may be specified by entering only the comma that normally follows the filename. Also note that if you enter the comma that follows the object filename, the source listing filename defaults to the source filename. You will be prompted for any files not specified in the command string. For example, the command

```
A:COBOL CORN,,
```

would (1) prompt you for the source listing filename (with the default name CORN.LST), (2) compile the source from CORN.COB, and (3) produce the object file CORN.OBJ.

COMPILING COBOL PROGRAMS

Operating the Compiler

Each prompt displays its default, which you may accept by pressing RETURN or override by entering another filename or device name.

If you enter an incomplete command string followed by a semicolon (;), default entries will be assumed for the unspecified files.

The following examples assume the compiler is on drive A and that drive B has been selected as the default drive:

`A:COBOL CORN;`	Compiles the source from CORN.COB and produces the object file CORN.OBJ. No listing file is produced.
`A:COBOL CORN,;`	Performs exactly the same functions as the previous example.
`A:COBOL CORN,,;`	Compiles the source from CORN.COB and produces the files CORN.OBJ and CORN.LST. (The second comma (,) tells the compiler to use the source filename as the default list filename.)
`A:COBOL CORN,,CON`	Compiles the source from CORN.COB and places the source listing file on the terminal. The object program is CORN.OBJ.
`A:COBOL CORN,CORNOBJ,PRN`	Compiles the source from CORN.COB, places the list file on the printer, and places the object into CORNOBJ.OBJ.
`A:COBOL A:CORN,CORNOBJ,A:;`	Compiles CORN.COB from disk A, places the object into CORNOBJ.OBJ on the disk in drive B, and places the listing into CORN.LST on the disk in drive A.

COMPILING COBOL PROGRAMS

Using Compiler Switches

/ Used to Indicate Switch

You can add one or more switches to the compiler command string or at the end of any interactive response. A switch is indicated by a slash (/). The switches and their effects are described here.

The syntax for a command string with switch(es) is:

```
drive:COBOL command string/switch(es)
```

Switches

/C Ordinarily, the compiler looks for the four overlay files (COBOL1.OVR through COBOL4.OVR) on the default drive, then it looks on drive A. To override the default drive, use the /C switch with the letter of the drive you want. (The colon is not required in the switch.)

Example: `A:COBOL CORN,,/CB`

In this example, the compiler looks for the overlay files on drive B.

/T The compiler puts its intermediate file COBIBF.TMP on the default drive unless you use the /T switch followed by the desired drive designation. The disk in the drive you specify must not be write-protected.

This option is particularly helpful for compiling very large programs on systems with more than two drives (see Compiling Large Programs in this chapter).

Example: `A:COBOL CORN,,A:CORNLIST/TC`

In this example, the intermediate file is placed on drive C. (The colon is not required in the switch.)

COMPILING COBOL PROGRAMS

Using Compiler Switches

/P Each /P allocates an extra 100 bytes of stack space for the compiler's use. Use /P if stack overflow errors occur during compilation.

Example: `A:COBOL CORN/P/P/P;`

In this example, 300 extra bytes of stack space are allocated.

/D This switch suppresses both generation of the debug information file (.DBG) and source line numbers, which are normally placed in the object file. The result is PROCEDURE DIVISION code that is about 16% shorter. However, when this switch is used, the run-time system will not be able to note the line number at which an error occurs. (See Chapter 7 for a discussion of the debug information file.)

Example: `A:COBOL CORN/D;`

In this example, the object file will not contain source line numbers.

/Fn Fn (FIPS) flagging lets you tell the compiler to output a warning for each COBOL statement above the Federal Information Processing Standard level (n). The n must be a digit from 0 through 4 (4 is the default):

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.

Example: `A:COBOL CORN/F1;`

In this example, the compiler will display a warning for each COBOL statement above low intermediate level. If you create a source listing file, the warning will be included with the error messages.

The Source Listing File

The source listing file is a line-by-line account of the source file(s) with page headings and error messages. Each source line is preceded by a four-digit decimal number. This number will be referenced by any error messages pertaining to that source line.

Files that are included in the compilation via COPY statements in the source file are also included in the listing.

Compiler error messages are shown at the end of the listing file (as well as being displayed on the terminal). See Appendix E for a list and explanation of error messages.

Compiling Large Programs

Occasionally a COBOL-86 program may be too large to compile in the available memory space or may exhaust the available disk space. There are four ways you can take care of this problem with COBOL-86:

1. Use the /D switch in your command string (see Using Compiler Switches in this chapter) to prevent generation of a debug information file and to suppress generation of line numbers in the object file.

2. Use the /T switch in your command string (see Using Compiler Switches) to place the intermediate file (COBIBF.TMP) on a separate disk.

3. Break the program into several program modules. These modules can be separately compiled and then combined into one program by the linker. See Appendix A, "Interprogram Communication," for information on using program modules.

4. Break the large program into several smaller programs that are chained. These programs are separately compiled and linked. See Appendix A, "Interprogram Communication," for information on chaining programs.

Compiling Large Programs

NOTE: If you want to check the contents of your disk to make sure that COBIBF.TMP has been deleted after compilation is completed, use the DIR operating system command. Then, to make sure the space has been released, use the CHKDSK program supplied with your operating system. CHKDSK reclaims available space from unclosed files and tells you the total amount of available space on the disk.

CHAPTER 3 LINKING AND EXECUTING COBOL PROGRAMS

Overview

As in previous chapters, this discussion assumes that: the COBOL-86 system disk is in drive A, the program disk is in drive B, and drive B has been selected as the default drive.

The linker converts the compiled object version of your program (the object file) into a version that is executable (the run file). To do so, it searches the disk in the default drive for the COBOL-86 runtime libraries COBOL1.LIB and COBOL2.LIB, which make up part of the common run-time system (described later in this chapter). COBOL1.LIB is a library of optional routines that may be required for running the program, and COBOL2.LIB contains the routines that are always necessary for running the program. The routines you need are then linked to the object version of your COBOL-86 program. The routines you need depend on which COBOL-86 language features you used in the program and program modules.

The linker can also be used to combine separately compiled program modules into one program. The modules may be specified individually or extracted from a library. They may be written in COBOL-86 or in Macro-86 Assembler language. See Linking Program Modules in this chapter for details.

Files that are to be linked or that will contain linker output can be specified in one of three ways: interactively, as part of the command line, or as a command file.

Using Link

To operate the linker, use one of the following procedures:

1. To specify files interactively, enter

   <u>A:LINK</u>

   (The device specification is necessary because LINK is not in the default drive.) Then reply to the following prompts:

LINKING AND EXECUTING COBOL PROGRAMS

Using Link

a. `Object Modules[.OBJ]:`

Name(s) of object file(s). If you do not specify an extension, .OBJ will be used. If multiple object files are linked, they must be separated by a plus (+).

Files that are to be linked must be in object format. (If they were compiled with COBOL-86 or generated by the Macro-86 Assembler, they will already be in object format.)

b. `Run File[object filename.EXE]:`

Name of file to contain executable code. The object filename is the default filename. The extension .EXE cannot be overridden.

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

Name of list file. Defaults work much the same way as in the compiler. The default is no list file, unless the run file is followed by a comma (see the discussion of partial command strings). If the run file is followed by a comma, the default list filename is the object filename, with the default extension .MAP.

d. `Libraries[.LIB]:`

`Libraries` refers to the runtime routines that COBOL-86 may need to run your program. All of these routines are included in COBOL1.LIB and COBOL2.LIB.

Normally you only have to press RETURN following this prompt. The names of the libraries are supplied to the linker by the COBOL-86 object file. If you wish however, you may specify your own libraries (see LIB documentation in *Z-DOS* manual), that will be searched before the COBOL-86 libraries.

The linker assumes that the COBOL-86 libraries are in the default drive. If they are not in the default drive, you must enter a drive specification, regardless of which drive you have selected as the default drive.

LINKING AND EXECUTING COBOL PROGRAMS

Using Link

In all of our examples, the libraries are on drive A and not the default drive. Therefore, you should indicate the drive specification for the libraries. If you don't, the linker will prompt you for the drive on which the libraries are located.

Filenames are specified in the same way as for the compiler (see Chapter 2), except that the default extension is always .EXE for the run file produced by the linker.

For example: The following series of responses links the files CORN.OBJ and MYOBJ.OBJ and searches your library MYLIB1.LIB before searching COBOL1.LIB and COBOL2.LIB. The linker produces the executable file MYRUN.EXE and the source listing file MYLIST.MAP.

```
A:LINK
Object Modules[.OBJ]:   CORN+MYOBJ
Run File[CORN.EXE]:     MYRUN
List File[NUL.MAP]:     MYLIST
Libraries[.LIB]:        MYLIB1+A:COBOL1+A:COBOL2
```

2. To use a command string, enter

```
A:LINK command string
```

where the command string contains

```
objfile(s), runfile, listfile, libfile(s)
```

as defined before.

You must specify an object filename. For the other files, a default filename may be selected in the command string by entering only the comma that would normally follow the filename (see examples following).

LINKING AND EXECUTING COBOL PROGRAMS

Using Link

As with the COBOL-86 Compiler, you may enter a partial command string or the entire string. If you specify an entry for all four files, or if an incomplete command string ends with a semicolon (;), linking will proceed without further prompting. Otherwise, the linker prompts for the remaining unspecified files. Each prompt displays its default, which you may either accept (by pressing RETURN) or override (by entering another filename and/or device name).

Examples (COBOL-86 system disk is in drive A, default drive is B): Since the COBOL libraries are in drive A, and the default drive is drive B, the linker will not find the libraries unless you specify the drive for the libraries or respond with an A drive designation to the linker prompts. In these examples, we have specified the library on drive A, unless indicated otherwise.

```
A:LINK CORN;
```

links CORN.OBJ and puts the runfile into CORN.EXE. No list file is produced. If CORN.OBJ was produced by the COBOL-86 Compiler, the linker prompts for the drive on which COBOL1.LIB and COBOL2.LIB are found. Type A in response to the prompt.

```
A:LINK CORN,,,A:;
```

operates the same as the first example, except that a listing is produced in CORN.MAP. (The second comma (,) indicates that the object filename is to be used as the default list filename.) The A: at the end of the command line tells the linker to find the COBOL-86 libraries on drive A instead of the default drive.

```
A:LINK CORN+SUBFILE1+SUBFILE2,,,A:;
```

operates the same as the previous example, except that SUBFILE1.OBJ and SUBFILE2.OBJ will be linked with CORN.

3. You can also create one or more files that contain responses to the linker prompts. They are especially useful when you are linking a number of object modules more than once (during debugging, for example), or when you are developing variations of a program. See Chapter 4 of this manual or the *Z-DOS* manual for details on creating response files.

   To specify this option on the command line, use the command:

   ```
   A:LINK @filename
   ```

   where filename is the name of your response file. You must include the drive if the file is not on the default drive. You may also specify a file extension.

   Example: `A:LINK @RESFIL.LNK`

After the command line is entered, the linker starts. If the linker needs more memory space to link your program than is in the computer, it will create a file called VM.TMP on the disk in the default drive and will display a message to that effect.

CAUTION: Do not remove this disk during linking. If this additional space is used up, or if the disk containing VM.TMP is removed before linking is completed, the linker will abort.

When the linker has finished, VM.TMP will be erased from the disk, and any errors that occurred during linking will be displayed. (Error messages are listed in the *Z-DOS* manual.) The run file will be stored (with the extension .EXE) on the disk in the default drive or in the specified drive.

NOTE: If you want to check the contents of your disk to make sure that VM.TMP has been deleted after a linker abort, use the DIR operating system command. Then, to make sure the space has been released, use the CHKDSK program supplied with your operating system. CHKDSK will reclaim available space from unclosed files and tell you the total amount of available space on the disk.

LINKING AND EXECUTING COBOL PROGRAMS

Linking Programs That Use Overlays

The COBOL-86 segmentation facility lets you run programs that are larger than the computer's central memory. Segmented programs have overlays that are referenced by COBOL-86 section numbers greater than 49 (see Segmentation, in Part II). Each section is an independent segment.

.OVL File Format After Linking

No special user commands are required for linking a segmented program. The linker creates a file for each independent segment of the program, with the filenames in the format:

PROGIDnn.OVL

where

- PROGID is the PROGRAM-ID that you defined in the IDENTIFICATION DIVISION. If the PROGRAM-ID is less than six characters, COBOL-86 extends it to six characters by adding underlines (_) to the end.

- nn is a two-digit hexadecimal number that is computed by subtracting 49 (decimal) from the program segment number (decimal).

Example: If the PROGRAM-ID is "SAMPLE" and the program contains segment number 99 (decimal), an overlay segment will be produced with the name SAMPLE32.OVL.

Linking Program Modules

Object File

If you have developed your program as separately compiled program modules, the linker can combine the modules into one program.

Before linking, compile or assemble all modules so that you have an object version of each. Then start the linker, specifying in the command string each module you want to link.

Example: `A:LINK CORN+SUBFILE1+SUBFILE2,,,A:;`

See Appendix A, "Interprogram Communication," for more information about linking program modules.

LINKING AND EXECUTING COBOL PROGRAMS

Linking Large Programs

This discussion assumes that your files are arranged on two disks as in the Sample Session in Chapter 1.

Methods for Separating Files

If your program disk will not hold all the object files, the run file, and the list (.MAP) file, you will need to separate the files. One of the following methods should take care of this problem.

1. Do not request a list file (.MAP). Accept the no list default, or specify NUL as its name, i.e., `A:LINK CORN,,NUL,A:`

2. Send the list file (.MAP) to the terminal (CON) or printer (PRN).

3. Copy all the .OBJ files to a separate disk. Use this disk in place of the program disk.

4. Break the program into several programs that are chained. Compile and link each program separately. Note that the common runtime system works very efficiently with CHAIN; it only needs to be loaded once, rather than once for each program in the chain. See Appendix A, "Interprogram Communication," for more information on chaining.

5. Break the program into program modules connected by CALL statements. Compile the modules separately and link them together using the linker. This procedure is similar to CHAIN except that the called program contains a return instruction. See Appendix A, "Interprogram Communication," for instructions on linking program modules.

6. Purchase more system memory and/or disk storage.

NOTE: If you want to check the contents of your disk to make sure that VM.TMP has been properly deleted after a linker abort, use the DIR operating system command. Then, to make sure the space has been released, use the CHKDSK program supplied with your operating system. CHKDSK will reclaim available space from unclosed files and tell you the total amount of available space on the disk.

LINKING AND EXECUTING COBOL PROGRAMS

Executing COBOL Programs

After your COBOL program has been compiled and linked successfully, the final step is loading and execution. These functions are performed by specifying the name of the executable file to the operating system, as explained here.

Using the Runtime Executor

Your runtime executor (COBRUN.EXE) is loaded automatically at the beginning of execution. When you begin execution, COBRUN.EXE must be in either the default drive or drive A.

Entering Run File Name

To run your program, just enter the name of your run file, without the .EXE filename extension. For example, if your program is in the current drive, type:

```
CORN
```

Execution of CORN.EXE should begin immediately.

CHAPTER 4 BATCH COMMAND FILES

.BAT Extension Required

Z-DOS, the operating system that COBOL-86 uses, allows you to create a batch file for executing a series of commands. This file must have the extension .BAT. It should be kept on the program disk.

As shown in the example that follows, the batch file may contain symbols that refer to parameters in its command line. The symbol %1 refers to the first parameter on the line, %2 to the second parameter, etc. The limit is %9. In the example below, %1 refers to the parameter, sourcefile.

The batch file may also pause, display a prompt (that you have defined), and wait for you to continue. The PAUSE command, followed by the text of the prompt, performs this function.

Expediting Compilation

If your program is already debugged and you are making only minor changes to it, you can speed up the compilation process by creating a batch file that issues the compile, link, and run commands.

For example, use the EDLIN editor to create the batch file CLGO.BAT (for "compile, link, and go"). The text of the file might be:

```
A:COBOL %1;
A:LINK %1,,NUL,A:;
%1
```

To execute this file, type

```
CLGO sourcefile
```

where sourcefile is the name of the source program you want to compile, link, and run. The first line of the batch file compiles the program; the second line links the object file; and the third runs the executable file.

NOTE: A .BAT file is only executed if there is neither a .COM file nor EXE file with the same name.

For more information about batch command files, see the *Z-DOS* manual.

CHAPTER 5 DATA INPUT AND OUTPUT

Overview

A COBOL program can read or write data to files on disk or to other Z-DOS devices. The instructions for creating and using these files are entered as part of the COBOL-86 source program. This chapter explains disk files and other types of files and tells you how to use them with your COBOL-86 programs. See Part II of the manual for more information.

To specify that a disk file is to be used in a program, include the ASSIGN TO DISK clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

The filename of the disk file must be declared in the VALUE OF FILE-ID clause in an FD paragraph, in the FILE SECTION of the DATA DIVISION. The FD paragraph must also include the clause LABEL RECORDS ARE STANDARD. BLOCK clauses are checked for syntax but have no effect on any filetype. The FILE-ID must not be one of the Z-DOS device names listed in Using Z-DOS and Nondisk Files in this chapter.

Using Disk Files

Disk files may be organized in one of four ways:

Sequential
Line sequential
Relative
Indexed.

Specifying the File Organization

When a COBOL-86 program reads from or writes to a disk file, the ORGANIZATION clause in the FILE-CONTROL paragraph of the program's ENVIRONMENT DIVISION must specify the file organization of the disk file, unless it is sequential. Disk files are assumed to be sequential unless declared otherwise.

Note also that only line sequential files can be created with a text editor. All others must be created by a COBOL-86 program or assembly language program. See Part II of the manual or one of the tutorials recommended in the "Introduction" to this manual for more information about creating disk files.

DATA INPUT AND OUTPUT

Using Disk Files

Types of Disk Files

Following is a list of the four types of disk files. All formats are subject to change without notice.

1. Sequential files have a two-byte count of the record length followed by the actual record, for as many records as are in the file.

2. In Line sequential files the record is followed by a return/line feed delimiter, for as many records as are in the file.

   Both sequential organizations pad any remaining space in the last physical block with CTRL-Z characters, indicating end of file. To make maximum use of disk space, records are packed together with no unnecessary bytes in between.

3. Relative files always have fixed length records of the size of the largest record defined for the file. Since no delimiter is needed, none is provided. Deleted records are filled with hex value "00." Additionally, six bytes are reserved at the beginning of the file to contain system bookkeeping information.

4. Each indexed file declared in a COBOL-86 program will generate two disk files: a key file and a data file. The file specification in the VALUE OF FILE-ID clause specifies a file containing data only. The filename included in the file specification is appended with an extension .KEY to form the file specification of the key file.

   Key Files

   The *key file* contains keys, pointers to keys, and pointers to data. The format of this file is very complicated, but follows the guidelines for a prefix B + tree.*

   Types of Granules

   A key file is divided into 256 byte units, called *granules*. There are five possible granule types. A type indicator is located in the first byte of each granule. The granule type indicators have the following values:

*See Comer, Douglas. "The Ubiquitous B-Tree." *Computing Surveys of the ACM*, Vol. 11, no. 2 (June 1979), pp. 121–137.

Using Disk Files

Value	Type Indicator
1	Data Set Control Block
2	Key Set Control Block
3	Node
4	Leaf
5	Deleted granule

The key file will have 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.

Each data set control block and key set control block contains, in the fourth byte, a "damaged" flag that notifies you when the last file use was not terminated properly. The runtime executor sets these flags to nonzero values when the file is opened for updating and restores them to zero when the file is closed.

Data Files

The *data file* consists of data records. Each data record is preceded by a two-byte long field and a one-byte "reference count" that indicates whether a record has been deleted. The data file is terminated by a control record with a length field containing a 2, followed by two bytes of HIGH-VALUES.

Using Z-DOS and Non-disk Files

Files that will only be output need not be placed on a disk, but should be considered as a stream of characters going to a printer or other device. No permanent file is created. Records should be defined as the fields to appear on the output device. No extra characters are needed in the record for carriage control. Return, line feed, and form feed are sent to the output device between lines. Note, however, that blank characters (spaces) on the end of a print line are truncated to make printing faster.

To send an output file to the printer, use the SELECT filename ASSIGN TO PRINTER clause. Then in an associated FD, specify the clause LABEL RECORD IS OMITTED. Do not specify the VALUE OF FILE-ID clause.

DATA INPUT AND OUTPUT

Using Z-DOS and Non-disk Files

Character Device Names

Z-DOS provides special device names for character devices. Data may be sent to or read from the following devices:

CON display on terminal
AUX serial port (RS-232)
PRN printer

If you assign these names to the VALUE OF FILE-ID clause, COBOL-86 treats the files as disk files. That is, you assign the files to disk with the SELECT clause, and the operating system uses the designated device.

CHAPTER 6 THE INTERACTIVE DEBUG FACILITY

Overview

The COBOL-86 Interactive Debug Facility allows you to control the execution of a program and to examine or change data items in a COBOL-86 program. When a program is compiled, a "debug information file" is created along with the object file. The information file contains line numbers and data-names from the DATA DIVISION and PROCEDURE DIVISION of your COBOL-86 program. The debug commands listed in this chapter can use these line numbers and data-names to affect data items and program execution in a number of ways.

The compiler will create the debug information file with the filename of the COBOL-86 source file, but with the extension .DBG. For example, compilation of a source file named MYFILE would produce MYFILE.OBJ (object file) and MYFILE.DBG (debug information file).

To suppress creation of a debug information file, use the /D compiler switch (see Using Compiler Switches in Chapter 2).

The Debugging Procedure

Requesting Debug

To use the Interactive Debug Facility, include the file COBDBG.OBJ in the command line when you link your program. For example:

```
A:LINK MYFILE+A:COBDBG,,NUL,A:
```

enables the debug facility. When you issue the command to execute your program (MYFILE, in this example), the following message will be displayed:

```
COBOL-86 Interactive Debug Facility v. xxx
Program: MYFILE
  Type help for list of commands
*
```

The Debugging Procedure

***Prompt**

The asterisk prompt (*) indicates that the debug facility is ready to accept any of the debug commands. The debug information file should be on the current disk. If is it not, the message

```
**No debug information file found
```

will follow the messages already displayed.

Note that without a debug information file, limited debugging is possible. By simply including COBDBG.OBJ in the linker command line, you can enable the Interactive Debug Facility and execute any of the debug commands except CHANGE, EXHIBIT, and GOTO line-number. However, without the debug information file, the debug facility cannot verify that line numbers specified in the BREAKPOINT command are valid PROCEDURE DIVISION line numbers that contain statements or section or paragraph names.

Use of Debug Commands

Syntax of Debug

Debug commands, as shown in Table 6.1, may be typed in full or abbreviated to the first letter of the command name. Upper- and lowercase are equivalent. Arguments to the commands (line numbers, data-names, ALL, OFF) must be given in full. Though spaces are shown, arguments can be separated from commands by any nonalphabetic character. When a numeric argument is expected, the debug facility will scan until the first digit on the line is found. For example, the following list of commands are all equivalent (they all set a breakpoint at line 100):

```
Breakpoint 100
BREAK @100
b100
break for me at line 100, if you would please
```

Use of Debug Commands

Table 6.1. Functions of the Interactive Debug Facility

FUNCTION	DESCRIPTION
ADDRESS [data-name]	Display absolute address (hexadecimal) of a data item in memory.
BREAKPOINTS	List all breakpoints.
BREAKPOINT line-num	Set a breakpoint at line-number. You may have up to 8 breakpoints set at any given time. Debug verifies that line-number is a PROCEDURE DIVISION line that contains a statement or a section or paragraph name.
CHANGE data-name	Display the contents of data-name* and allow a new value to be entered.
DUMP [addr1[,addr2]]	Display memory addresses (hexadecimal and ASCII equivalents) from addr1 through addr2.
EXHIBIT data-name	Display contents of data-name.*
GO	Resume execution from the last breakpoint or current program position until a breakpoint or end of program is encountered.
GOTO line-num	Begin execution at line-number; continue until breakpoint or end of program is encountered.
HELP	Display the list of debug commands.
KILL line-num	Remove the breakpoint at line-number.
KILL All	Remove all breakpoints from the breakpoint list.
LINE	Display the line-number of the current line.
QUIT	Terminate the program (closing all open files).
STEP	Execute one statement.
TRACE	Set trace mode. When trace mode is set, the line number of each line will be displayed as the line is executed.
TRACE Off	Turn off trace mode. This command sets trace mode off. (See description of Trace.)

*Subscripted variables cannot be used as data-names.

Part II
Language Overview

CHAPTER 7 INTRODUCTION

Overview

Parts II and III of this manual comprise a detailed reference guide to the COBOL language as set forth in the 1974 ANSI definition and implemented in the COBOL-86 Compiler. Students of COBOL are cautioned that this reference guide assumes a level of familiarity with the language and is not intended to be a substitute for a good tutorial textbook. Throughout Parts II and III, ANSI standard information is presented in parallel with extensions and variations peculiar to this compiler.

Part II treats elements of the COBOL language that are fundamental and pervasive in nature. For the most part, it avoids dealing with the syntax and function of specific reserved words. This material can be found in Part III, which is organized as an alphabetical guide for quick access to reserved words.

Exceptions are made, however, in the case of indexed and relative files. Because the reserved words pertinent to the handling of these file types are highly interrelated in their effects, a coherent explanation of their use cannot be achieved by treating each word in isolation. Therefore, Chapters 10 and 11, respectively, describe in a comprehensive manner the use of indexed and relative files. The reserved words treated there are excluded from Part III unless they have an additional function not related to these filetypes.

The figurative constants are also treated in Part II, since they represent values rather than logical functions and are used throughout COBOL programs in a variety of ways.

The ANSI Standard

Processing Modules

COBOL-86 is based upon American National Standards Institute X3.23-1974. Elements of the language are allocated to twelve functional processing modules:

1. Nucleus
2. Table Handling
3. Sequential I/O
4. Relative I/O

INTRODUCTION

The ANSI Standard

5. Indexed I/O
6. Interprogram Communication
7. Library
8. Communication
9. Debug
10. Report Writer
11. Segmentation
12. Sort/Merge.

Implementation Levels

Each module has two defined levels of implementation—Level I and Level II. In order to be called COBOL, a system must provide at least a Level I implementation of the Nucleus, Table Handling, and Sequential I/O modules. In general, COBOL-86 provides all of Level I plus a substantial portion of Level II implementation for eight of the twelve modules. Specifically:

1. Nucleus—all Level I and II plus USAGE COMP-3 or COMP-0 and additional extensions to ACCEPT and DISPLAY. The following, however, are excluded:

   a. no figurative constant ALL with an operand length greater than 1
   b. no qualifiers in ENVIRONMENT DIVISION
   c. no Switch Testing Facility (Level I feature)
   d. no alphabet-name other than ASCII
   e. no level 88 conditions with lists and ranges intermixed
   f. no unsigned COMP items
   g. no RENAMES phrase
   h. no use of MOVE, ADD, or SUBTRACT CORRESPONDING
   i. no multiple destinations in arithmetic statements
   j. no REMAINDER
   k. no arithmetic expressions in conditions
   l. no ALTER with multiple procedure names.

   Additionally, INSPECT is implemented at Level I.

2. Table Handling—All Level I, plus full Level II SEARCH formats, except no OCCURS DEPENDING ON.

The ANSI Standard

3. Sequential I/O—All Level I plus these Level II:

    a. RESERVE clause
    b. OPEN and CLOSE with multiple operands and individual options per file
    c. VALUE OF FILE-ID IS data-name
    d. OPEN EXTEND
    e. WRITE ADVANCING data-name
    f. LINAGE phrase
    g. AT END-OF-PAGE clause.

4. Relative and Indexed I/O—All Level I plus these Level II:

    a. RESERVE clause
    b. OPEN and CLOSE with multiple operands and individual options per file
    c. VALUE OF FILE-ID IS data-name
    d. DYNAMIC access mode with READ NEXT
    e. START with key relations EQUAL, GREATER, or NOT LESS.

5. Interprogram Communication—Level I.

6. Library—Level I.

7. Communication—not implemented.

8. Debug—not implemented (however, numerous debugging extensions are implemented, including the IBM COBOL Debug Facility, READY TRACE procedure tracing feature, and WITH DEBUGGING MODE in SOURCE-COMPUTER paragraph).

9. Report Writer—not implemented.

10. Segmentation—Level I.

11. Sort/Merge—not implemented.

Coding Fundamentals

In addition, several exclusions exist in the file-handling modules, most of which pertain to tape systems (which are not supported). They are:

1. no multiple index keys
2. no SELECT OPTIONAL filename clause
3. no functional RESERVE integer AREAS clause
4. no MULTIPLE FILE TAPE CONTAINS clause
5. no fully functional BLOCK CONTAINS or RECORD CONTAINS clauses
6. no Level I Rerun facility
7. no multireeling, tape reversal, or tape positioning through the use of OPEN or CLOSE.

Character Set

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

1. Letters A through Z
2. Blank or space
3. Digits 0 through 9
4. Special characters:
   a. + Plus sign
   b. − Minus sign
   c. * Asterisk
   d. = Relational sign (equals)
   e. > Relational sign (greater than)
   f. < Relational sign (less than)
   g. $ Dollar sign
   h. , Comma
   i. ; Semicolon
   j. . Period or decimal point
   k. " Quotation mark
   l. (Left parenthesis
   m.) Right parenthesis
   n. ' Apostrophe (alternate of quotation mark)
   o. / Slash.

Punctuation of Source Code

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

Punctuation

The following characters are used for punctuation:

1. (Left parenthesis
2.) Right parenthesis
3. , Comma
4. . Period
5. ; Semicolon.

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

1. As punctuation, a period, semicolon, or comma should not be preceded by a space, but *must* be followed by a space.

2. At least one space *must* appear between two successive words and/or literals. Two or more successive spaces are treated as single space, except in non-numeric literals.

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

4. When you use a period, comma, plus, or minus character in the PICTURE clause, it is governed solely by rules for report items.

5. A comma may be used as a separator between successive operands of a statement or between two subscripts.

6. A semicolon or comma may be used to separate a series of statements or clauses.

INTRODUCTION

Coding Fundamentals

Word Formation

Legal Characters

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

1. 0 through 9 (digits)
2. A through Z (letters)
3. – (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. All words are either *reserved words*, which have preassigned meanings, or programmer-supplied names. 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.

Source Line Structure

Coding Rules for COBOL-86 Programs

Since COBOL-86 is a subset of American National Standards Institute (ANSI) COBOL, programs may be written on standard COBOL coding sheets, and the following rules are applicable. If you are not familiar with the standard COBOL coding form, a sample is shown in Figure 8.2.

1. Each line of code may have a six-digit number in columns 1-6, such that the line numbers are 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 columns 12-72, governed by the other rules of statement punctuation.

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

5. Explanatory comments may be inserted on any line within a source program by placing an asterisk in column 7 of the line. The line will be produced on the source listing but serves no other purpose. If a slash (/) appears in column 7, the associated line is treated as a comment and will be printed at the top of a new page when the compiler lists the program. See Part III, TRACE and EXHIBIT, for use of 0 in column 7.

6. Any program element may be "continued" on the following line of a source program. The rules for continuation of a non-numeric ("quoted") literal are explained in Chapter 8. Any other word or literal or other program element is continued by placing a hyphen in the column 7 position of the continuation line. The effect is concatenation of successive word parts, exclusive of all trailing spaces of the last predecessor word and all leading spaces of the first successor word on the continuation line. On a continuation line, Area A must be blank.

7. Tab stops are set at columns 8, 12, 20, 28, 36, 44, 52, 60, 68, and 73 unless the compiler has been modified as described in Appendix B.

Program Structure

Every COBOL source program is divided into four required divisions, namely: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. Each division begins with a division header in the form division-name DIVISION. Divisions are themselves subdivided into smaller logical units. Listed in descending order of precedence, these subdivisions are: section, paragraph, sentence, clause, phrase, and word.

Because the first two divisions are relatively limited in scope, complete information regarding them can be found by looking up the words IDENTIFICATION, ENVIRONMENT, CONFIGURATION, and INPUT-OUTPUT in Part III. (CONFIGURATION and INPUT-OUTPUT are sections within the ENVIRONMENT DIVISION.)

INTRODUCTION

Coding Fundamentals

Chapters 8 and 9 detail the information needed to write the DATA and PROCEDURE divisions.

Program 7.1 shows the structure of a complete, generic COBOL program as it would exist if the programmer made use of all available structural options.

Program 7.1. Generic COBOL Program

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

CHAPTER 8 DATA DIVISION

Overview

The DATA DIVISION allocates storage, defines the format, and specifies the names for data items used in a COBOL program. It is subdivided into four sections:

1. File Section
2. Working-Storage Section
3. Linkage Section
4. Screen Section.

Although each section is optional, they must occur in the order listed. Each section is treated in Part III, which is composed of Chapter 12, "Alphabetical Reserved Word List." In Part III, each section can be located by its respective section name.

Data Items

Structures and Types

Several kinds of data items can be defined and utilized in a COBOL program. Data items are distinguished both by their structure and their type. The four structures are:

1. group items
2. elementary items
3. stand-alone items
4. condition-names.

The five types are:

1. alphanumeric
2. alphanumeric-edited
3. numeric
4. numeric-edited (report)
5. index.

Although COBOL-86 accepts a sixth type, alphabetic, in source code statements, it is treated internally as an alphanumeric.

NOTE: For a complete treatment of condition-names, see VALUE in Part III; index data items can be found under OCCURS in Part III. Structures and Types

DATA DIVISION

Data Items

Data Item Structures

Group Items

A *group item* is one having further subdivisions, such that it contains one or more elementary items. In addition, a group item may contain other groups. An item is a group item if, and only if, its level number is less than the level number of the immediately succeeding item (see Level Numbers in this chapter). Group items use level numbers in the range 1-49. The maximum size of a group item is 4095 characters.

Elementary Items

An *elementary item* is a data item contained within a group, but which contains no subordinate items. Elementary items use level numbers in the range 2-49.

Stand-Alone Items

A *stand-alone item* is identical to an elementary item in every respect, except that it is not contained within a group. Stand-alone items are identified by level number 77.

Data Item Types

Alphanumeric and Alphanumeric-edited

Alphanumeric items store character string values. Any character or combination of characters constitutes a valid string. Alphanumeric items are not valid operands in arithmetic operations, even if their contents are entirely numeric. If special editing characters are included in the item's field description, it becomes an *alphanumeric-edited item*. For a detailed treatment of the editing characters, see PICTURE in Part III.

Numeric and Numeric-edited

Numeric items store quantitative values in a format suitable for arithmetic processing. Only the digits 0-9 and S, V, or P editing characters may be included in a numeric field. The maximum field size is limited to 18 digits.

If special editing characters ($. , + −, etc.) are included in the field description, it becomes a numeric-edited item. For a detailed treatment of the editing characters, see PICTURE in Part III. Special editing characters may not extend the field size to more than 30 characters. A numeric-edited item may act as a receiving field in an arithmetic process, but may not act as a sending field.

Internal Storage Formats

Actually, three distinct internal formats are available for storage of numeric items. The format you choose is governed by an optional USAGE clause, with USAGE DISPLAY assumed if the clause is omitted (see USAGE in Part III). The details of each format follow.

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

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

An external decimal item corresponds to DISPLAY usage.

Internal Decimal Item. An *internal decimal item* is one that is stored in packed decimal format.

A packed decimal item will occupy a number of bytes equal to the next larger whole number than the result of dividing the number of characters by 2. For example,

PIC S9(5).
5/2 = 2.5. Bytes used = 3.
PIC S9(6).
6/2 = 3. Bytes used = 4.

All bytes except the right-most 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.

The packed decimal format corresponds to COMPUTATIONAL-3 (COMP-3) usage.

Binary Item. A *binary item* uses the base 2 system to represent an integer in the range −32768 to 32767. It occupies one 16-bit word. The left-most bit of the reserved area is the operational sign. A binary item corresponds to COMPUTATIONAL-0 (COMP-0) usage.

Level Numbers

Subdividing Records

For purposes of processing, the contents of a file are divided into logical records. Level numbers allow you to specify subdivisions of a record necessary for referring to data. Once a subdivision is specified, it may be further subdivided to permit more detailed data reference. This is illustrated by Figure 8.1, in which the weekly timecard record is divided into four major items: name, employee-number, date, and hours, with more specific information appearing for name and date.

DATA DIVISION

Level Numbers

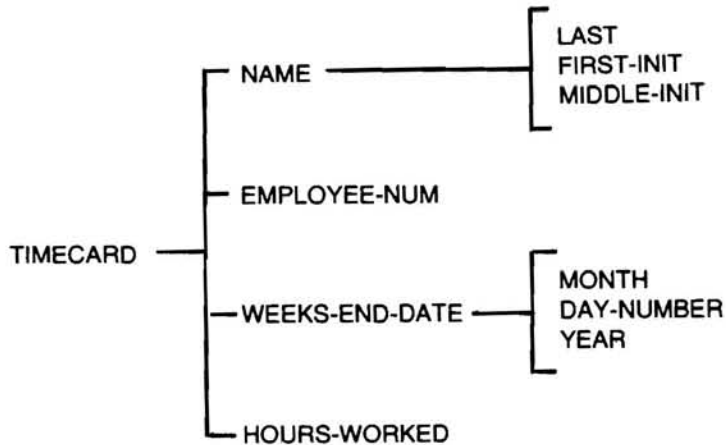


Figure 8.1 Example of Subdivided Records

Separate entries are written in the source program for each level. To illustrate the purpose of level numbers and their relationship to the contents of a record, the weekly timecard in the previous example may be described by DATA DIVISION entries having the level numbers, data-names, and PICTURE definitions shown in Figure 8.2.

IBM COBOL Coding Form

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

Figure 8.2. TIMECARD Data Definition

DATA DIVISION

Level Numbers

Data items that contain subdivisions (group items) in the example are: TIMECARD, NAME, and WEEKS-END-DATE. Subdivisions of a record that are not themselves further subdivided are called *elementary items*. The elementary items in the example are: LAST-NAME, FIRST-INIT, MIDDLE-INIT, EMPLOYEE-NUM, MONTH, DAY-NUMBER, YEAR, and HOURS-WORKED. All elementary items must include a PICTURE clause unless their USAGE IS INDEX.

Less inclusive groups are assigned numerically higher level numbers. Level numbers may be assigned with gaps to facilitate program maintenance. A group whose level is k includes all groups and elementary items described under it until a level number less than or equal to k is encountered.

A logical record is always initiated by a 01 level entry. (Level numbers less than 10 may be written as a single digit.) Additionally, a WORKING-STORAGE data item that is logically unrelated to other items may be coded as a stand-alone item with level number 77.

When a PROCEDURE DIVISION statement makes reference to a group item, the reference applies to the area reserved for the entire group.

In the FILE SECTION, consecutive logical records under a given file description (FD) constitute implicit *redefinitions* of the same physical area (see FILE and REDEFINES in Part III). In the WORKING-STORAGE SECTION, consecutive 01 levels define separate areas.

Data Description Entry

Components of the Entry

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

The *data-name* is a programmer-supplied word used to refer to the area being defined. Data-names must conform to the rules for word formation listed in Chapter 7, and, additionally, must begin with an alphabetic character. If an area is defined but not referred to in the program, the reserved word FILLER may be substituted for a data-name.

General Syntax

The general syntax of a data description entry is:

```
             { data-name }
level-number { FILLER    }  [REDEFINES clause]  [JUSTIFIED clause]

     [PICTURE clause]  [USAGE clause]  [SYNCHRONIZED clause]  [OCCURS clause]

     [BLANK clause]  [VALUE clause]  [SIGN clause].
```

When this syntax is applied to specific items of data, it is limited by the nature of the data being described. The format allowed for the description of each data type follows. 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 square brackets. The clauses may appear in any order except that a REDEFINES clause, if used, should come first. A detailed treatment of each clause can be found in Part III.

Group Item Syntax

```
level-number  data-name  [REDEFINES clause]  [USAGE clause]  [OCCURS clause]

     [SIGN clause].
```

Example:

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

NOTE: A USAGE clause coded at a group level may not be contradicted by an elementary level USAGE clause.
General Syntax

DATA DIVISION

Data Description Entry

Elementary Item Syntax

Alphanumeric or Alphanumeric-edited Item

```
             { data-name }
level-number { FILLER    }  [REDEFINES clause]    [OCCURS clause]

    PICTURE  IS an-form  [USAGE IS DISPLAY]  [JUSTIFIED clause]

    [VALUE IS non-numeric-literal]  [SYNCHRONIZED clause].
```

Examples:

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

NOTE: Inclusion of a VALUE clause will not cause editing to occur. For example, if MISC-2 included value "12345", the field would contain the characters 12345 followed by three spaces.

Numeric Item

```
             { data-name }
level-number { FILLER    }  [REDEFINES clause]  [OCCURS clause]

    PICTURE IS numeric-form  [SIGN clause]  [USAGE clause]

    [VALUE IS numeric-literal]  [SYNCHRONIZED clause].
```

Examples:

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

Numeric-edited Item

This is also called a report item.

```
             { data-name }
level-number { FILLER    }  [REDEFINES clause]  [OCCURS clause]

     PICTURE IS report-form  [BLANK WHEN ZERO]  [USAGE IS DISPLAY]

     [VALUE IS non-numeric literal]  [SYNCHRONIZED clause].
```

Example:

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

Literals and Figurative Constants

Literals and figurative constants are often used in place of data-names in COBOL programs. A *literal* is a constant that is not identified by a data-name in a program, but is completely defined by its own identity. A *figurative constant* is a special type of literal identified by a reserved word. Use of the reserved word causes the compiler to substitute the appropriate literal value.

Non-Numeric Literals

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

The following are examples of non-numeric literals:

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

Literals and Figurative Constants

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

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

Literals on Continuation Lines

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

1. A hyphen is placed in column 7 of the continuation line.

2. A delimiter is placed in Area B preceding the continuation of the literal.

3. All spaces at the end of the previous line and any spaces following the delimiter in the continuation line and preceding the final delimiter of the literal are considered to be part of the literal.

4. On any continuation line, Area A should be blank.

Numeric Literals

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

A decimal point may appear anywhere within the numeric literal, except as the 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 −6 −.333 0.5

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

Figurative Constants

A *figurative constant* is a special type of literal. It represents a value to which a reserved word has been assigned. A figurative constant is not bounded by quotation marks.

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

SPACE	the blank character represented by hexadecimal 20
LOW-VALUE	the character whose hexadecimal representation is 00
HIGH-VALUE	the character whose hexadecimal representation is 7F
QUOTE	the quotation mark, whose hexadecimal representation is 22
ALL literal	one or more instances of the literal, which must be a one-character non-numeric literal or another figurative constant (in which case ALL is redundant but serves for readability).

The plural forms of these figurative constants are acceptable to the compiler but are equivalent in effect. The plural of ZERO may be written as ZEROS or ZEROES. A figurative constant represents as many instances of the associated character as are required in the context of the statement.

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

DATA DIVISION

Size Limitations

Maximum Data Items per Section

There is a limit to the number of items that may be included in the Working-Storage, Linkage, and File sections of the Data Division. The result of

$$\frac{W + 4095}{4096} + F + L$$

must be less than or equal to 14, where W is the size of Working-Storage in bytes, F is the number of files described in the File Section, and L is the number of level 01 or 77 entries in the Linkage Section. Furthermore, the maximum number of files that may be open in the same run unit (main program linked together with an arbitrary number of subprograms) is 14.

CHAPTER 9 PROCEDURE DIVISION

Overview

The PROCEDURE DIVISION contains the executable instructions of a COBOL program. These instructions are expressed in statements similar to English. They employ the concept of verbs to denote actions, and statements and sentences to describe procedures. Examples of typical constructions for most reserved words can be found in the Application sections of Chapter 12, "Alphabetical Reserved Word List," in Part III.

Statements

Statement Types

A *statement* consists of a verb followed by appropriate operands (data-names or literals) and other words necessary to define a logical procedure. 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. Such a statement consists of any reserved verb and its operands, excluding SEARCH and any statement containing an IF, INVALID KEY, AT END, SIZE ERROR, or OVERFLOW clause.

Conditional Statements

A *conditional statement* directs program flow according to the result of a specified test condition. The IF and SEARCH statements provide this capability. Additionally, any I/O statement containing an INVALID KEY or AT END clause, any arithmetic statement with a SIZE ERROR clause, and any STRING or UNSTRING statement with an OVERFLOW clause is also considered a conditional.

PROCEDURE DIVISION

Statements

Statement Structures

Sentences

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

Paragraphs

A *paragraph* is a logical entity consisting of any number of sentences. Each paragraph must begin with a paragraph-name.

NOTE: Paragraph-names and section-names are procedure-names. Procedure-names must follow the rules for word-formation (see Chapter 7). In addition, a procedure-name may consist entirely of digits. An all-digit procedure-name may not consist of more than 18 digits. If it has leading zeros, they are all significant.

Sections

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

Design of the PROCEDURE DIVISION

Organization

The PROCEDURE DIVISION may be organized in three possible ways:

1. paragraphs only, or

2. one or more paragraphs followed by one or more sections (each section subdivided into one or more paragraphs), or
3. a DECLARATIVES portion and a series of sections (each section subdivided into one or more paragraphs).

Division Header

The PROCEDURE DIVISION header occurs in three forms, depending on how control is transferred to the program. If it is not called or chained from an external program (see CALL and CHAIN in Part III), or it is chained but without parameter passing, the format is standard:

```
PROCEDURE DIVISION.
```

If the program is operated by a CHAIN statement in an external program and data items will be passed to it, the syntax is:

```
PROCEDURE DIVISION CHAINING data-name-1... .
```

If the program is operated by a CALL statement in an external program, the syntax is:

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

An external program operated by a CALL must be linked with the main program, even though it is compiled separately. Include the optional USING list only if data items will be passed to the subprogram.

Passing Data Items

Data items are passed according to the corresponding position between the list in the PROCEDURE DIVISION header and the list in the associated CALL or CHAIN statement. This eliminates any requirement to use identical data-names. However, the number of entries in corresponding lists must be identical, and corresponding data-names must reference areas of the same size and USAGE. Failure to do so will not be diagnosed and will cause unpredictable results at runtime.

Design of the PROCEDURE DIVISION

A called subprogram accesses the same data storage area used by the main program. Therefore, any values altered by the subprogram will remain in effect after control is returned to the main program. Since no return is possible from a chained program, data items can be passed in one direction only.

For a thorough understanding of the CALL and CHAIN processes, read Appendix A, "Interprogram Communication." Also see the entries for CALL and CHAIN in Part III.

Declaratives and I/O Error Handling

Inclusion of the DECLARATIVES portion is optional and provides a means of creating programmer-defined error handling procedures (see DECLARATIVES in Part III). If you include it, you may use only design #3, described previously.

Hierarchy of COBOL-86 Error Checks

When an I/O error occurs, the following events take place:

1. The appropriate FILE STATUS item, if one exists, is set to the proper two-character code (see INPUT-OUTPUT in Part III).

2. a. If the error occurs on execution of a statement containing an AT END or INVALID KEY clause, the associated imperative statement(s) is executed. Otherwise,

   b. if you have specified a DECLARATIVES procedure, the error handling logic included in it is executed. Control then passes to the sentence following the one that produced the error. Otherwise,

   c. if a DECLARATIVES procedure does not exist but a FILE STATUS item does exist, program flow continues normally on the assumption that the STATUS item will be tested to determine a course of action. Otherwise,

   d. the runtime system receives control and program execution is aborted.

This process applies to the processing of any file, regardless of its ORGANIZATION type.

Design of the PROCEDURE DIVISION

Segmentation

The program segmentation facility is provided to enable the execution of COBOL-86 programs that are larger than physical memory. When segmentation is used (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].
```

Range for Segment-Number

Segment-number must be an integer with a value in the range from 0 through 99. If you do not include a segment-number, it is assumed to be 0. DECLARATIVES must have a segment-number less than 50. All sections that have the same segment number constitute a single program segment and must occur contiguously in the source program.

Furthermore, all segments with numbers less than 50 must occur at the beginning of the PROCEDURE DIVISION, after the DECLARATIVES (if present), but before any segment with a segment-number greater than 49.

Fixed and Independent Segments

Segments with numbers 0 through 49 are called *fixed segments* and are always resident in memory during execution. Segments with numbers greater than 49 are called *independent segments*. Each independent segment is treated as a program overlay.

An independent segment is in its initial state when control is passed to it from a segment with a different segment-number.

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

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

2. A PERFORM statement in a fixed segment may have within its range only
   a. sections and/or paragraphs wholly contained within fixed segments, or
   b. sections and/or paragraphs wholly contained in a single independent segment.

Design of the PROCEDURE DIVISION

3. A PERFORM statement in an independent segment may have within its range only

   a. sections and/or paragraphs wholly contained within fixed segments, or
   b. sections and/or paragraphs wholly contained within the same independent segment as the PERFORM statement.

CHAPTER 10 INDEXED FILES

Overview

An *indexed-file* is one that permits random access of records by maintaining a directory of the unique key value embedded in the contents of each record. Indexed organization is limited to disk files.

You can access an indexed file in one of three ways: sequentially, dynamically, or randomly.

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

In the random access mode, you control the order of access to records. Each record desired is accessed by placing the value of its key in the RECORD KEY prior to an access statement.

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

Syntax in ENVIRONMENT DIVISION

SELECT Clause

In the ENVIRONMENT DIVISION, the SELECT entry must specify ORGANIZATION IS INDEXED, and the ACCESS clause syntax is:

```
ACCESS MODE IS SEQUENTIAL | RANDOM | DYNAMIC.
```

ASSIGN, RESERVE, and FILE STATUS clause formats are identical to those specified in Chapter 12, the "Alphabetical Reserved Word List," in Part III. An indexed file must be assigned to disk. In the FD entry for an INDEXED file, both LABEL RECORDS STANDARD and a VALUE OF FILE-ID clause must appear, as for any file assigned to disk.

RECORD KEY Clause

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

```
RECORD KEY IS data-name
```

INDEXED FILES

Syntax in ENVIRONMENT DIVISION

where *data-name* is an item defined within the record descriptions of the associated file description, and 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 the data-name 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 organization file, the designated two-character data item is set after every I-O statement. Table 10.1 summarizes the possible settings.

Error Reporting

File status 21 arises if ACCESS MODE IS SEQUENTIAL and the RECORD KEY is altered between execution of two WRITE statements or prior to 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 executed. 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 occurred with Permanent Error (3) for sequential files.

If an error occurs at execution time and no AT END or INVALID KEY statements are given *and* no appropriate DECLARATIVES procedure is supplied *and* no FILE STATUS is specified, the error will be displayed on the screen and the program will terminate. See Chapter 9.

Syntax in ENVIRONMENT DIVISION

Table 10.1. File Status Values

STATUS DATA ITEM LEFT CHARACTER	STATUS DATA ITEM RIGHT CHARACTER NO FURTHER DESCRIPTION (0)	SEQUENCE ERROR (1)	DUPLICATE KEY (2)	NO RECORD FOUND (3)	DISK SPACE FULL (4)
Successful Completion (0)	X				
At End (1)	X				
Invalid Key (2)		X	X	X	X
Permanent Error (3)	X				
Special Cases (9)		X			

Syntax in PROCEDURE DIVISION

OPEN Statement

The syntax of the sequential file OPEN statement (see OPEN in Part III) also applies to Indexed files, except EXTEND is inapplicable.

Table 10.2 summarizes the available statement types and their permissibility in terms of ACCESS MODE and OPEN option in effect. Where X appears, the statement is permissible; otherwise it is not valid.

CLOSE Statement

In addition to the statements listed in Table 10.2, CLOSE is permissible under all conditions; the same format as shown for CLOSE in Part III is used.

INDEXED FILES

Syntax in PROCEDURE DIVISION

Table 10.2. Procedure Statement Options

ACCESS MODE	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

READ Statement

Format 1 (sequential or dynamic 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...]
```

Syntax in PROCEDURE DIVISION

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 dynamic access mode file. The AT END clause is executed when the logical end-of-file condition arises. If this clause is not written in the source statement, an appropriate DECLARATIVES procedure is given control, if available.

Format 2 is used for files in random access mode or for files in dynamic access mode when records are to be retrieved randomly.

In Format 2, the INVALID KEY clause specifies action to be taken if the access key value does not refer to an existing key in the file. If the clause is not given, the appropriate DECLARATIVES procedure, if supplied, is given control.

The optional KEY IS clause must designate the record key item declared in the file's SELECT statement. This clause serves as documentation only. You must ensure that a valid key value is in the designated key field prior to execution of a random access READ.

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

WRITE Statement

The WRITE statement releases a logical record to a file opened for OUTPUT or I-O. Its general syntax is:

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

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

INDEXED FILES

Syntax in PROCEDURE DIVISION

In the event of an improper key value, the imperative statements are executed if the INVALID KEY clause appears in the statement; otherwise an appropriate DECLARATIVES procedure is executed, if available. The INVALID KEY condition arises if:

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

REWRITE Statement

The REWRITE statement logically replaces an existing record in a file opened for I-O; the syntax of the statement is:

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

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

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

DELETE Statement

The DELETE statement logically removes a record from an indexed file. The general syntax of the statement is:

```
DELETE filename RECORD [INVALID KEY imperative-statement...]
```

Syntax in PROCEDURE DIVISION

For a file in the sequential access mode, the last input-output statement executed for filename would have been a successful READ statement. The record that was read is deleted. Consequently, you should not specify an INVALID KEY phrase for sequential access mode files.

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

START Statement

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. The syntax of this statement is:

```
                    ┌         ┌ GREATER THAN  ┐            ┐
START filename      │ KEY IS  │ NOT LESS THAN │  data-name │ [INVALID KEY imperative statement...]
                    └         └ EQUAL TO      ┘            ┘
```

Data-name must be the declared RECORD KEY and the value to be matched by a record in the file must be present in the data-name. When you execute this statement, the file must be OPEN for INPUT or I-O.

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

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

CHAPTER 11 RELATIVE FILES

Overview

Records in a relative file are differentiated on the basis of a relative record number that ranges from 1 to 32,767, or to a lesser maximum for a smaller file. Unlike 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. Relative organization is restricted to disk files.

A relative 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 intermix random and sequential access at will.

Syntax in ENVIRONMENT DIVISION

SELECT Clause

In the ENVIRONMENT DIVISION, the SELECT entry must specify ORGANIZATION IS RELATIVE, and the ACCESS clause syntax is:

```
ACCESS MODE IS SEQUENTIAL | RANDOM | DYNAMIC.
```

ASSIGN, RESERVE, and FILE STATUS clause formats are identical to those used for sequential or indexed files.

In the associated FD entry, you must declare LABEL RECORDS STANDARD, and you must include a VALUE OF FILE-ID clause.

The first byte of the record area associated with a relative file should not be described as part of a COMP-0 or COMP-3 item by any record description for the file.

RELATIVE FILES

Syntax in ENVIRONMENT DIVISION

RELATIVE KEY Clause

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

```
RELATIVE KEY IS data-name
```

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

Syntax in PROCEDURE DIVISION

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, Tables 10.1 and 10.2 also apply to RELATIVE files.) The syntax for OPEN and CLOSE (see Part III) are applicable to Relative files, except for the EXTEND phrase.

READ Statement

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...]
```

Syntax in PROCEDURE DIVISION

Format 1 must be used for all files in sequential access mode. The NEXT phrase must be present to achieve sequential access if the file's declared mode of access is dynamic. The AT END clause, if given, is executed when the logical end-of-file condition exists, or, if not given, the appropriate DECLARATIVES procedure is given control, if available.

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

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

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

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

WRITE Statement

The syntax of the WRITE statement is the same for a relative file as for an indexed file:

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

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

If access mode is random or dynamic, then 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.

RELATIVE FILES

Syntax in PROCEDURE DIVISION

REWRITE Statement

The syntax of the REWRITE statement is the same for a relative file as for an indexed file:

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

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

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

DELETE Statement

The syntax of the DELETE statement is the same for a relative file as for an indexed file:

```
DELETE filename RECORD [INVALID KEY imperative statement...]
```

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

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

Syntax in PROCEDURE DIVISION

START Statement

The syntax of the START statement is the same for a relative file as for an indexed file:

```
                [          { GREATER THAN  }              ]
START filename  [ KEY IS   { NOT LESS THAN } data-name-1  ] [INVALID KEY imperative statement...]
                [          { EQUAL TO      }              ]
```

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

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

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

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

Part III
COBOL-86 Reserved Words

COBOL-86 RESERVED WORDS

CHAPTER 12 ALPHABETICAL RESERVED WORD LIST

Introduction

Parts II and III of this manual comprise a detailed reference guide to the COBOL language as set forth in the 1974 ANSI definition and implemented in the COBOL-86 Compiler. A full description of this system's compliance with the 1974 standard is provided at the beginning of Chapter 7. As in Part II, an effort has been made to simplify your work by combining the treatment of ANSI standard information with the extensions and variations pertinent to COBOL-86. If you have questions about the syntax diagrams, refer to Syntax Notation in the "Introduction."

An understanding of the organization of this part is essential to your successful use of it. As much as possible, the syntax, details, and application of each COBOL reserved word have been treated on an individual word basis. Organization of this part is alphabetical by reserved word. Essential information about the language that is inappropriate for such a format can be found in Part II, "Language Overview."

The following organizational guidelines apply to the use of Part III:

1. Primary reserved words are those that define the function of an entire statement, e.g., ADD, COMPUTE, and SEARCH. These words are the principal entries in the listing and are arranged in *alphabetical order*. They are shown in Table 12.1 in **boldface** type.

2. Secondary reserved words only occur in conjunction with primary words. In general, they serve to elaborate the syntax of the primary word and cannot themselves define the function of the statement, e.g., ROUNDED, VARYING, and DELIMITER. These words are shown in Table 12.1 with cross-references that refer you to the primary word in whose syntax they appear.

   (A few words fall into categories 1 and 2. These are shown in boldface, but also include cross-references.)

3. Optional reserved words are not shown in the directory, e.g., ON, AT, and ARE. They are, however, included in syntax diagrams wherever they are legal and are also included liberally in the Application examples.

ALPHABETICAL RESERVED WORD LIST

Introduction

(A few words fall into categories 2 and 3. These are shown in the directory only in their required uses.)

Appendix H lists additional words that are reserved in the ANSI 1974 standard, but are not used in COBOL-86.

Table 12.1. COBOL-86 Reserved Words

RESERVED WORD	CROSS-REFERENCES
ACCEPT	
ACCESS	see INPUT-OUTPUT
ADD	
AFTER	see DECLARATIVES, INSPECT, PERFORM, WRITE
ALL	see INSPECT, SEARCH, UNSTRING, also Part II: "FIGURATIVE CONSTANTS"
ALPHABETIC	see IF
ALTER	
AND	see IF
AREA (Input/Output)	
ASCENDING	see OCCURS, SEARCH
ASCII	see CODE-SET, CONFIGURATION
ASSIGN	see INPUT-OUTPUT
AUTHOR	see IDENTIFICATION
AUTO	see SCREEN
AUTO-SKIP	see ACCEPT
BACKGROUND-COLOR	see SCREEN
BEEP	see ACCEPT
BEFORE	see INSPECT, WRITE
BELL	see SCREEN
BLANK	see SCREEN
BLOCK	
BOTTOM	see LINAGE
BY	see DIVIDE, INSPECT, MULTIPLY, PERFORM, SET
CALL	
CHAIN	
CHAINING	see PROCEDURE
CHARACTERS	see CONFIGURATION, INSPECT, BLOCK, RECORD
CLOSE	
CODE-SET	
COL	see ACCEPT, DISPLAY, EXHIBIT
COLUMN	see SCREEN
COMMA	see CONFIGURATION
COMPUTATIONAL	see USAGE
COMPUTATIONAL-3	see USAGE

ALPHABETICAL RESERVED WORD LIST

Introduction

Table 12.1 (continued). COBOL-86 Reserved Words

RESERVED WORD	CROSS-REFERENCES
COMPUTE	
CONFIGURATION	see ENVIRONMENT
CONTAINS	see BLOCK, RECORD
COPY	
COUNT	see IF, UNSTRING
CURRENCY	see CONFIGURATION
DATA (DIVISION header)	
DATA (RECORD clause)	
DATE	see ACCEPT
DATE-COMPILED	see IDENTIFICATION
DATE-WRITTEN	see IDENTIFICATION
DAY	see ACCEPT
DEBUGGING	see CONFIGURATION
DECIMAL-POINT	see CONFIGURATION
DECLARATIVES	see PROCEDURE
DELETE	see Part II: Chapter 10-11
DELIMITED	see STRING, UNSTRING
DELIMITER	see UNSTRING
DEPENDING	see GO
DESCENDING	see OCCURS, SEARCH
DISK	see INPUT-OUTPUT
DISPLAY	see USAGE
DIVIDE	
DIVISION	see DATA (division), ENVIRONMENT, IDENTIFICATION, PROCEDURE
DOWN	see SET
DYNAMIC	see INPUT-OUTPUT
ELSE	see IF
END	see DECLARATIVES, READ, SEARCH, also Part II: Chapter 10-11
END-OF-PAGE	see WRITE
ENVIRONMENT	
EOP	see WRITE
EQUAL	see IF, also Part II: Chapter 10
ERASE	see DISPLAY, EXHIBIT
ERROR	see ADD, COMPUTE, DECLARATIVES, DIVIDE, MULTIPLY, SUBTRACT
ESCAPE	see ACCEPT
EXCEPTION	see DECLARATIVES
EXHIBIT	
EXIT	
EXIT PROGRAM	
EXTEND	see DECLARATIVES, OPEN
FD	see FILE
FILE	see DATA (division)
FILE-CONTROL	see INPUT-OUTPUT

ALPHABETICAL RESERVED WORD LIST

Introduction

Table 12.1 (continued). COBOL-86 Reserved Words

RESERVED WORD	CROSS-REFERENCES
FILE-ID	see VALUE (OF FILE-ID)
FILLER	see Part II: "DATA DESCRIPTION ENTRY"
FIRST	see INSPECT
FOOTING	see LINAGE
FOR	see INSPECT
FOREGROUND-COLOR	see SCREEN
FROM	see ACCEPT, PERFORM, REWRITE, SCREEN, SUBTRACT, WRITE, also Part II: Chapter 10-11
GIVING	see ADD, SUBTRACT, MULTIPLY, DIVIDE
GO	
GREATER	see IF, also Part II: Chapter 10
HIGH-VALUE(S)	see Part II: "FIGURATIVE CONSTANTS"
I-O	see DECLARATIVES, OPEN
I-O-CONTROL	see INPUT-OUTPUT
IDENTIFICATION	
IF	
IN	
INDEX	see USAGE
INDEXED	see INPUT-OUTPUT, OCCURS
INITIAL	
INPUT	see DECLARATIVES, OPEN
INPUT-OUTPUT	see ENVIRONMENT
INSPECT	
INSTALLATION	see IDENTIFICATION
INTO	see DIVIDE, READ, STRING, UNSTRING, also Part II: Chapter 10-11
INVALID	see Part II: Chapter 10-11
IS	see CONFIGURATION
JUST	see SCREEN
JUSTIFIED	see SCREEN
KEY	see ACCEPT, also Part II: Chapter 10, also OCCURS
LABEL	see FILE
LEADING	see INSPECT, SIGN
LEFT-JUSTIFY	see ACCEPT
LENGTH-CHECK	see ACCEPT
LESS	see IF, also Part II: Chapter 10
LIN	see ACCEPT, DISPLAY, EXHIBIT
LINAGE	see DATA (division)
LINAGE-COUNTER	see LINAGE
LINE	see ACCEPT, INPUT-OUTPUT, SCREEN, WRITE
LINES	see WRITE, also LINAGE
LINKAGE	see DATA (division)
LOCK	see CLOSE
LOW-VALUE(S)	see Part II: "FIGURATIVE CONSTANTS"
MEMORY	see CONFIGURATION

Table 12.1 (continued). COBOL-86 Reserved Words

RESERVED WORD	CROSS-REFERENCES
MODE	see CONFIGURATION MODULES see CONFIGURATION
MOVE	
MULTIPLY	
NATIVE	see CONFIGURATION
NEGATIVE	see IF
NEXT	see IF, SEARCH, also Part II: Chapter 10-11
NOT	see IF, also Part II: Chapter 10
NUMBER	see ACCEPT, SCREEN
NUMERIC	see IF
OBJECT-COMPUTER	see CONFIGURATION
OCCURS	
OF	see IF (synonym)
OMITTED	see FILE
OPEN	
OR	see UNSTRING
ORGANIZATION	see INPUT-OUTPUT
OUTPUT	see DECLARATIVES, OPEN
OVERFLOW	see STRING, UNSTRING
PAGE	see WRITE
PERFORM	
PIC	see SCREEN
PICTURE	see SCREEN
PLUS	see SCREEN
POINTER	see STRING, UNSTRING
POSITIVE	see IF
PRINTER	see CONFIGURATION, INPUT-OUTPUT
PROCEDURE	see DECLARATIVES
PROGRAM	see EXIT (PROGRAM)
PROGRAM-ID	see IDENTIFICATION
PROMPT	see ACCEPT
QUOTE(S)	see Part II: "FIGURATIVE CONSTANTS"
RANDOM	see INPUT-OUTPUT
READ	see Part II: Chapter 10-11
READY	see TRACE
RECORD	see DATA (RECORD clause), FILE, INPUT-OUTPUT, READ
RECORDS	see BLOCK, DATA (RECORD clause), FILE
REDEFINES	
RELATIVE	see INPUT-OUTPUT
REPLACING	see INSPECT
RESERVE	see INPUT-OUTPUT
RESET	see TRACE
REVERSE-VIDEO	see SCREEN
REWRITE	see Part II: Chapter 10-11
RIGHT-JUSTIFY	see ACCEPT

ALPHABETICAL RESERVED WORD LIST

Introduction

Table 12.1 (continued). COBOL-86 Reserved Words

RESERVED WORD	CROSS-REFERENCES
ROUNDED	see ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT
RUN	see STOP
SAME	see INPUT-OUTPUT
SCREEN	see DATA (division)
SEARCH	
SECTION	see CONFIGURATION, DATA (division), DECLARATIVES, ENVIRONMENT, FILE, INPUT-OUTPUT, LINKAGE, PROCEDURE, SCREEN, WORKING-STORAGE
SECURE	see SCREEN
SECURITY	see IDENTIFICATION
SELECT	see INPUT-OUTPUT
SENTENCE	see IF, SEARCH
SEPARATE	see SIGN
SEQUENCE	see CONFIGURATION
SEQUENTIAL	see INPUT-OUTPUT
SET	
SIGN	
SIZE	see ADD, COMPUTE, DIVIDE, MULTIPLY, STRING, SUBTRACT
SORT	see INPUT-OUTPUT
SORT-MERGE	see INPUT-OUTPUT
SOURCE-COMPUTER	see CONFIGURATION
SPACE(S)	see PART II: "FIGURATIVE CONSTANTS"
SPACE-FILL	see ACCEPT
SPECIAL-NAMES	see CONFIGURATION
STANDARD	see FILE
STANDARD-1	see CONFIGURATION
START	see Part II: Chapter 10
STATUS	see INPUT-OUTPUT
STOP	
STRING	
SUBTRACT	
SYNC	
SYNCHRONIZED	
TALLYING	see INSPECT, UNSTRING
THROUGH	see PERFORM, VALUE (in condition-names)
THRU	see PERFORM, VALUE (in condition-names)
TIME	see ACCEPT
TIMES	see PERFORM
TO	see ADD, ALTER, MOVE, RECORD, SCREEN, SET
TOP	see LINAGE
TRACE	
TRAILING	see SIGN
TRAILING-SIGN	see ACCEPT
UNSTRING	
UNTIL	see PERFORM

Table 12.1 (continued). COBOL-86 Reserved Words

RESERVED WORD	CROSS-REFERENCES
UP	see SET UPDATE see ACCEPT
USAGE	
USE	see DECLARATIVES
USING	see CALL, CHAIN, PROCEDURE, SCREEN
VALUE (OF FILE-ID)	
VALUE (to initialize a)	see SCREEN
VALUE (in condition-names)	
VALUES	see VALUE (in condition-names)
VARYING	see PERFORM, SEARCH
WHEN	see SEARCH
WITH	see ACCEPT, also CLOSE, CONFIGURATION
WORDS	see CONFIGURATION
WORKING-STORAGE	see DATA (division)
WRITE	see Part II: Chapter 10-11
ZERO (ZEROS, ZEROES)	see BLANK, IF, SCREEN, also Part II: "FIGURATIVE CONSTANTS"
ZERO-FILL	see ACCEPT

ALPHABETICAL RESERVED WORD LIST

ACCEPT

Syntax in PROCEDURE DIVISION

Format 1:

```
                                 { DATE         }
ACCEPT identifier-1 FROM         { DAY          }
                                 { TIME         }
                                 { LINE NUMBER  }
                                 { ESCAPE KEY   }
```

Format 2:

```
ACCEPT identifier-2
```

Format 3:

```
                                              { SPACE-FILL     }
                                              { ZERO-FILL      }
                                              { LEFT-JUSTIFY   }
                                              { RIGHT-JUSTIFY  }
ACCEPT position-spec identifier-3 [WITH       { TRAILING-SIGN  } ...]
                                              { PROMPT         }
                                              { UPDATE         }
                                              { LENGTH-CHECK   }
                                              { AUTO-SKIP      }
                                              { BEEP           }
```

Format 4:

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

The function of the ACCEPT statement is to acquire data from a source external to the program and place them in a specified receiving field or set of receiving fields.

Details

The formats differ primarily in the data source with which they are designed to interface. The Format 1 ACCEPT is used primarily to obtain date or time information from the operating system clock. Formats 2 and 3 receive data that you key in. For Format 2, this device is assumed to be a teletype, a glass teletype, or a CRT terminal in scrolling mode. For Format 3, it is assumed that the input device is a video terminal and that scrolling is not desired. The Format 4 ACCEPT receives an entire data entry form (as defined in the SCREEN SECTION) when it has been completed by the terminal operator. Note that an ordinary terminal is suitable as an input device for a Format 2, 3, or 4 ACCEPT, although the appearance of the screen will differ as indicated in the following discussion.

Format 1 ACCEPT Statement

Use

Format 1 is used to obtain any of several standard values at execution time.

Value Formats

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.

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

LINE NUMBER—The `ACCEPT...FROM LINE NUMBER` statement is provided for compatibility, but in the COBOL-86 system, the value of LINE NUMBER is always zero.

ACCEPT

ESCAPE KEY—a two-digit code generated by the key that terminated the most recently executed Format 3 or Format 4 ACCEPT statement. Identifier-1 can be interrogated to determine exactly which key was typed. You may terminate input using any of the following keys, and cause the ESCAPE KEY value to be set as shown:

Backtab (terminates only format 3 ACCEPTs)	99
Escape	01
Field-terminator (of the last field if format 4 ACCEPT is used)	00
Function key	02-nn

Key Designations

On Zenith Z-100 series computers, a backtab is entered as CTRL-B, and an escape is entered with the ESC key. Field-terminator keys are RETURN, LINE FEED, and TAB.

Function keys are CTRL-D, -E, -F, and -G. If input is terminated by the AUTO-SKIP function (i.e., 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 it does not, the standard rules for a MOVE govern truncation and zero-fill of the source value.

Format 2 ACCEPT Statement

Uses

Format 2 of the ACCEPT statement is used to accept a string of input characters from a scrolling device such as a teletype or a CRT in scrolling mode. When the ACCEPT statement is executed, input characters are read from the terminal until a RETURN is encountered, then a carriage return/line feed pair is sent back to the terminal. The input data string is considered to consist of all characters keyed prior to (but not including) the RETURN.

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 a length equal to the number of characters in the string. (That is, left justifying, space filling, and right truncating occur by default, and right justifying and left truncating occur if the receiving field 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 will occur.

For a Format 2 ACCEPT with a numeric or numeric-edited receiving field, the input data string is subjected to a validity test that depends on the PICTURE of the receiving field. The digits 0 through 9 are considered valid anywhere in the input data string.

Character Validity Rules

The decimal point character (period or comma, depending on the DECIMAL POINT IS clause of the CONFIGURATION SECTION) is considered valid if:

1. it occurs only once in the input data string, and
2. if the PICTURE of the receiving field contains a fractional digit position, that is, a 9, Z, *, or floating insertion character that 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 message `INVALID NUMERIC INPUT--PLEASE RETYPE` is displayed on the screen, and another input data string is read.

When a valid numeric data string has been obtained, data are transferred to the receiving field exactly as if the instruction being executed were a MOVE to the receiving field from a hypothetical source field with the following characteristics:

1. a PICTURE of the form S9...9V9...9
2. USAGE DISPLAY
3. a total length equal to the number of digits in the input data string

4. 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)

5. current contents equal to the string of digits embedded in the input data string

6. a separate sign with a current negative status if the input data string contains the – character (hyphen), and a current positive status otherwise.

Format 3 ACCEPT Statement

Uses

Format 3 of the ACCEPT statement is used to accept data into a field from a nonscrolling video terminal. The following syntax rules must be observed when the Format 3 ACCEPT is used.

Syntax Rules

1. Identifier-3 must reference a data item whose length is less than or equal to 1920 characters.

2. The options SPACE-FILL and ZERO-FILL may not both be specified in the same ACCEPT statement.

3. The options LEFT-JUSTIFY and RIGHT-JUSTIFY may not both be specified within the same ACCEPT statement.

4. If identifier-3 is described as a numeric-edited item, the UPDATE option must not be specified.

5. 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.

6. For 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.

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

Characteristics of the 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 screen of the terminal.

The position-spec is expressed in either of the following forms:

```
(LIN[ {+ | -} integer-1], COL[ {+ | -} integer-2])

(integer-3, integer-4)
```

The opening and closing parentheses and the comma are required. The position-spec specifies the position on the screen where the data input field will begin. LIN and COL are COBOL special registers. Each behaves like a numeric data item with USAGE COMP, but they may be referenced without being declared in the DATA DIVISION.

If you specify LIN, the data input field will begin in the screen row whose number is equal to the value of the LIN special register. If + integer-1 or − integer-1 is included, the effective value of LIN will be increased or decreased accordingly, without affecting the value in the LIN register. If you specify integer-3, the data input field will begin on the row whose number is integer-3. If you specify neither LIN nor integer-3, the data input field will begin on the screen row containing the current cursor position.

If you specify COL, the data input field will begin in the screen column whose number is equal to the value of the COL special register. If + integer-2 or − integer-2 is included, the effective value of COL will be increased or decreased accordingly, without affecting the value in the COL register. If you specify integer-4, the data input field will begin in the column whose number is integer-4. If neither COL nor integer-4 is specified, the data input field will begin in the screen column containing the current cursor position.

ALPHABETICAL RESERVED WORD LIST

ACCEPT

The characteristics (other than screen position) of the data input field are determined by the receiving field's PICTURE specification. If identifier-3 is defined by an alphanumeric or alphanumeric-edited PICTURE clause, the data input field is simply a string of characters starting at the screen location specified by position-spec. The length of the data input field in character positions is equal to the length of the receiving field in memory.

If identifier-3 if defined by a numeric or numeric-edited PICTURE clause, the data input field may contain any or all of the following: integer digit positions, fractional digit positions, sign position, and/or decimal point position. There will be one digit position for each 9, Z, *, or P, noninitial floating insertion symbol in the PICTURE of identifier-3. (A floating insertion symbol may be a +, −, or $.) See PICTURE.

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 will be one sign position if identifier-3 is defined as signed, and no sign position otherwise. There will be 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 will 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 will be in the following sequence: integer digit positions (if any), decimal point position (if any), fractional digit positions (if any), sign position (if any). If TRAILING-SIGN is not specified, the data input positions will be in the following sequence: sign position (if any), integer digit positions (if any), decimal point position (if any), fractional digit positions (if any).

Each character entered from the terminal is treated either as a terminator character, a data character, or an editing character.

Terminator Characters

The terminator keys are RETURN, LINE FEED, and TAB. When you press a terminator key, the ACCEPT is terminated and the ESCAPE KEY value is set as described in Format 1 ACCEPT Statement. This value can be interrogated by using a Format `1 ACCEPT identifier FROM ESCAPE KEY` statement.

Data Character with Alphanumeric Receiving Field

An alphanumeric-edited receiving field is treated as an alphanumeric field of the same length (as if every character in its PICTURE were X). Insertion editing does not occur.

The initial appearance of the data input field depends on the specifications in the WITH phrase of the ACCEPT statement. If you specify UPDATE, the current contents of identifier-3 are displayed in the input field. In this case all data input positions will be treated as if they were keyed from the terminal. If you do not specify UPDATE, but PROMPT is specified, a period (.) is displayed in each input data position. If you specify neither UPDATE nor PROMPT, the data input field is not changed.

The cursor is placed in the first data input position, and characters are accepted as you key them until a terminator character (normally RETURN) is encountered. If you specify AUTO-SKIP in the ACCEPT statement, the ACCEPT will also be terminated if you key a character into the last (right-most) data input position. If you specify LENGTH-CHECK, terminator characters are ignored until every data input position is filled.

As each input character is received, it is echoed to the screen, except that nondisplayable characters are echoed as ? (question marks). If all positions of the data input field are filled, additional input is ignored until a terminator character or editing character is encountered. If RIGHT-JUSTIFY was specified in the ACCEPT statement, the operator-keyed characters are shifted to the right-most positions of the data input field when the ACCEPT is terminated. All unkeyed character positions are filled on termination; the fill character is either space (which is the default) or zero (if ZERO-FILL was specified).

The contents of the receiving field will be the same set of characters as appear in the input field; however, the justification will be controlled by the JUSTIFIED specification that is operative in the receiving field's data description, not by the RIGHT- or LEFT-JUSTIFY option of the ACCEPT. Excess positions of the receiving field will be filled with spaces unless the ZERO-FILL specification is included in the ACCEPT statement.

ALPHABETICAL RESERVED WORD LIST

ACCEPT

Data Character with Numeric Receiving Field

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 (which is not permitted for a numeric-edited receiving field), the integer and fractional parts of the data input field will be 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 data. If you do not specify UPDATE, but PROMPT is specified, a zero will be displayed in each input digit position. In either of these cases (UPDATE or PROMPT) a decimal point will be displayed at the decimal point position.

If you specify neither UPDATE nor PROMPT, the input field on the screen will not be initialized, except for the sign position. The sign position is always initialized positive unless UPDATE is specified, when 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 you key them. A received character may be treated in one of several ways. If the incoming character is a digit, previously keyed digits are shifted one position to the left in the input field and the new digit is displayed 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 has been filled and AUTO-SKIP was specified, the integer part is terminated and the cursor is moved to the left-most fractional digit position.

If the integer part has been filled and AUTO-SKIP was not specified, the cursor is moved to the decimal point position, and any further digits keyed are ignored until the integer part is terminated with a decimal point.

If the character entered is a sign character (+ 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 terminated and the cursor is moved to the left-most fractional digit position.

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

When the integer part is terminated, the cursor is placed in the left-most fractional digit position, and operator-keyed characters are again accepted. Digits are simply echoed to the terminal. The sign characters (+ and −) are treated exactly as they were while integer digits were being entered. The field-terminator character terminates the ACCEPT. (If AUTO-SKIP is in effect, filling the entire fractional part also terminates 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 digits may be terminated only by terminating 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 a valid numeric data input field has been obtained, data are transferred to the receiving field exactly as if the instruction being executed were a MOVE to the receiving field from a hypothetical source field with the following characteristics:

1. a PICTURE of the form S9...9V9...9

2. USAGE DISPLAY

3. a total length equal to the number of digits in the data input field

4. 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 data input field (zero if there is no decimal point in the data input field)

ACCEPT

5. current contents equal to the string of digits embedded in the data input field

6. a separate sign with a current negative status if the data input field contains the – character (hyphen), and a current positive status otherwise.

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

Editing Characters

The editing characters are line-delete (CTRL-X), forward-space (CTRL-L), BACK SPACE (or CTRL-H), and DELETE. These characters may be used to change data that have already been keyed (or supplied as a result of a WITH UPDATE specification).

Entering the line-delete character will cause the ACCEPT to restart and all data keyed by the operator or initially present in the receiving field to be lost. The data input field on the console screen will be reinitialized if PROMPT is in effect. Otherwise, the data input field will be filled with spaces or zeros according to the SPACE-FILL or ZERO-FILL specification.

Typing the forward-space or BACK SPACE characters will move the cursor forward or back one data input position in the case of an alphanumeric or alphanumeric-edited receiving field, or one digit position in the case of a numeric or numeric-edited receiving field. In no case, however, will the forward-space or BACK SPACE characters move the cursor outside the range of positions, including (1) the positions already keyed (or filled by specifying WITH UPDATE) and (2) the right-most data input position that the cursor has occupied during the execution of this ACCEPT. If the cursor is moved to a position within 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 (for alphanumeric or alphanumeric-edited fields) or one digit position (for numeric or numeric-edited fields).

Entering DELETE cancels the last data character or digit entered. The cursor is moved back one position and a fill character (space or zero) is displayed above the cursor, except when the cursor is to the left of the decimal point for a numeric ACCEPT. In this case 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 it are shifted one position to the right. The DELETE key has no effect if the cursor has previously been moved from the position where the next character (or digit) would normally be entered.

WITH Phrase Summary

For alphanumeric receiving fields:

1. SPACE-FILL causes unkeyed character positions of the data input field and the receiving field to be space-filled when the ACCEPT is terminated.

2. ZERO-FILL causes unkeyed character positions of the data input field and the receiving field to be set to ASCII zeros when the ACCEPT is terminated.

3. LEFT-JUSTIFY is treated by this compiler as commentary.

4. RIGHT-JUSTIFY causes operator-keyed characters to occupy the right-most positions of the data input field after the ACCEPT is terminated. The justification of 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.

5. PROMPT causes the data input field to be set to all periods (.) before input characters are accepted.

6. UPDATE causes the data input field to be initialized with the current contents of the receiving field. The data displayed by UPDATE are treated as if they were operator-keyed data. UPDATE supercedes the action of PROMPT if both are specified.

7. LENGTH-CHECK causes a field-terminator character to ignored unless every data input position has been filled.

8. AUTO-SKIP forces the ACCEPT to be terminated when all data input positions have been filled. A terminator character explicitly keyed has its usual effect.

ACCEPT

9. BEEP causes an audible tone to sound when the system is ready to ACCEPT operator input.

For numeric and numeric-edited fields:

1. SPACE-FILL causes unkeyed integer digit positions of the data input field (not the receiving field) to be space-filled. Any leading operational sign is displayed in the right-most blank space.

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

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

4. 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.

5. PROMPT causes the data input field to be initialized as follows: digit positions to zero, decimal point position (if any) to the decimal point character, and sign position (if any) to a space.

6. UPDATE causes the data input field to be initialized to the current contents of the receiving field. These initial data are treated as if they were data you keyed. UPDATE supercedes the action of PROMPT if both are specified.

7. LENGTH-CHECK causes a received decimal point character to be ignored unless all integer digit positions have been keyed and a field terminator character to be ignored unless all digit positions have been keyed.

8. AUTO-SKIP causes the integer part of the ACCEPT to be terminated when all integer digit positions have been keyed and the entire ACCEPT to be terminated when all digit positions have been keyed.

9. BEEP causes an audible tone to sound when the system is ready to accept operator input.

Format 4 ACCEPT Statement

Uses

Format 4 of the ACCEPT statement causes each data input field in the group data item "screen-name" to be transferred from the terminal to its respective TO or USING field. Screen items having only VALUE or FROM clauses have no effect on the operation of the ACCEPT statement. Each such transfer consists of an implicit Format 3 ACCEPT of a field defined by the appropriate elementary screen item's PICTURE followed by an implicit MOVE to the associated TO or USING field.

Effects of Escape Key

If you press the ESC key during data input, the entire ACCEPT is terminated, the current field is not moved, the ESCAPE KEY value is set to 01, and the ON ESCAPE statement is executed. If you press a function key, the appropriate ESCAPE KEY value is set and the entire ACCEPT is terminated. If you press a field-terminator key (RETURN, TAB, etc.), the ESCAPE KEY value is set to 00 and the cursor moves to the next input field (if any) defined under screen-name. If the current field is the last field, the entire ACCEPT is terminated. If the backtab key is typed, the current field is terminated 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 terminated 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 you to tab forward or backward through the input fields without affecting the contents of the receiving items.

All the editing and validation features described in Format 3 ACCEPT statement also apply to the Format 4 ACCEPT. Furthermore, the PROMPT and TRAILING-SIGN functions that are optional in Format 3 are always in effect in a Format 4 ACCEPT.

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

ALPHABETICAL RESERVED WORD LIST

ACCEPT

Moves from screen fields to receiving items follow the standard COBOL-86 rules for MOVE statements, except that moves from numeric-edited fields are allowed. In this case, the data are input as if the field were numeric and the move uses only the sign, decimal point, and digit characters.

Displaying text or prompts in conjunction with a Format 4 ACCEPT requires the DISPLAY statement (see DISPLAY).

Application

Format 1:

Current system date is 6/30/82.
Current system time is 13:20:14:00.

```
77  WS-DATE PIC     X(6).
77  WS-TIME     PIC X(8).
```

```
ACCEPT WS-DATE FROM DATE
```

Result: `WS-DATE = 820630`

```
ACCEPT WS-TIME FROM TIME
```

Result: `WS-TIME = 13201400`

Format 2:

```
77  NUMERIC-VALUE PIC S9(5).
77  ALPHA-VALUE   PIC X(8).
77  EDIT-VALUE    PIC XX/XX/XX.
```

```
ACCEPT NUMERIC-VALUE
```

Terminal input = 123
Result: `NUMERIC-VALUE = +00123`

```
ACCEPT ALPHA-VALUE
```

Terminal input = "Mississippi"
Result: `ALPHA-VALUE = "Mississi"`

```
ACCEPT ALPHA-VALUE
```

Terminal input = "Utah"
Result: `ALPHA-VALUE = "Utah "`

```
ACCEPT EDIT-VALUE
```

Console input = "Illinois"
Result: `EDIT-VALUE = "Illinois"`

Format 3:

```
77  ALPHA-VALUE   PIC X(8).

MOVE 10 TO LIN
MOVE 20 TO COL
ACCEPT (LIN, COL) ALPHA-VALUE
```

Terminal input = "New York"
Result: Cursor moves to line 10, column 20 before input is accepted; `ALPHA-VALUE = "New York"`

```
MOVE 10 TO LIN
MOVE 20 TO COL
ACCEPT (LIN + 5, COL - 10) ALPHA-VALUE
```

Terminal input = "New York"
Result: Cursor moves to line 15, column 10 before input is accepted; `ALPHA-VALUE = "New York"`

```
ACCEPT (20, 10) ALPHA-VALUE
```

Terminal input = "New York"
Result: Cursor moves to line 20, column 10 before input is accepted; `ALPHA-VALUE = "New York"`

ALPHABETICAL RESERVED WORD LIST

ACCEPT

Format 4:

```
ACCEPT SCREEN-1 ON ESCAPE NEXT SENTENCE
```

Result: The cursor will move, in sequence, to the beginning of each on-screen data input field (as defined for SCREEN-1 in the SCREEN SECTION of the DATA DIVISION) and wait for terminal input. The data item entered will be temporarily loaded into the appropriate data-name (as designated in the SCREEN SECTION) and then immediately moved to the data-name specified after TO or USING. The MOVE follows the rules governing Format 3 ACCEPT statements. If at any time you press ESC, the ACCEPT statement will be aborted at its current state of completion and the NEXT SENTENCE will be executed.

ALPHABETICAL RESERVED WORD LIST

ADD

Syntax in PROCEDURE DIVISION

```
ADD {numeric-literal} ... {TO    } data-name-n
    {data-name-1    }     {GIVING}

[ROUNDED] [ON SIZE ERROR statement...]
```

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

Details

Use of the TO option causes all of the literals and data-names preceding the word TO to be added to the value found in data-name-n. Use of the GIVING option causes the sum of all the data-names and literals preceding the word GIVING to be placed in data-name-n. At least two values must be entered ahead of the word GIVING.

All of the operands (excluding literals) must be elementary numeric data items, except that the operand following GIVING may be a numeric edited item. Decimal point alignment and proper sizing of intermediate storage fields is provided automatically by the compiler, except that any intermediate result that cannot fit in 18 digits will be left-truncated.

Inclusion of the ON SIZE ERROR option makes the ADD statement conditional rather than imperative. If the integer portion of the result cannot fit in the specified receiving field, the receiving field will be unchanged and the statement(s) in the SIZE ERROR clause will be executed. If a size error occurs and no SIZE ERROR clause is present, no assumption should be made about the contents of the receiving field.

If the number of digits to the right of the decimal point in the result exceeds the number available in the result PICTURE clause, right truncation will occur unless you include the optional ROUNDED clause. If ROUNDED is specified, the right-most digit transferred to the result field will be increased by 1 whenever the most significant digit of the truncated portion is equal to or greater than 5. Negative values are affected in a similar fashion. If the result field is an integer containing one or more P editing characters, ROUNDED will add 1 to the right-most digit stored in the result field when the value masked by the left-most P is 5 or greater.

ALPHABETICAL RESERVED WORD LIST

ADD

Application

Data values at start of each application:

```
05  DATA-1 PIC S99  VALUE +99.
05  DATA-2 PIC S999 VALUE +99.
05  DATA-3 PIC SV9  VALUE +.9.
```

```
ADD 1 TO DATA-2
```

Result: DATA-2 = +100

```
ADD 1 DATA-1 GIVING DATA-2
```

Result: DATA-2 = +100

```
ADD DATA-3 TO DATA-2
```

```
ADD DATA-3 TO DATA-2 ROUNDED
```

Result: DATA-2 = + 100

```
ADD 1 TO DATA-1 ON SIZE ERROR DISPLAY "OVERFLOW"
```

Result: DATA-1 = +99, display statement executed

ALTER

Syntax in PROCEDURE DIVISION

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

The ALTER statement is used to change the operand of a GO TO statement or add an operand to a GO TO statement written without one.

Details

The line to be altered must be a single-line paragraph consisting of only a GO TO statement. The operand of the GO TO that follows paragraph-name will be replaced by the address of procedure-name. In practice, the alter statement usually follows the affected GO TO statement and modifies it to jump around code that should only be executed once. However, if GO TO is written without an operand, an ALTER statement must precede the GO TO during execution.

Application

```
GATE.
    GO TO MF-OPEN.
MF-OPEN.
    OPEN INPUT MASTER-FILE.
    ALTER GATE TO PROCEED TO NORMAL.
NORMAL.
    READ MASTER-FILE
```

BLANK

Syntax in DATA DIVISION

```
BLANK WHEN ZERO
```

This clause is appended to a data definition statement and causes the associated data item to contain only spaces if its numeric value is 0.

Details

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

Application

```
77 REPORT-VAL  PIC +9,999.999 BLANK WHEN ZERO.
```

BLOCK

Syntax in DATA DIVISION

```
BLOCK CONTAINS integer {RECORDS | CHARACTERS}
```

The BLOCK CONTAINS clause is appended to an FD paragraph and serves to specify the size of physical records in relation to the concept of logical records.

Details

Files you ASSIGN to the PRINTER must not have a BLOCK clause in the FD paragraph. Furthermore, the BLOCK clause has no effect on DISK files in COBOL-86, but it is examined for correct syntax. It is normally applicable to tape files, which are not supported by COBOL-86.

You should usually state the size of a physical block in RECORDS, except when the records are variable in size or exceed the size of a physical block. In these cases, express the size in CHARACTERS.

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

Application

```
FD  INPUT-FILE LABEL RECORDS ARE STANDARD
               BLOCK CONTAINS 5 RECORDS
```

ALPHABETICAL RESERVED WORD LIST

CALL

Syntax in PROCEDURE DIVISION

```
CALL literal [USING data-name ...]
```

The CALL statement transfers control to a separately compiled subprogram whose PROGRAM-ID is specified by the literal.

Details

Only a quoted (alphanumeric) literal may be used to specify the name of the subprogram. The subprogram name must be identical to the name given in its PROGRAM-ID paragraph.

Data-names you specify in the optional USING list are used to create a positional list of addresses to be passed to the subprogram. This permits data items defined in the calling program to be accessed by the subprogram. Since correspondence is by position, identical spelling of the data-names in each USING list is not required. Both lists, however, must contain the same number of data-names.

In addition to specifying them in both USING lists, data items to be shared must also be defined in the LINKAGE SECTION of the subprogram (see LINKAGE).

Application

Main program WORKING-STORAGE SECTION:

```
01  SHARED-DATA.
    05  SHARE-1 PIC 999.
    05  SHARE-2 PIC 999.

01  COMMON-DATA.
    05  COMMON-1 PIC XXX.
    05  COMMON-2 PIC XXX.
```

Main program PROCEDURE DIVISION:

```
CALL "SUBPROG" USING SHARED-DATA COMMON-DATA.
```

Subprogram LINKAGE SECTION:

```
01  GROUP-1.
    05  ITEM-1 PIC 999.
    05  ITEM-2 PIC 999.

01  GROUP-2.
    05  ITEM-3 PIC XXX.
    05  ITEM-4 PIC XXX.
```

Subprogram PROCEDURE DIVISION header:

```
PROCEDURE DIVISION USING GROUP-1 GROUP-2.
```

ALPHABETICAL RESERVED WORD LIST

CHAIN

Syntax in PROCEDURE DIVISION

```
CHAIN  { data-name-1 }  [USING data-name-2...]
       {     literal }
```

The CHAIN statement loads and executes another program. The chaining program is not preserved.

Details

Literal and data-name-1 must be alphanumeric. Data-name-1 must be followed by at least one space character. Each data-name-2 must be defined in the WORKING-STORAGE, LINKAGE, or FILE SECTION. If an identifier-2 is defined in the FILE SECTION, the associated file must be OPEN at the time CHAIN is executed.

When the CHAIN statement is executed, the value of identifier-1 or literal is interpreted as the name of a program executable under Z-DOS. When the program is found and loaded, all data and program structures of the chaining program are destroyed, except those passed to the chained program in the optional USING list.

There is no requirement that the chained program be a COBOL program. If it is, it must be a main program.

Application

Chaining program WORKING-STORAGE SECTION:

```
77  DATA-1 PIC X(8).
```

Chaining program PROCEDURE DIVISION:

```
PROCECURE DIVISION.
    CHAIN  "NEXT-PROGRAM.EXE" USING DATA-1.
    STOP RUN.
```

ALPHABETICAL RESERVED WORD LIST

CLOSE

Syntax in PROCEDURE DIVISION

```
CLOSE {filename [WITH LOCK]}...
```

A CLOSE statement is required to provide proper disposition of a file that will not be accessed again during the current job. Its function is opposite that of the OPEN statement.

Details

You may not attempt a READ, WRITE, or REWRITE function on a closed file, nor may you CLOSE a file that is not currently open. A fatal runtime error will result in each case.

Once the LOCK phrase is executed, any subsequent attempt to OPEN the file during the current job will cause a runtime error. If LOCK is omitted, the file may be reopened as needed by program logic.

Application

```
CLOSE MASTER-FILE-IN WITH LOCK, WORK-FILE.

CLOSE PRINT-FILE, TAX-RATE-FILE, JOB-PARAMETERS WITH LOCK.
```

ALPHABETICAL RESERVED WORD LIST

CODE-SET

Syntax in DATA DIVISION

```
CODE-SET IS ASCII
```

The CODE-SET clause normally specifies the character representation code used on a tape file when it is different from the internal code. Since tape files are not supported, this statement is nonfunctional in COBOL-86.

Details

All file character representation in COBOL-86 will be in the ASCII code. However, any signed numeric data description entries in the file's record should include the SIGN IS SEPARATE clause and all data in the file should be of DISPLAY USAGE.

Application

```
FD  INPUT-FILE LABEL RECORDS ARE STANDARD
             CODE-SET IS ASCII
```

COMPUTE

Syntax in PROCEDURE DIVISION

```
                                           ⎧ data-name-2          ⎫
COMPUTE data-name-1 [ROUNDED]... =  ⎨ numeric-literal      ⎬
                                           ⎩ arithmetic expression⎭

[ON SIZE ERROR statement...]
```

The COMPUTE statement evaluates the expression to the right of the equal sign and stores the value in data-name-1.

Details

Operands The operands to the right of the equal sign must be elementary numeric data items or numeric literals. The operands to the left of the equal sign may be numeric edited items. Decimal point alignment is provided automatically by the compiler.

Operators The operators may be any of the following:

**	specifies exponentiation
*	specifies multiplication
/	specifies division
+	specifies addition
−	specifies subtraction
(and)	specify alteration of evaluation priority.

Each of these operators must be preceded and followed by one or more spaces, except that a space is not required after a left parenthesis or before a right parenthesis. Additionally, the following unary operators are permitted:

+	specifies a positive value
−	specifies a negative value.

Enter a unary operator immediately preceding the associated literal. No space is allowed.

COMPUTE

Precedence for Evaluating the Expression

Expression evaluation occurs from left to right. Unless parentheses are used, the following order of precedence applies:

1. perform all unary operations
2. perform all exponentiation
3. perform all multiplication and division
4. perform all addition and subtraction.

Parentheses may be nested to any level to alter the normal evaluation sequence. The operation in the most deeply nested parentheses is completed first and the result stored in an intermediate field created by the compiler. An intermediate result that cannot fit in 18 digits will be left-truncated. The evaluation proceeds outward in a similar fashion until the operation in the outer-most parentheses is completed. The appropriate intermediate values are substituted at the proper time for each pair of parentheses and the operands and operators they enclose. When parentheses enclose only portions of an expression, operators not separated from their operands by a parenthesis follow the default evaluation priority.

Inclusion of the ON SIZE ERROR option makes the COMPUTE statement conditional rather than imperative. If the integer portion of the result cannot fit in the specified receiving field, the receiving field will be unchanged and the statement(s) in the SIZE ERROR clause will be executed. If a size error occurs and no SIZE ERROR clause is present, no assumption should be made about the contents of the receiving field.

If the number of digits to the right of the decimal point in the result exceeds the number available in the result PICTURE clause, right truncation will occur unless you include the optional ROUNDED clause. If ROUNDED is specified, the right-most digit transferred to the result field will be increased by 1 whenever the most significant digit of the truncated portion is equal to or greater than 5. Negative values are affected in a similar fashion. If the result field is an integer containing one or more P editing characters, ROUNDED will add 1 to the right-most digit stored in the result field if the value masked by the left-most P is 5 or greater.

Application

Data values at start of each application:

```
05  DATA-1 PIC S999     VALUE +120.
05  DATA-2 PIC S999     VALUE +140.
05  DATA-3 PIC S999     VALUE +160.
05  DATA-4 PIC ----.99 "6666.00"
```

```
COMPUTE DATA-4 = DATA-1 + DATA-2 / DATA-3
```

Result: DATA-4 = 120.87

```
COMPUTE DATA-4 ROUNDED = DATA-1 + DATA-2 / DATA-3
```

Result: DATA-4 = 120.88

```
COMPUTE DATA-1 ROUNDED = (DATA-1 + DATA-2) / DATA-3
```

Result: DATA-4 = 1.63

```
COMPUTE DATA-4 = 10 **3 ON SIZE ERROR DISPLAY "OVERFLOW".
```

Result: DATA-4 is unchanged, display statement executed

ALPHABETICAL RESERVED WORD LIST

CONFIGURATION

Syntax in 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].
```

The CONFIGURATION SECTION describes elements of the hardware environment not related to file processing. If included, the CONFIGURATION SECTION must occur first within the ENVIRONMENT DIVISION.

Details

The CONFIGURATION SECTION contains three possible paragraphs: SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL-NAMES.

The contents of the first two paragraphs are treated as commentary, except for the clause WITH DEBUGGING MODE. If this clause is included, PROCEDURE DIVISION statements containing a D in column seven will be compiled. If this clause is omitted, such statements are skipped during compilation.

The third paragraph, SPECIAL-NAMES, relates implementor names to names you define and changes default editing characters. The PRINTER IS phrase allows definition of a mnemonic name to be used in a DISPLAY UPON statement. A mnemonic name must follow the rules for word formation (see Part II).

Currency Symbols

If a currency symbol other than the dollar sign is desired, you may specify a single character non-numeric literal in the CURRENCY SIGN clause. However, the designated character may not be a quotation mark, a digit, nor any of the characters defined for PICTURE clause representations.

The "European" convention of separating integer and fractional positions of numbers with the comma character may be employed by specifying the DECIMAL-POINT IS COMMA clause.

ASCII Clause

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

ALPHABETICAL RESERVED WORD LIST

COPY

Syntax in PROCEDURE DIVISION

```
COPY filename.cob
```

The COPY statement inserts the contents of a disk file (other than the source file) in the source code input to the compiler.

Details

The filename must be a Z-DOS text file and must include an extension. You may include the COPY statement in a line that contains other source code.

Application

If a disk file named BDEF.COB contains this source code:

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

these two blocks of source code will compile exactly the same:

```
05  A.
    10 A1 PIC 9.
05  B.                           05 A.
    10 B1 PIC X.        and         10 A1 PIC 9 COPY BDEF.COB.
    10 B2 PIC X.                 05 C.
05  C.                              10 C1 PIC Z.
    10 C1 PIC Z.
```

Use this option to simplify source file development or to include optional or experimental modules in your compilation.

DATA (in DATA DIVISION Header)

Syntax in Division Header

```
DATA DIVISION.

[FILE SECTION.]

[WORKING-STORAGE SECTION.]

[LINKAGE SECTION.]

[SCREEN SECTION.]
```

The DATA DIVISION defines data storage areas used by the program.

Details

The DATA DIVISION is required and must follow the ENVIRONMENT DIVISION in every COBOL program. It is subdivided into four sections (see FILE, WORKING-STORAGE, LINKAGE, and SCREEN).

ALPHABETICAL RESERVED WORD LIST

DATA (in DATA RECORD Clause)

Syntax in DATA DIVISION

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

This clause is appended to an FD paragraph and serves to identify the records in the file by name.

Details

This clause is documentary only in all COBOL systems. The presence of more than one data-name indicates that the file area is defined by more than one record structure (01 level item). The order of the data-names is not significant.

Application

```
FD  INPUT-FILE LABEL RECORDS ARE STANDARD
               DATA RECORDS ARE RECORD-1 RECORD-2.

01  RECORD-1.
    05  ITEM-1        PIC XXX.
    05  ITEM-2        PIC XXX.

01  RECORD-2.
    05  ITEM-3        PIC X(6).
```

Syntax in PROCEDURE DIVISION

```
DECLARATIVES.

{section-name SECTION. USE AFTER STANDARD  { EXCEPTION }  PROCEDURE
                                           { ERROR     }

ON  { OUTPUT      }
    { INPUT       }
    { I-O         }
    { EXTEND      }
    { filename... }  .

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

END DECLARATIVES.
```

The DECLARATIVES subdivision provides a method with which you can create procedures that deal with I/O errors. These procedures are executed, not as part of the regular program logic flow, but rather when a condition occurs that you cannot normally test.

Details

If you include the DECLARATIVES subdivision, it must occur immediately below the PROCEDURE DIVISION header. The statements DECLARATIVES and END DECLARATIVES are both written in area A. The USE sentence must immediately follow a SECTION header within the DECLARATIVES subdivision and must be terminated by a period/space sequence.

Additional Procedures

Although the COBOL-86 system automatically handles checking and creation of standard labels and executes error recovery routines when I/O errors occur, you may, at your option, specify additional procedures.

Related procedures are grouped under the same SECTION header and USE sentence. The USE sentence is not executed, but serves to specify the applicability of the procedure. The details of the procedure are then coded in standard sentence and paragraph formats. A declarative section ends with the occurrence of a SECTION header for a subsequent section, or, if none exists, the END DECLARATIVES statement.

DECLARATIVES

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

Within a DECLARATIVES procedure there must be no reference to any nondeclarative procedure. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the DECLARATIVES subdivision, except that PERFORM statements may refer to a USE statement and its procedures. (If PERFORM THRU is used to refer to a DECLARATIVES procedure, its entire range must lie within the DECLARATIVES subdivision.)

Implicit EXIT

An implicit EXIT from a declarative section is inserted by the compiler following the last statement in each section. All logical paths within the section must terminate at this exit point.

DISPLAY

Syntax in PROCEDURE DIVISION

```
                           ⎧ data-name   ⎫
DISPLAY [position-spec]    ⎨ screen-name ⎬ ... [UPON mnemonic-name]
                           ⎪ literal     ⎪
                           ⎩ ERASE       ⎭
```

The DISPLAY statement permits low-volume data to be output to the screen or printer; a file definition is not required.

Details

The position-spec may be repeated for each display-item. The length of the display-item may not be greater than 1920 characters. If you include the mnemonic-name, it must be defined in the PRINTER IS clause of the SPECIAL-NAMES paragraph in the CONFIGURATION SECTION. A screen-name, if included, must be defined in the SCREEN-SECTION.

Output is directed to the screen unless `UPON mnemonic-name` is included. The printer is the only device assignable through the mnemonic-name option.

Each display-item is processed in the order of its occurrence in the DISPLAY statement. After every item has been processed, a return/line-feed sequence will be sent, provided no position-specs were used in the statement.

A position-spec, if included, positions the cursor at the coordinates you choose prior to the transfer of data. It may be expressed in either of the following forms:

```
(LIN[ {+ | -} integer-1], COL[ {+ | -} integer-2])
```

```
(integer-3, integer-4)
```

The opening and closing parentheses and the comma are required. The position-spec specifies the position on the screen where the data output field will begin.

DISPLAY

Special Registers LIN and COL

LIN and COL are COBOL special registers. Each behaves like a numeric data item with USAGE COMP, but you may reference a special register without having declared it in the DATA DIVISION.

If you specify LIN, the data output field will begin on the screen row whose number is equal to the value of the LIN special register. If + integer-1 or − integer-1 is included, the effective value of LIN will be increased or decreased accordingly, without affecting the value in the LIN register. If you specify integer-3, the data output field will begin on the row whose number is integer-3. If you specify neither LIN nor integer-3, the data output field will begin on the screen row containing the current cursor position.

If you specify COL, the data output field will begin in the screen column whose number is equal to the value of the COL special register. If + integer-2 or − integer-2 is included, the effective value of COL will be increased or decreased accordingly, without affecting the value in the COL register. If you specify integer-4, the data output field will begin in the column whose number is integer-4. If you specify neither COL nor integer-4, the data output field will begin in the screen column containing the current cursor position.

Use of Data-name, Literal, and ERASE

If you specify either a data-name or a literal among the operands, the contents of the data-name or the value of the literal is sent to the receiving device. Since no data conversion occurs during processing, only data-names whose USAGE IS DISPLAY should be included as operands.

If you include ERASE among the operands and a position-spec has been coded for this or any previous operand, the screen will be cleared from the current cursor position to the bottom of the screen. If no position-spec has been coded for the next display-item, the initial cursor position will be the position from which the ERASE was begun. If you specify ERASE and no position-spec has been coded prior to its occurrence, it will be ignored.

Use of Screen-name

If you include a screen-name among the operands, the contents of each elementary data-item that includes a VALUE, FROM, or USING clause will be transferred to the appropriate position on the screen. Except in the case of a VALUE clause, DISPLAY of a screen-item consists of a MOVE from its source into the appropriate field of the screen definition, followed by a "DISPLAY data-name" utilizing the associated position-spec. For a field having only a TO clause, the effect will be as if FROM ALL "." (period) had been specified.

Application

```
77  WS-ENROUTE-TIME  PIC 99 VALUE 34.

DISPLAY "TIME ENROUTE: " WS-ENROUTE-TIME " MINUTES"
```

Result: Screen displays "TIME ENROUTE: 34 MINUTES" at the current cursor position.

```
MOVE 10 TO LIN
MOVE 1 TO COL
DISPLAY (LIN, COL) "TIME ENROUTE:" (LIN + 2, COL + 4)
WS-ENROUTE-TIME " MINUTES"
```

Result: After positioning the cursor at line 10, column 1, the screen displays

```
TIME ENROUTE:

    34 MINUTES
```

```
DISPLAY (1, 1) ERASE "TIME ENROUTE: " WS-ENROUTE-TIME " MINUTES"
```

Result: The entire screen is cleared, the cursor is repositioned at line 1, column 1, and the screen displays

```
TIME ENROUTE: 34 MINUTES.
```

ALPHABETICAL RESERVED WORD LIST

DIVIDE

Syntax in PROCEDURE DIVISION

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

[ROUNDED] [ON SIZE ERROR statement...]
```

The DIVIDE statement divides two numeric values and stores the quotient.

Details

When you use BY in a DIVIDE statement, the first value in the statement is the dividend and the second value is the divisor. When INTO is used, the first value is the divisor and the second is the dividend. The quotient is stored in the dividend unless you include the optional GIVING clause. GIVING causes the quotient to be stored in data-name-3. GIVING is required if the dividend is a numeric literal or with the BY clause.

All of the operands (excluding literals) must be elementary numeric data items, except that the operand following GIVING may be a numeric edited item. Decimal point alignment and proper sizing of intermediate storage fields is provided automatically by the compiler, except that any intermediate result that cannot fit in 18 digits will be left-truncated.

Inclusion of the ON SIZE ERROR option makes the DIVIDE statement conditional rather than imperative. If the integer portion of the result cannot fit in the specified receiving field, the receiving field will be unchanged and the statement(s) in the SIZE ERROR clause will be executed. If a size error occurs and no SIZE ERROR clause is present, no assumption should be made about the contents of the receiving field. Division by 0 always causes a size error condition.

If the number of digits to the right of the decimal point in the result exceeds the number available in the result PICTURE clause, right truncation will occur unless you include the optional ROUNDED clause. If ROUNDED is specified, the right-most digit transferred to the result field will be increased by 1 whenever the most significant digit of the truncated portion is equal to or greater than 5. Negative values are affected in a similar fashion. If the result field is an integer containing one or more P editing characters, ROUNDED will add 1 to the right-most digit stored in the result field when the value masked by the left-most P is 5 or greater.

Application

Data values at start of each application:

```
05  DATA-1 PIC 99 VALUE 10.
05  DATA-2 PIC 99 VALUE 20.
```

```
DIVIDE DATA-1 INTO DATA-2
```

Result: `DATA-2 = 2`

```
DIVIDE DATA-2 BY DATA-1 GIVING DATA-2
```

Result: `DATA-2 = 2`

```
DIVIDE 40 BY DATA-2 GIVING DATA-1
```

Result: `DATA-1 = 2`

```
DIVIDE DATA-2 INTO DATA-1
```

Result: `DATA-1 = 0`

```
DIVIDE DATA-2 INTO DATA-1 ROUNDED
```

Result: `DATA-1 = 1`

```
DIVIDE DATA-1 BY .1 GIVING DATA-2 ON SIZE ERROR DISPLAY "OVERFLOW".
```

Result: `DATA-1 = 10, display statement executed`

ALPHABETICAL RESERVED WORD LIST

ENVIRONMENT

Syntax in Division Header

```
ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.]

[INPUT-OUTPUT SECTION.]
```

The ENVIRONMENT DIVISION provides a standard method of describing the aspects of a COBOL program that are dependent upon the physical aspects of the host computer.

Details

The ENVIRONMENT DIVISION is required and must follow the IDENTIFICATION DIVISION in every COBOL program. It is subdivided into two sections (see CONFIGURATION, INPUT-OUTPUT).

EXHIBIT

Syntax in PROCEDURE DIVISION

```
                                     { literal   }
EXHIBIT NAMED [position-spec]        { data-name } ...
                                     { ERASE     }

[UPON mnemonic-name]
```

EXHIBIT is a debugging function that causes the values of selected data items to be displayed at key points during program execution.

Details

The position-spec, ERASE, and UPON follow the same syntax conventions as those described for the DISPLAY statement. If you write the EXHIBIT statement in a line that includes a D in column 7, it will be ignored unless you specify WITH DEBUGGING MODE in the SOURCE-COMPUTER paragraph in the CONFIGURATION SECTION.

Application

Source code:

```
WORKING-STORAGE SECTION.
    77 VAL1 PIC XXX VALUE "YES".
    77 VAL2 PIC XXX VALUE "NO ".

PROCEDURE DIVISION.
MAIN.
    EXHIBIT VAL1 " " VAL2.
```

Screen displays:

```
VAL1 = YES VAL2 = NO
```

ALPHABETICAL RESERVED WORD LIST

EXIT

Syntax in PROCEDURE DIVISION

```
paragraph-name.
    EXIT.
```

The EXIT statement is used to terminate execution of a procedure prior to completion.

Details

The EXIT statement may be used only as a single-word paragraph preceded by a paragraph-name. It may be invoked by transferring control to paragraph-name or by "falling through." Control will then be transferred to the same statement that would have been executed after the last instruction in the procedure.

Application

```
000010 MAINLINE-CONTROL.
000020     PERFORM ANOTHER-PROCEDURE.
000030     PERFORM STILL-ANOTHER-PROCEDURE.
000040     STOP RUN.

000050 EXIT-PROCEDURE.
000060     EXIT.

000070 ANOTHER-PROCEDURE.
000080     IF ERR-FLAG NOT = "YES" GO TO EXIT-PROCEDURE.
000090     PERFORM PROCESS-ERROR.
```

If EXIT-PROCEDURE is executed (@ line 80), line 90 will be skipped and control will return to line 30.

EXIT PROGRAM

Syntax in PROCEDURE DIVISION

```
paragraph-name.
    EXIT PROGRAM.
```

This statement terminates execution of a subprogram.

Details

The EXIT PROGRAM statement must be a paragraph by itself. Control returns to the statement following the CALL statement in the main program, unless program flow is affected by completion of an active PERFORM statement. If EXIT PROGRAM is encountered in a main program, it is assumed to mean EXIT.

Application

```
SUBPROG-MAINLINE.
    PERFORM PROCEDURE-FIRST.
    PERFORM PROCEDURE-LAST.
EXIT-PROCEDURE.
    EXIT PROGRAM.
```

FILE

Syntax in DATA DIVISION

```
FILE SECTION.

{FD filename

LABEL {RECORD | RECORDS} {IS | ARE} {OMITTED | STANDARD}

{data-description-entry}... }...
```

The FILE SECTION contains one or more FD paragraphs in which the data file structures used by the program are defined. Every file that appears in a SELECT clause in the ENVIRONMENT DIVISION must also have an FD (file definition) paragraph in the FILE SECTION. This paragraph precedes data definition statements for the file.

Details

The FILE SECTION is required, unless a program relies exclusively on ACCEPT and DISPLAY statements for its I/O. When included, it must occur first in the DATA DIVISION.

The filename must be constructed according to the rules for word formation presented in Chapter 7 of Part II. Filename as specified in an FD paragraph refers to the data structure coded within the DATA DIVISION and referenced in I/O related statements such as OPEN, CLOSE, and READ. It is not to be confused with FILE-ID as used in the VALUE OF FILE-ID clause.

Numerous optional clauses applicable to the FD paragraph are treated separately. See VALUE, DATA, BLOCK, RECORD, CODE-SET, and LINAGE.

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 and conform to system specifications; you must specify the STANDARD option for files assigned to disk.

Selection of the singular or plural expression has no effect on the compilation.

Application

```
FILE SECTION.

FD  INPUT-FILE LABEL RECORDS ARE STANDARD.

01  INPUT-RECORD.
    05  FILLER       PIC X(80).

FD  OUTPUT-FILE LABEL RECORDS ARE OMITTED.

01  OUTPUT-RECORD.
    05  FILLER       PIC X(132).
```

ALPHABETICAL RESERVED WORD LIST

GO TO

Syntax in PROCEDURE DIVISION

```
GO TO procedure-name

    [... DEPENDING ON data-name]
```

The GO TO statement is used to transfer control to a specific section or paragraph.

Details

The procedure-name may be either a section-name or a paragraph-name. Multiple procedure-names may be used if the DEPENDING clause is specified. Control will then be passed to the procedure-name in the position that corresponds to the value of the data-name. For example, if data-name has a value of 2, control will pass to the second procedure-name listed. If data-name has a value that is not equal to the position of any of the procedure-names, the GO TO instruction will be ignored during execution. The data-name must be an elementary item and a whole number.

A GO TO statement may be written without an operand provided it constitutes a single-sentence paragraph and is modified by an ALTER statement before it is executed (see ALTER).

Application

```
GO TO SECTION-ONE.

GO TO PARAGRAPH-ONE.

GO TO PARA-1 PARA-2 PARA-3 DEPENDING ON JUMP-VECTOR.
```

IDENTIFICATION

Syntax in Division Header

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

The IDENTIFICATION DIVISION provides program documentation.

Details

The IDENTIFICATION DIVISION is required at the beginning of every COBOL program.

PROGRAM-ID is the only required paragraph. Optional paragraphs, if you include them, must follow the PROGRAM-ID.

The program-name may be any alphanumeric string of characters that begins with an alphabetic character and does not contain an embedded period. Only the first six characters are retained by the compiler. The program-name identifies the object program and is printed in the heading portion of each listing page.

ALPHABETICAL RESERVED WORD LIST

IF

Syntax in PROCEDURE DIVISION

```
    ⎧ class-test     ⎫
IF  ⎨ condition-name ⎬  ... action... [ELSE alternate-action...]
    ⎪ sign-test      ⎪
    ⎩ relation-test  ⎭
```

The IF statement causes a series of statements to be executed contingent on the basis of one or more specified conditions. An alternate series of statements may be specified that will be executed only if the initial series is not. If more than one subject is specified, each must be joined to the others by an AND or OR operator.

Details

If you specify more than one subject (i.e., class-test, condition-name, sign-test, or relation-test), each must be joined to the others by an AND or OR operator to stipulate their logical interrelationship. Also, the logical operator NOT may precede any of the subjects to specify logical negation.

Evaluation of Multiple Subjects

If you specify more than two subjects, the following order of evaluation is applied: First, each simple subject is evaluated, then, reading from left to right, each pair of subjects joined by AND is evaluated, finally, beginning again at the left, each of the resultant OR relationships is evaluated. Parentheses may be inserted to alter the normal evaluation precedence, to clarify the precedence in a lengthy statement, or to include compound conditions within other compound conditions.

If the entirety of the compound subject evaluates true, the statements immediately following are executed. If it evaluates false, the statements following ELSE are executed. If ELSE is not specified, control is passed to the next sentence.

Compound conditional statements may be abbreviated if each condition shares a common subject. If both the subject and the relational operator are common to each condition, both may be omitted. For example, these two statements are equivalent:

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

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

If NOT is included in an abbreviated condition, two considerations apply:

1. If the item immediately following NOT is a relational operator, then the NOT participates as part of the relational operator.

2. If the item following NOT is not a relational operator, then the abbreviated condition ends and a new condition begins.

You may specify multiple actions for both the true and the false alternatives. The expression NEXT SENTENCE may be used for either alternative.

Nesting IF-ELSE Statements

The power of conditional statements may also be enhanced through nesting, a structure in which IF occurs more than once in a single statement. ELSE may also be repeated up to the number of times IF is used in the same statement. There are two forms of nesting: linear and nonlinear. They may be combined in a single statement.

The execution of each IF in a linear nested conditional is contingent upon the result of the entire preceding portion of the conditional statement. Each occurrence of ELSE is associated with the nearest preceding IF. For example:

```
IF condition-1
    IF condition-2 action-1
        ELSE IF condition-3 action-2
            ELSE action-3.
```

If condition-1 is false, control passes out of the statement. If condition-1 and condition-2 are true, action-1 is performed. If -1 is true and -2 is false, either action-2 or action-3 is performed, dependent upon condition-3.

IF

In a nonlinear nested conditional, ELSE clauses do not intervene between two IF clauses. Instead, the first IF is paired with the last occurrence of ELSE. The second IF is paired with the next to the last ELSE, and so on. For example:

```
IF condition-1
     IF condition-2 action-2
        ELSE action-3
ELSE action-1.
```

Condition-2 is evaluated only if condition-1 is true. If condition-1 is false, action-1 is performed. See the Application for further examples of nested conditions.

Relation-test

A *relation-test* evaluates the logical or mathematical relationship of two operands, both of which must be either data items, literals, or figurative constants. The relation to be tested is specified by coding one of the expressions [NOT] EQUAL TO, [NOT] LESS THAN, or [NOT] GREATER THAN between the operands being tested. The symbols =, <, and > are accepted equivalents of these expressions.

If the operands being tested are numeric, the test procedure provides automatic decimal point alignment. Negative values are considered to be less than zero and positive values greater than zero. Numeric operands need not be of the same USAGE. Index names and index data items are legal operands.

If the operands being tested are alphanumeric, the collating sequence is ASCII. The test is performed character by character from left to right. When items of unequal length are compared, the shorter item is assumed to contain enough trailing spaces to match the length of the longer item. An alphanumeric item may be compared to a numeric item if the numeric item is an integer whose USAGE IS DISPLAY. A relation-test that includes a group data item is always treated as an alphanumeric comparison.

Class-test

A *class-test* determines whether the contents of a specified data item are (or are NOT) alphabetic or numeric. It is coded in the form:

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

The NUMERIC test cannot be applied to an item with an alphabetic PICTURE (e.g., PIC AA). The ALPHABETIC test may be applied only to an item with an alphanumeric PICTURE (e.g., PIC XX). For the NUMERIC test to return true, the data item may contain only valid digits, a sign, and an assumed decimal point. For the ALPHABETIC test to return true, the data item may contain only the letters A-Z and spaces. A class-test may be applied to both group and elementary data items.

Sign-test

A *sign-test* determines whether the sign of a numeric data item is (or is NOT) a specified sign. It is coded in the form:

```
data-name IS [NOT] {NEGATIVE | ZERO | POSITIVE}
```

Condition-name

A *condition-name*, as defined by an 88-level entry in the DATA DIVISION, may be used as the subject of a conditional statement. It may be preceded by NOT to imply logical negation. For the test to return true, the current value of the associated elementary data item must equal (or NOT equal) a value within the list or range specified in the 88-level entry.

Application

Simple conditions:

Relation–

```
IF A > B PERFORM ERROR-ROUTINE.
```

ALPHABETICAL RESERVED WORD LIST

IF

Class–

```
IF C IS NOT NUMERIC PERFORM ERROR-ROUTINE.
```

Sign–

```
IF A IS NEGATIVE PERFORM ERROR-ROUTINE
ELSE PERFORM NEGATIVE-BALANCE-ROUTINE.
```

Condition–

```
IF OUT-OF-RANGE GO TO EXIT-PROCEDURE.
```

Compound conditions:

```
IF A > B AND B > C OR A > 8 NEXT SENTENCE
    ELSE PERFORM ERROR-ROUTINE.

IF A IS NEGATIVE OR NOT NUMERIC PERFORM ERROR-ROUTINE.

IF A = 7 OR 9 OR 11 GO TO JACKPOT
    ELSE PERFORM NEXT-ROLL.
```

Linear nested conditional:

```
IF A > B MOVE "YES" TO RELATION-SWITCH
    ELSE IF A < B MOVE "NO " TO RELATION-SWITCH
        ELSE MOVE SPACES TO RELATION-SWITCH.
```

Nonlinear nested conditional:

```
IF NO-ERROR
    IF BALANCE-IN IS POSITIVE
        IF FLAG-OFF PERFORM MONTHLY-STATEMENT
        ELSE PERFORM FLAGGED-ACCT-REPORT
    ELSE PERFORM NEG-BALANCE-STATEMENT
ELSE PERFORM ERROR-ROUTINE.
```

IN, OF

Syntax in PROCEDURE DIVISION

```
identifier-1 {IN | OF} identifier-2
```

Reserved words OF and IN permit the use of non-unique data-names, condition-names, and paragraph-names by specifying the intended location as a qualifier.

Details

Identifier-1 is non-unique and is defined at a higher level number than identifier-2, which is a group item or filename containing the first. The second name must either be unique or must itself be qualified. You may chain qualifiers of data-names and condition-names up to a limit of five. Paragraph-names may be qualified by a section-name. Filenames and mnemonic-names must be unique. They cannot be qualified.

A qualified name may only be written in the SCREEN SECTION or PROCEDURE DIVISION. A reference to a multiply defined paragraph-name need not be qualified when referred to from within the same section.

Application

```
YEAR OF RETIREMENT-AGE

RATE OF AUTO-EXPENSE IN TRAVEL-EXPENSE-TABLE

CONTROL-FIELD IN INPUT-FILE
```

ALPHABETICAL RESERVED WORD LIST

INPUT-OUTPUT

Syntax in ENVIRONMENT DIVISION

```
INPUT-OUTPUT SECTION.

[FILE-CONTROL.]

    [SELECT filename ASSIGN TO {DISK | PRINTER}

    [RESERVE integer {AREAS | AREA} ]

    [FILE STATUS IS data-name-1]

    [RECORD KEY IS data-name-2]

    [RELATIVE KEY IS data-name-3]

    [ACCESS MODE IS {SEQUENTIAL | RANDOM | DYNAMIC} ]

    [ORGANIZATION IS
     {SEQUENTIAL | LINE SEQUENTIAL | INDEXED | RELATIVE} ] ]...

[I-O-CONTROL.]

    [SAME {RECORD | SORT | SORT-MERGE} AREA FOR filename...]...
```

The INPUT-OUTPUT SECTION describes elements of the hardware environment that pertain to file processing. If both the INPUT-OUTPUT SECTION and CONFIGURATION SECTION are included, the INPUT-OUTPUT SECTION must appear last.

Details

The INPUT-OUTPUT SECTION is required if the program uses data files. It contains two possible paragraphs, FILE-CONTROL and I-O-CONTROL, which are used to define file assignment parameters and buffering specifications.

FILE-CONTROL Paragraph

Each file defined by an FD entry in the DATA DIVISION must have a SELECT statement in the FILE-CONTROL paragraph. Clauses shown as optional in the SELECT syntax may be written in any order. Random and dynamic access modes are applicable only to files whose ORGANIZATION is relative or indexed. The RECORD KEY clause is applicable only to indexed files, and the RELATIVE KEY clause is applicable only to relative files. A detailed treatment of indexed and relative files is provided in Chapters 10 and 11 in Part II.

Serial Files

Serial files utilize either of two formats, which are specified by the phrases `ORGANIZATION IS SEQUENTIAL` and `ORGANIZATION IS LINE SEQUENTIAL`. Both formats assume the file records are variable-length. The regular SEQUENTIAL organization is a two-byte count of the record length followed by the record itself for each record in the file. LINE SEQUENTIAL organization follows each record in the file with a return/line feed delimiter.

No COMP-0 or COMP-3 information should be written into a LINE SEQUENTIAL file, since these items may contain the same binary codes used for return/line feed, thus causing an erroneous end-of-record indication.

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

RESERVE Clause

The RESERVE clause is not functional in COBOL-86, but 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 span physical block boundaries. For files assigned to PRINTER, the logical record area is used as the physical buffer as well.

FILE STATUS and Operation Outcome

In the FILE STATUS clause, data-name-1 must refer to a two-character alphanumeric item defined in the WORKING-STORAGE or LINKAGE SECTION. After each I-O statement is executed, the runtime data management facility places information in the FILE STATUS item that describes the outcome of the operation. Data-name-1 assumes one of the following values:

INPUT-OUTPUT

'00' for successful completion
'10' for end-of-file indication
'20' for INVALID KEY (pertains to indexed and relative files)
'30' for file not found
'34' for disk space full

A '0' in the right-hand character of data-name-1 indicates that no STATUS information exists for the I-O operation beyond what is expressed by the left-hand character. Additional FILE STATUS values that pertain to indexed and relative files can be found in Chapters 10 and 11 in Part II.

I-O-CONTROL Paragraph

Only the SAME RECORD AREA form is functional in COBOL-86. The other forms are permitted only in versions that support the SORT utility and, when allowed, are only scanned for correct syntax.

The SAME RECORD AREA form causes all the named files to share the same logical record area in order to conserve memory space. It is not necessary for the named files to have the same ACCESS or ORGANIZATION; however, no filename may be listed in more than one SAME AREA clause.

Syntax in PROCEDURE DIVISION

```
                                                   { CHARACTERS          }
INSPECT data-name-1 [TALLYING data-name-2 FOR     { LEADING  operand-3  }
                                                   { ALL      operand-3  }

                                                   { CHARACTERS          }
[ { BEFORE }  INITIAL operand-4]] [REPLACING       { LEADING operand-5   }
  { AFTER  }                                       { FIRST   operand-5   }
                                                   { ALL     operand-5   }

BY operand-6 [ { BEFORE }  INITIAL operand-7]]
               { AFTER  }
```

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

1. count the occurrences of a character.
2. replace a specified character with another.
3. limit the above actions by requiring the occurrence of other specified characters.

NOTE: Either the TALLYING or the REPLACING clause must be specified; they may not both be omitted.

Details

data-name-1 must be of DISPLAY usage. data-name-2 must be numeric. Fields designated operand-n must have a length of one character and may be quoted literals, figurative constants, or data items. The TALLYING clause must precede a REPLACING clause if both are used in the same statement. The effect of including both TALLYING and REPLACING is as if two INSPECT statements had been written, one including only a TALLYING clause and the other only a REPLACING clause.

TALLYING causes character by character comparison, from left to right, of data-name-1 with the character specified by operand-3. Data-name-2 is incremented by 1 each time a match is found. If you include the optional AFTER INITIAL operand-4 clause, counting begins only after a match is found with the character specified by operand-4. Similarly, if you include BEFORE INITIAL operand-4, counting stops when a match is found for the character specified by operand-4.

INSPECT

REPLACING causes substitution, from left to right, of the character specified by operand-6 for each occurrence in data-name-1 of the character specified by operand-5. If the optional BEFORE or AFTER is included, the substitution ends or begins when the character specified by operand-7 is first matched.

These general descriptions must be supplemented by the effects of the words CHARACTERS, ALL, LEADING, and FIRST.

CHARACTERS—stipulates that every character in data-name-1 is to be affected by the TALLY and/or REPLACE operation. The only means to limit its scope is by inclusion of the optional BEFORE AFTER clause.

ALL—stipulates that all characters matching the succeeding operand are to be affected.

LEADING—stipulates that all characters that match the succeeding operand and that occur contiguously from the first position of data-name-1 be affected.

FIRST—stipulates that only the first occurrence of the succeeding operand be affected. Since the result would always be equal to 1, this option is illegal in a TALLYING clause.

Because TALLYING increments data-name-2, its final value is equal to its initial value plus the number of matches found.

Data-name-1 is always treated as a character string, regardless of its PICTURE. Therefore a position containing an embedded sign will be treated as a single unsigned character.

Application

Data values at start of each application:

```
05  STRING-1  PIC X(13) VALUE "0000123456789".
05  COUNTER   PIC 9(18) VALUE 0.
```

```
INSPECT STRING-1 TALLYING COUNTER FOR CHARACTERS
```

Result: `COUNTER = 13`

```
INSPECT STRING-1 TALLYING COUNTER FOR CHARACTERS AFTER "1"
```

Result: `COUNTER = 8`

```
INSPECT STRING-1 TALLYING COUNTER FOR ALL "5" AFTER "1"
```

Result: `COUNTER = 1`

```
INSPECT STRING-1 TALLYING COUNTER FOR ALL "5" BEFORE "1"
```

Result: `COUNTER = 0`

```
INSPECT STRING-1 REPLACING LEADING "0" BY " "
```

Result: `STRING-1 = "    123456789"`

```
INSPECT STRING-1 REPLACING CHARACTERS BY "+" AFTER "1"
```

Result: `STRING-1 = "00001++++++++"`

```
INSPECT STRING-1 REPLACING ALL "5" BY "+" AFTER "1"
```

Result: `STRING-1 = "00001234+6789"`

```
INSPECT STRING-1 REPLACING ALL "5" BY "+" BEFORE "1"
```

Result: `STRING-1 = "0000123456789"`

ALPHABETICAL RESERVED WORD LIST

JUSTIFIED

Syntax in DATA DIVISION

```
JUSTIFIED RIGHT
```

This clause is appended to a data definition statement, and causes right justification when the data item is used as a receiving field.

Details

JUSTIFIED is only allowed with unedited alphanumeric fields. If the field is larger than the item it receives, left space fill will occur. If the field is shorter than the item, left truncation will occur. JUST is an acceptable abbreviation of JUSTIFIED.

Application

```
77  ALPHA-VAL   PICX(6) JUSTIFIED RIGHT.
```

Syntax in DATA DIVISION

```
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} ]
```

The LINAGE clause provides a means for specifying the size of the printable portion of a page (called the "page body"), the size of the top and bottom margins, and the line number at which the footing area begins.

Details

The LINAGE clause should be specified only for files assigned to the PRINTER. All data-names must refer to unsigned numeric integer data items. Integer-1 must be greaater 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 margin sizes are not specified, zero is assumed. The footing area comprises the part of the page body between the line indicated by the FOOTING value and the last line of the page body, inclusive.

The values in each phrase at the time the file is opened specify the number of lines that comprise each of the sections of the first logical page. Whenever a WRTIE statement with the ADVANCING PAGE phrase is executed or a "page overflow" condition occurs (see WRITE), the values in each phrase, at that time, will be used to specify the number of lines in each section of the next logical page.

Modifying the LINAGE-COUNTER

A LINAGE-COUNTER is created by the presence of the 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 may not be modified, by PROCEDURE DIVISION statements. It is automatically modified during execution of a WRITE statement, according to the following rules:

ALPHABETICAL RESERVED WORD LIST

LINAGE

1. When you specify the ADVANCING PAGE phrase of the WRITE statement or a page overflow condition occurs (see WRITE), the LINAGE-COUNTER is reset to 1.

2. When you specify an ADVANCING data-name LINES or ADVANCING integer LINES phrase, LINAGE-COUNTER is incremented by the ADVANCING value.

3. When you do not specify the ADVANCING phrase, LINAGE-COUNTER is incremented by one.

Application

```
FD  PRINT-FILE LABEL RECORDS ARE OMITTED
            LINAGE 55 TOP 5 BOTTOM 5
```

LINKAGE

Syntax in DATA DIVISION

```
LINKAGE SECTION.
```

The LINKAGE SECTION is the section in which externally stored data items are described. This section is permitted only in a subprogram, and, if used, occurs immediately after the WORKING-STORAGE SECTION.

Details

No storage is allocated by data descriptions written in the LINKAGE SECTION. Instead, these data descriptions duplicate the structure of data items defined in the DATA DIVISION of the main (calling) program, where storage is actually reserved. Consequently, VALUE clauses other than condition-names are prohibited in the LINKAGE SECTION.

Only descriptions of data items that are passed to or from the main program are included in the LINKAGE SECTION. Memory mapping is achieved through the USING list (see CALL).

Application

In main program:

```
WORKING-STORAGE SECTION.

01  SHARED-DATA-ITEMS.
    05  REAL-DEFINITION-1       PIC S9(5)V99.
    05  REAL-DEFINITION-2       PIC SV999.
```

In subprogram:

```
LINKAGE SECTION.

01  LINKED-ITEMS.
    05  ITEM-1                  PIC S9(5)V99.
    05  ITEM-2                  PIC SV999.
```

MOVE

Syntax in PROCEDURE DIVISION

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

MOVE is the COBOL assignment statement. It is used to move data from one area of main storage to another and to perform conversions and/or editing on the data that are moved.

Details

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

Subscripts or indexes associated with data-name-2 are evaluated immediately before data are moved to the receiving field. The same is true for any optional receiving fields. However, any subscripts associated with the sending field (data-name-1) is evaluated only once, before any data are sent. For example, in the following statement the receiving fields B and C(B) are set to the same value, which is the value of A(B) before the instruction is executed:

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

Rules for Using MOVE

The following considerations pertain to use of the MOVE instruction:

1. Numeric or alphanumeric source with a numeric or numeric-edited (report) destination:

    a. Decimal alignment will occur with truncating or zero padding both ends of the field, as needed. If the source is alphanumeric, it is treated as an unsigned integer and should not be longer than 31 characters.

b. The data moved assume the type specified in the destination PICTURE clause. `USAGE IS DISPLAY` is assumed for alphanumeric sending fields.

c. Editing will occur as specified by editing symbols contained in the destination PICTURE clause. Editing will not occur, however, if the destination is a group item.

d. An alphabetic or alphanumeric edited item cannot be moved to a numeric or numeric-edited (report) field. Moves to alphabetic fields are limited to items containing only letters of the alphabet. Unsigned numeric integers and numeric-edited items can be moved to alphanumeric or alphanumeric-edited fields.

2. Non-numeric source and destination:

   a. Characters are placed in the receiving area from left to right, unless you specify `JUSTIFIED RIGHT` for the destination.

   b. Padding with spaces will occur if the receiving field is longer than the data item.

   c. Truncating will occur if the receiving field is shorter than the data item.

3. Results are unpredictable when field boundaries are not aligned in a group move.

4. Table 12.2 contains all legal source and destination field combinations.

5. Data items for which you specify `USAGE IS INDEX` are not legal operands in the MOVE statement (see SET).

Application

Examples of data moves (b represents a blank) are shown in Table 12.3.

ALPHABETICAL RESERVED WORD LIST

MOVE

Table 12.2. Permissible MOVE Operands

	RECEIVING OPERAND IN 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 Noninteger	OK	OK	OK			OK (B)
Numeric-edited				OK	OK	OK (B)
Alphanumeric-Edited				OK	OK	OK (B)
Alphanumeric	OK (C)	OK (C)	OK (C)	OK	OK	OK (B)
Group	OK (B)	OK (B)	OK (B)	OK (B)	OK (B)	OK (B)

KEY: (A) Source sign, if any, is ignored.

(B) If the source operand or the receiving operand is a group item, the move is considered to be a group MOVE.

(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; however, you should not move numeric items to alphabetic items and vice versa.

Table 12.3. Data Moves with PICTURE

SOURCE FIELD		RECEIVING FIELD		
PICTURE	Value	PICTURE	Before MOVE	After
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

MULTIPLY

Syntax in PROCEDURE DIVISION

```
MULTIPLY { data-name-1       } BY
         { numeric-literal-1 }

{ data-name-2        [GIVING data-name-3] }
{ numeric-literal-2   GIVING data-name-3  }     [ROUNDED]

[ON SIZE ERROR statement...]
```

The MULTIPLY statement multiplies two numeric values and stores the product.

Details

The result is stored in data-name-2 unless the optional GIVING clause is included. GIVING causes the result to be stored in data-name-3. If data-name-2 is replaced by a literal, the GIVING clause is required.

All of the operands (excluding literals) must be elementary numeric data items, except that the operand following GIVING may be a numeric edited item. Decimal point alignment and proper sizing of intermediate storage fields is provided automatically by the compiler, except that any intermediate result that cannot fit in 18 digits will be left-truncated.

Inclusion of the ON SIZE ERROR option makes the MULTIPLY statement conditional rather than imperative. If the integer portion of the result cannot fit in the specified receiving field, the receiving field will be unchanged and the statement(s) in the SIZE ERROR clause will be executed. If a size error occurs and no SIZE ERROR clause is present, no assumption should be made about the contents of the receiving field.

If the number of digits to the right of the decimal point in the result exceeds the number available in the result PICTURE clause, right truncation will occur unless you include the optional ROUNDED clause. If ROUNDED is specified, the right-most digit transferred to the result field will be increased by 1 whenever the most significant digit of the truncated portion is equal to or greater than 5. Negative values are affected in a similar fashion. If the result field is an integer containing one or more P editing characters, ROUNDED will add 1 to the right-most digit stored in the result field when the value masked by the left-most P is 5 or greater.

MULTIPLY

Application

Data values at start of each application:

```
05  DATA-1 PIC 99  VALUE 10.
05  DATA-2 PIC 99  VALUE 0.
```

```
MULTIPLY 2 BY DATA-1
```

Result: DATA-1 = 20

```
MULTIPLY DATA-1 BY 2 GIVING DATA-2
```

Result: DATA-2 = 20

```
MULTIPLY .05 BY DATA-1
```

Result: DATA-1 = 0

```
MULTIPLY.05 BY DATA-1 ROUNDED
```

Result: DATA-1 = 1

```
MULTIPLY 10 BY DATA-1 ON SIZE ERROR DISPLAY "UNDERFLOW".
```

Result: DATA-1 = 10, display statement executed

Syntax in DATA DIVISION

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

[ { ASCENDING  } KEY IS data-name...]
  { DESCENDING }
```

The OCCURS clause is appended to a data definition statement at any level other than 01 or 77. It is used to define related sets of data such as tables, lists, and arrays by specifying the total number of times that the same data format is repeated.

Details

The maximum legal value of the integer is 1023 and the maximum size of a table in memory is 4095 bytes. Data description clauses associated with an item whose description includes an OCCURS clause apply to each repetition of the item being described.

When you use the OCCURS clause, you must subscript or index the data-name that is the defining name of the entry whenever it appears in the PROCEDURE DIVISION (except when it is the operand of a SEARCH statement). 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.

Use of Subscripts

A subscript is written by enclosing it in parentheses after the terminal space of the associated data-name. Subscripts other than literals may be qualified.

Subscripts provide the means for identifying data items in a table, list, or array that have not been assigned individual data-names. This may be achieved through the use of simple subscripts, which indicate an occurrence number in the table, or indexes, which define a displacement from the first byte of storage allocated to the table.

A simple *subscript* is a positive nonzero integer or a data-name having such a value. The value may not be larger than the table size defined in the OCCURS clause. A subscript value of 20, for example, would reference the twentieth data item in a table, provided the table contained at least 20 items. A data-name used as a simple subscript must be of DISPLAY or COMPUTATIONAL-0 USAGE. COMPUTATIONAL-0 USAGE results in a more efficient algorithm.

OCCURS

Use of Indexes

An *index* is a data item whose size, memory location, and storage format are assigned automatically by the compiler when an INDEXED BY clause is included. You may make no other definition of the index other than inclusion of the INDEXED BY clause. The desired value of an index is achieved either by assigning it in a SET statement, or by automatic variation of the index in a SEARCH statement. An index may only be used or referred to in:

1. a SET
2. a SEARCH statement
3. a CALL statement's USING list
4. a PROCEDURE DIVISION USING list
5. a relation condition
6. the variation item in a PERFORM VARYING statement
7. a subscript.

Relative indexing may be implemented by following an index with a plus or minus sign, which is then followed by an integer. The sign must be preceded and followed by a space. The effective index is then equal to the index value plus or minus the integer value. The effective index must lie within the range of the table.

COBOL-86 tables are limited to three dimensions. Therefore, one, two, or three subscripts may be necessary, depending upon the number of OCCURS clauses that pertain to a given item. When more than one subscript is required, they are separated by commas and written in order of successively less inclusive dimensions of the table.

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

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

For an explanation of ASCENDING KEY and DESCENDING KEY, see SEARCH (Format 2) in Part III.

Application

One-dimensional table using simple subscripts:

```
DATA DIVISION.

01  ARRAY.
    05  ELEMENT OCCURS 10 TIMES PIC XXX.

77 SUBSCRIPT-STORAGE            PIC 99 COMP-0 VALUE 5.

PROCEDURE DIVISION.
PARA-1.
    MOVE ELEMENT (8) TO ELEMENT (SUBSCRIPT-STORAGE)
```

One-dimensional table using a relative index:

```
DATA DIVISION.

01  TINY-TABLE.
    05  TINY-DATA OCCURS 3 TIMES
                  INDEXED BY TINY-INDEX
                  PIC 999.

PROCEDURE DIVISION.
PARA-1.
    SET TINY-INDEX TO 1.
    MOVE TINY-DATA (TINY-INDEX) TO TINY-DATA (TINY-INDEX + 1)
                                   TINY-DATA (TINY-INDEX + 2).
```

Result: Each element is set to the value of the first element.

Two-dimensional table using indexes:

```
DATA DIVISION.

01  BIG-ARRAY.
    05  MAJOR-ELEMENT OCCURS 5 TIMES INDEXED BY MAJOR-INDEX.
        10 MINOR-ELEMENT OCCURS 8 TIMES
                         INDEXED BY MINOR-INDEX
                         PIC 999.
```

ALPHABETICAL RESERVED WORD LIST

OCCURS

```
PROCEDURE DIVISION.
PARA-1.
    SET MAJOR-INDEX TO 3.
    SET MINOR-INDEX TO 5.
    MOVE MINOR-ELEMENT (MAJOR-INDEX, MINOR-INDEX) TO
        OUTPUT-FIELD.
```

ALPHABETICAL RESERVED WORD LIST

OPEN

Syntax in PROCEDURE DIVISION

```
       { INPUT  }
OPEN   { I-O    }  filename...
       { OUTPUT }
       { EXTEND }
```

The OPEN statement makes a file available for processing.

Details

For a sequential INPUT file, entering OPEN prepares the file for reading the file's first available records into memory, so that subsequent READ statements may be executed without waiting.

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

An OPEN I-O statement is valid only for a DISK file; it permits you to use the REWRITE statement to modify records that have been accessed by a READ statement. The WRITE statement may not be used in I-O mode for files with sequential organization. The file must exist on disk when OPEN is EXECUTED; it cannot be created by OPEN I-O.

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

Attempting to READ or WRITE to a file that has not been opened is a runtime error. A file cannot be opened if it has been closed WITH LOCK during the same program run.

ALPHABETICAL RESERVED WORD LIST

OPEN

Sequential files opened for INPUT or I-O access must conform to the format described in INPUT-OUTPUT of Part III.

Application

```
OPEN INPUT  data-source-1
            data-source-2
     OUTPUT data-destination-1
            data-destination-2
     I-O    disk-filename
```

Syntax in PROCEDURE DIVISION

Format 1:

```
           ⎧ section-name                       ⎫
PERFORM    ⎨ paragraph-name                     ⎬
           ⎩ procedure-name THRU procedure-name ⎭

    [ {  integer | data-name }  TIMES]
```

The PERFORM statement, in its simplest form, permits the execution of a program module either once or the number of times specified in the optional TIMES clause.

Format 2:

```
           ⎧ section-name                       ⎫
PERFORM    ⎨ paragraph-name                     ⎬
           ⎩ procedure-name THRU procedure-name ⎭

    [VARYING  { index-name }  FROM amount-1 BY amount-2]
              { data-name  }

    UNTIL condition
```

The PERFORM/UNTIL construction causes the PERFORM statement to be executed repetitively until a specified condition is met. The condition is tested prior to each execution of PERFORM. Use of the optional VARYING clause simplifies table handling by automatically adjusting an index or subscript each time PERFORM is executed.

Format 3:

```
           ⎧ section-name                       ⎫
PERFORM    ⎨ paragraph-name                     ⎬
           ⎩ procedure-name THRU procedure-name ⎭
```

ALPHABETICAL RESERVED WORD LIST

PERFORM

```
VARYING { index-name }  FROM amount-1 BY amount-2
        { data-name  }

    UNTIL condition-1

AFTER { index-name }  FROM amount-3 BY amount-4
      { data-name  }

    UNTIL condition-2

AFTER { index-name }  FROM amount-5 BY amount-6
      { data-name  }

    UNTIL condition-3
```

A functional equivalent of the complex PERFORM statement, using VARYING, UNTIL, and AFTER clauses, is shown in the Application.

Details

The program module may be either a section, a paragraph, or (if THRU is used) any number of contiguous sections and/or paragraphs. THROUGH may be written for THRU. When the module has been executed, control returns to the next statement after the PERFORM statement.

Anytime you specify UNTIL, the condition is tested before PERFORM is executed. Therefore, if the condition is initially true, the PERFORM will be skipped during execution. Similarly, in Format 1, if data-name is less than or equal to zero, the PERFORM will be skipped.

Anytime you specify VARYING, the value following FROM is moved to the data item that follows VARYING on the first pass. On subsequent passes, the data item following VARYING is adjusted according to the value that follows BY.

In Format 2, the operands designated amount-1 and -2 may be numeric literals, index-names, or data-names. In Format 3, the operands designated identifier-1, -2, and -3 may be data-names or index-names. Those designated amount-1, -3, and -5 may be data-names, index-names, or literals. Amount-2, -4, and -6 may be data-names or literals only.

It is a runtime error to have concurrently active PERFORM ranges whose terminus points are the same.

Application

Format 1:

Example A:

```
PERFORM PARAGRAPH-ONE
```

Example B:

```
PERFORM SECTION-ONE 2 TIMES
```

Format 2:

Example A:

```
PERFORM PARAGRAPH-ONE UNTIL RESULT < 0
```

Example B:

```
PERFORM PARAGRAPH-ONE
    VARYING TABLE-INDEX FROM 1 BY 1
        UNTIL TABLE-INDEX = 100
              OR RESULT = TRUE
```

Format 3:

(This is not an application example. It is a coding analogy that will have the same functional effect as the construction shown under Format 3 in Syntax in PROCEDURE DIVISION.)

PERFORM

```
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 procedure-name THRU procedure-name
    ADD amount-6 TO identifier-3
    GO TO TEST-CONDITION-3.

END-PERFORM. Next statement.
```

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

Syntax in DATA DIVISION

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

A PICTURE clause is required to define the size and type of an elementary data item. It may also specify editing characters to be assumed in the defined field.

Details

Types of PIC

PICTURE IS is commonly shortened to the legal abbreviation PIC. There are three possible types of pictures: An-form (alphanumeric), Numeric-form, and Report-form.

Alphanumeric AN-Form Option

Valid Characters

This option applies to alphanumeric (character string) items. The PICTURE of an alphanumeric item is a combination of data description characters X, A, or 9 and, optionally, editing characters B, 0, and /. An X indicates that the character position may contain any character from the computer's ASCII character set. A PICTURE that contains at least one of the combinations:

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

in any order is considered as if every 9, A, or X character were X. The characters B, 0, and / may be used to insert blanks or zeros or slashes in the item. This is then called an *alphanumeric-edited item.*

Numeric-Form Option

Valid Characters

The PICTURE of a numeric item may contain a valid combination of the following characters:

PICTURE

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

V The optional character V indicates the position of an assumed decimal point. Since a numeric item cannot contain an actual decimal point, an assumed decimal point is used to provide the compiler with information concerning the scaling alignment of items involved in computations. Storage is never reserved for the character V. Only one V is permitted in any single PICTURE, and is redundant if it is the right-most character.

S The optional character S indicates that the item has an operational sign. It must be the first character of the PICTURE. See SIGN in Part III.

P The character P indicates an assumed decimal scaling position. It is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character P is not counted in the size of the data item; that is, memory is not reserved for these positions. However, scaling position characters are counted in determining the maximum number of digit positions (18) in numeric-edited items or in items that appear as operands in arithmetic statements. The scaling position character P may appear only to the left or right of the other characters in the string as a continuous string of P's within a PICTURE description.

The sign character S and the assumed decimal point V are the only characters that 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.

Report-Form Option

Valid Characters

This option describes a data item suitable as an "edited" receiving field for presentation of a numeric value. The editing characters that may be combined to describe a report item are as follows:

9 V . Z CR DB , $ + * B O – P /

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

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

Z, * — The characters Z and * are called *replacement characters*. Each one represents a digit position. During execution, leading zeros to be placed in positions defined by Z or * are suppressed, becoming blank or *. Zero suppression terminates at the decimal point (. or V) or the first 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.

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

PICTURE

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. It must not be the last character in the PICTURE character string.

Floating String

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

$$,$$$,$$$
++++
−−,−−−,−−
+(8).++
$$,$$$.$$

A floating string containing N + 1 occurrences of $ or + or − defines N digit positions. When a numeric value is moved into a report item, the appropriate character floats from left to right, so that the developed report item has exactly one actual $ or + or − immediately to the left of the most significant nonzero digit, in one of the positions indicated by $ or + or − in the PICTURE. Blanks are placed in all character positions to the left of the single developed $ or + or − (see Table 12.4). If the most significant digit appears in a position to the right of positions defined by the floating string, then the developed item contains $ or + or − in the 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 were indicated by the PICTURE character 9. In the following examples, b represents a blank in the developed items.

Table 12.4. A Floating String Example

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

A floating string need not constitute the entire PICTURE of a report item, as shown in the preceding examples. Restrictions on characters that may follow a floating string are given later in the description.

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

+ # − The character + or − may appear in a PICTURE either singly or in a floating string. As a fixed sign control character, the + or = must appear as the last symbol in the PICTURE. The plus sign indicates that the sign of the item is indicated by either a plus or minus placed in the character position, depending on the algebraic sign of the numeric value placed in the report field. The minus sign indicates that blank or minus is placed in the character position, depending on whether the algebraic sign of the numeric value placed in the report field is positive or negative, respectively.

B Each appearance of B in a PICTURE represents a blank in the final edited value.

/ Each slash in a PICTURE represents a slash in the final edited value.

0 Each appearance of 0 in a PICTURE represents a position in the final edited value where the digit zero will appear.

Other Rules for a Report-Form PICTURE

1. The appearance of one type of floating string precludes any other floating string.

2. There must be at least one digit position character.

3. The appearance of a floating sign string or fixed plus or minus insertion character precludes the appearance of any other of the sign control insertion characters, namely, +, −, CR, DB.

PICTURE

4. The characters to the right of a decimal point up to the end of a PICTURE, excluding the fixed insertion characters +, −, CR, DB (if present), are subject to the following restrictions:

   a. Only one type of digit position character may appear. That is, Z * 9 and floating-string digit position characters $ + − are all 6, mutually exclusive.

   b. If one of the numeric character positions to the right of a decimal point is represented by + or − or $ or Z, then all the numeric character positions in the PICTURE must be represented by the same character.

5. The PICTURE character 9 can never appear to the left of a floating string or replacement character.

Additional Notes on the PICTURE Clause

1. A PICTURE clause must only be used at the elementary level.

2. An integer enclosed in parentheses and following X 9 $ Z P * B − or + indicates the number of consecutive occurrences of the PICTURE character.

3. Characters V and P are not counted in the space allocation of a data item. CR and DB occupy two character positions.

4. A maximum of 30 character positions is allowed in a PICTURE character string. For example, PICTURE X(89) consists of five PICTURE characters.

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

6. The characters '.' S V CR and DB can appear only once in a PICTURE.

7. When DECIMAL-POINT IS COMMA is specified, the explanations for period and comma are understood to apply to comma and period, respectively.

PICTURE

Table 12.5. Editing DATA with PICTURE

SOURCE DATA		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

Application

The examples in Table 12.5 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.)

ALPHABETICAL RESERVED WORD LIST

PROCEDURE

Syntax in Division Header

```
PROCEDURE DIVISION

[ {USING [data-name-1...] | CHAINING data-name-1...} ].

[section-name SECTION.

paragraph-name...]...
```

The PROCEDURE DIVISION contains the executable portion of a program.

Details

The PROCEDURE DIVISION is required and must follow the DATA DIVISION in every COBOL program. The optional USING list may be included only if the program is a subprogram. The CHAINING list is used only if the program is invoked by a CHAIN statement in another program and parameters are to be passed (see CALL, CHAIN). The use of sections you define is treated in Part II, Chapter 9.

READ (to Perform Sequential Input)

Syntax in PROCEDURE DIVISION

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

The READ statement makes available the next logical record of the specified file. It also updates the FILE STATUS data item, if you have specified one. READ for relative and indexed files is discussed in Chapters 10 and 11.

Details

The imperative statement(s) in the optional AT END clause specifies the action to be taken if an end-of-file indication is received from the assigned storage device. If an end of file occurs and no AT END procedure is specified, an applicable declarative procedure will be performed. If you specify neither an imperative statement nor a DECLARATIVES procedure, and no FILE STATUS data item has been established, a runtime error will occur. If a FILE STATUS item has been established, execution will proceed normally on the assumption that the file status will be checked to determine a course of action.

When INTO is included, the record will be read into the appropriate file description and immediately copied into the area designated by the data-name. You should not specify INTO if the file includes variable-size records. Any subscript or index in the data-name is evaluated after the record is read but before it is moved to data-name.

Right truncation or left justification with space padding will occur if the length of a record differs from the length of its associated file description.

Application

```
READ INPUT-FILE INTO WS-IMAGE
    AT END MOVE "Y" TO EOF-SWITCH
           GO TO EXIT-PROCEDURE.
```

ALPHABETICAL RESERVED WORD LIST

RECORD

Syntax in DATA DIVISION

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

This clause is appended to an FD paragraph and specifies the size of the logical record in characters.

Details

Since the actual size of a record is determined by the PICTURE clauses contained in the group, this clause is always documentary in nature.

Integer-2 contains the record size if all records are of the same size (integer-1 is omitted). If variable size records are used, the size of the smallest and the largest should be placed in integer-1 and integer-2, respectively.

Application

```
FD  INPUT-FILE LABEL RECORDS ARE STANDARD
               RECORD CONTAINS 80 CHARACTERS.
```

REDEFINES

Syntax in DATA DIVISION

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

The REDEFINES clause is appended to a data definition statement and either specifies that the same memory area is to contain different data items or provides an alternative grouping or description of the same data.

Details

When written, the REDEFINES clause should be the first clause following the data-name (data-name-1) 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 an area remain in effect. Thus, if B and C are two separate items that share the same storage area due to *redefinition*, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same memory area would receive Y according to the description of C.

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

1. s must equal t, but must not equal 88.

2. The object must be contained in the same record (01 level item), unless s = t = 01.

3. Prior to definition of the *subject* and subsequent to definition of the *object*, there can be no level numbers that are numerically less than s.

ALPHABETICAL RESERVED WORD LIST

REDEFINES

The length of data-name-1 (multiplied by the operand of any OCCURS clause that may be present) may not exceed the length of data-name-2, unless the level of data-name-1 is 01 (permitted only outside the FILE SECTION). Data-name-1 and entries subordinate to it must not contain any VALUE clauses, except level 88 conditions.

In the FILE SECTION, multiple level 01 entries subordinate to any given FD clause represent implicit redefinitions of the same area and do not require a REDEFINES clause.

Application

```
WORKING-STORAGE SECTION.

01  FIRST-GROUP-ITEM.
    05  ALPHA-VAL-1        PIC XXX.
    05  ALPHA-VAL-2        PIC XXX.

01  SECOND-GROUP-ITEM REDEFINES FIRST-GROUP-ITEM.
    05  NUMERIC-VAL        PIC 9(6).

01  ANOTHER-GROUP.
    05  SUBGROUP-1.
        10  ALPHA-VAL-3    PIC XXX.
        10  ALPHA-VAL-4    PIC XXX.
    05  SUBGROUP-2 REDEFINES SUBGROUP-1.
        10  ALPHA-VAL-6    PIC X(6).
```

REWRITE (to Perform Sequential I/O)

Syntax in PROCEDURE DIVISION

```
REWRITE record-name [FROM data-name]
```

The REWRITE statement replaces the most recently READ record in a sequential disk file. REWRITE for indexed and relative files is discussed in Chapters 10 and 11.

Details

Record-name is the name of a logical record defined in the FILE SECTION of the DATA DIVISION and may be qualified. Record-name and data-name must refer to separate storage areas. If the FROM clause is included, the effect is as if the statement `MOVE data-name TO record-name` were executed just prior to the REWRITE.

The file containing record-name must be OPEN in the I/O mode at the time REWRITE is executed. The READ executed prior to the REWRITE must be completed successfully. If the record is lengthened during processing, it will be truncated to the size of the original record when REWRITE is executed. If the record is shortened during processing, unpredictable information will be stored between the end of the logical record and the end of the physical record.

Application

```
READ employee-file INTO employee-detail-line.
PERFORM modify-employee-record.
REWRITE employee-record FROM employee-detail-line.
```

ALPHABETICAL RESERVED WORD LIST

SCREEN

Syntax in DATA DIVISION

```
SCREEN SECTION.

level-number {screen-name | screen-item}.

[BLANK SCREEN]

[LINE NUMBER IS [PLUS] integer-1]

[COLUMN NUMBER IS [PLUS] integer-2]

[BLANK LINE]

[BELL]

[REVERSE-VIDEO]

[FOREGROUND-COLOR integer-3]

[BACKGROUND-COLOR integer-4]

[BLANK WHEN ZERO]

[JUSTIFIED RIGHT]

[AUTO]

[SECURE]

{[VALUE IS] literal-1} | {PICTURE IS picture-string

   {USING identifier-3} | {[FROM] data-name-1] [TO data-name-2]}}
                                  literal-2
```

The SCREEN SECTION contains data definitions for formatted screens. If you include it, it must be the last section in the DATA DIVISION.

Details

Use of Levels

As in the FILE and WORKING-STORAGE sections, descriptions may be grouped through the assignment of appropriate level numbers. Thus there are two types of screen items. 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 that you ACCEPT or DISPLAY by executing a single PROCEDURE DIVISION statement.

A 01 level entry must be followed by a screen-name, and an entry at level 02-49 must be followed by the name of a group or elementary screen-item. Each group item must be followed by one or more elementary items, indicated by the assignment of a higher level number.

All optional clauses are applicable at the elementary level. Only the options AUTO and SECURE may be included at a group level. If used, the effect is as if AUTO or SECURE had been specified for each of the elementary items within the group. AUTO and SECURE may only be included if each elementary item affected contains a PICTURE clause. If you specify PICTURE, then you must include either USING or at least one FROM or TO. The order in which optional entries are coded is not significant.

Effects of Optional Clauses

Each optional clause has a specific effect on data input and data display operations when ACCEPT and DISPLAY statements are executed at runtime. The effects of each specification are as follows:

1. BLANK SCREEN causes the entire screen to be erased and the cursor to be placed at the home position (line 1, column 1).

2. LINE and COLUMN affect the screen location associated with an elementary screen item. As the SCREEN SECTION is processed at compiletime, 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 CRT screen.

ALPHABETICAL RESERVED WORD LIST

SCREEN

You may change the position current at the start of any elementary screen data description by using the LINE and COLUMN specifications. If you code neither the LINE nor COLUMN, the current screen position is not changed. If you code the COLUMN without LINE, the current screen line is not adjusted. If you code the LINE 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; the result is 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.

3. BLANK LINE causes erasure of the screen from the current cursor position to the end of the current line.

4. BELL will sound the terminal's audio tone when a program is ready to ACCEPT keyboard input.

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

6. FOREGROUND-COLOR controls the color of the characters when a color video display terminal is installed. The color is chosen by the value of integer-1, which must be in the range 0-7. White is the default. Color definitions are as follows:

0	black	4	red
1	blue	5	magenta
2	green	6	yellow
3	cyan	7	white

7. BACKGROUND-COLOR controls the color of the background. Integer-2 must contain a value taken from the table above. Black is the default.

8. BLANK WHEN ZERO causes a screen item to be displayed as spaces if its value is zero.

9. JUSTIFIED and JUST specify that operator-keyed data or data from a FROM field, USING field, or literal will be aligned with the right boundary of the screen item when the data are displayed on the screen.

10. VALUE IS literal explicitly specifies the character string that should be displayed on the screen when the screen item being defined is referenced by a DISPLAY statement. A screen item for which VALUE is specified is ignored by all ACCEPT statements.

11. PICTURE specifies the format in which data are to be presented on the screen. It is coded according to the rules for WORKING-STORAGE PICTURE clauses (see PICTURE). During a DISPLAY statement, the contents of a FROM or USING field are moved to an implicit temporary item with the specified PICTURE before the field contents are displayed on the screen. During an ACCEPT statement, the displayed contents of the field being entered are punctuated to conform with the given PICTURE format.

12. FROM, TO, and USING describe relationships between a screen item and literals and/or fields in the FILE, WORKING-STORAGE, and/or LINKAGE sections. On DISPLAY of a screen item, 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 exhibited on the screen. On an ACCEPT of the screen item, the runtime system implicitly moves the accepted data to any TO or USING field you have specified for the item.

13. AUTO specifies that when you have filled a field with input, the cursor automatically skips to the next input field, rather than waiting for you to type a terminator character. If there are no more input fields remaining, the ACCEPT is terminated.

14. SECURE suppresses the echoing of input characters. Instead, an asterisk is displayed for each data character accepted.

ALPHABETICAL RESERVED WORD LIST

SCREEN

NOTE: The following functions are always executed in the order shown below, regardless of the order in which you specify them:

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

Syntax in PROCEDURE DIVISION

Format 1:

SEARCH table-name [VARYING index-data-name | index-name]

[AT END imperative-statement-1]

{WHEN condition-1 {NEXT SENTENCE imperative-statement-2} }...

The Format 1 SEARCH statement is used to perform a linear search of a table by automatically varying its index until a specified condition is met.

Format 2:

SEARCH ALL table-name [AT END imperative-statement-1...]

WHEN condition-1 {NEXT SENTENCE imperative-statement-2...}

The Format 2 SEARCH statement is used to perform a nonserial search of a table of ordered data.

Details

Format 1

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

Use of the optional VARYING phrase permits an additional index or index-data-name to be varied simultaneously with the index associated with table-name. This option is most often used to correlate related positions in separate tables.

SEARCH

Rules of Index Values

The following rules apply to values of index as you SEARCH:

1. The initial value must be preset by a SET statement before a SEARCH is performed.

2. If the initial value of an index exceeds the maximum declared in the OCCURS clause, the SEARCH will terminate immediately and the AT END clause, if present, will be executed.

3. If the value of the index falls within the range of legal values, then each WHEN clause is evaluated until one is found to be true or all are found to be false. That is, the WHEN clauses are considered to be connected in an OR relationship. If one is true, its associated imperative statement is executed and the SEARCH operation terminates. If none is true, the index (plus any items coded with VARYING) is incremented by one and the SEARCH is repeated.

If the table is subordinate to another table (a dimension of a multidimensional table), an index must be specified in an INDEXED BY phrase for each OCCURS clause. However, since the SEARCH statement varies only a single index (plus any items coded with VARYING), searching an entire multidimensional table requires coding multiple SEARCH statements. The indexes not being varied are preset before each dimension is searched, often by using a PERFORM VARYING construction.

Format 2

Rules of WHEN Condition

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

1. Only simple relational conditions or condition-names may be employed, and the subject must be properly indexed by the index associated with the table (or table dimension) being searched. Furthermore, each subject data-name (or the data-name associated with the condition-name) in the condition must be mentioned in the KEY clause of the table. The KEY clause is an appendage to the OCCURS clause of the format

```
{ASCENDING | DESCENDING} KEY IS data-name...
```

where data-name is an item defined at a higher level number within the same group item. The KEY phrase indicates that the repeated data are arranged in ascending or descending order according to the data-names that 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 IS EQUAL TO or =) is permitted.

3. Any condition-name (level 88 item) must be defined as having only a single value.

4. The condition may be compounded by use of the logical operator AND, but not OR.

5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or a data item. A data item must not be listed in the KEY clause of the table or be indexed by the index associated with table-name.

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

In a Format 2 SEARCH, a nonserial type of search operation takes place, which relies upon the declared ordering of data. The initial setting of the index for table-name is ignored and its value is varied automatically within the bounds of the table as the search progresses. If the WHEN condition cannot be satisfied for any valid index value, the AT END clause, if present, will be executed. If no AT END clause is specified, the next executable sentence receives control.

If all the simple conditions in the single WHEN condition are satisfied, the resultant index value points to a table item that causes those conditions to be true. If no such table item is found, the final index value is unpredictable.

ALPHABETICAL RESERVED WORD LIST

SEARCH

Application

Format 1:

```
WORKING-STORAGE SECTION.

01  WATER-TABLE.
    05  WATER-RATE-TABLE OCCURS 40 TIMES INDEXED BY RATE-INDEX.
        10  USAGE-LIMIT          PIC 9(6).
        10  MONTHLY-CHARGE       PIC 9(3)V99.

PROCEDURE DIVISION.

SEARCH-ROUTINE.
    SET RATE-INDEX TO 1.
    SEARCH WATER-RATE-TABLE AT END PERFORM TABLE-ERROR-ROUTINE
        WHEN USAGE-LIMIT (RATE-INDEX) > USAGE-IN
            MOVE MONTHLY-CHARGE (RATE-INDEX) TO USER-CHARGE.
```

Format 2:

```
WORKING-STORAGE SECTION.

01  QUANTITY-TABLE.
    05  QUANTITY-DISCOUNT-TABLE OCCURS 100 TIMES
                                INDEXED BY DISCOUNT-INDEX
                                ASCENDING KEY PURCHASE-QUANTITY.
        10  PURCHASE-QUANTITY       PIC 999.
        10  DISCOUNT-FACTOR         PIC V99.

PROCEDURE DIVISION.

SEARCH ROUTINE.
    SET DISCOUNT-INDEX TO 1.
    SEARCH ALL QUANTITY-DISCOUNT-TABLE
        AT END MOVE MAX-DISCOUNT TO USER-DISCOUNT
        WHEN PURCHASE-QUANTITY (DISCOUNT-INDEX) = QUANTITY-IN
            MOVE DISCOUNT-FACTOR (DISCOUNT-INDEX) TO USER-DISCOUNT.
```

SET

Syntax in PROCEDURE DIVISION

Format 1:

```
     ⎧index-1      ⎫          ⎧index-2      ⎫
SET  ⎨index-item-1 ⎬ ... TO   ⎨index-item-2 ⎬
     ⎩data-name-1  ⎭          ⎨data-name-2  ⎬
                              ⎩integer      ⎭
```

Format 2:

```
SET index... {UP BY | DOWN BY} {data-name | integer}
```

The SET statement permits the manipulation of indexes, index data items, or binary subscripts for table-handling purposes.

Details

Because they are stored in a special internal format, use of the MOVE, ADD, and SUBTRACT statements is not permitted with either indexes or index data items. Format 1 of the SET statement provides the effect of a MOVE, and Format 2 provides the effect of an ADD or SUBTRACT. Operands of the SET statement are limited to positive integer values.

Application

```
SET TABLE-1-INDEX TO 1

SET TABLE-1-INDEX TO INDEX-ITEM-1

SET TABLE-1-INDEX INDEX-ITEM-1 UP BY 5
```

SIGN

Syntax in DATA DIVISION

```
SIGN IS {LEADING | TRAILING} [SEPARATE CHARACTER]
```

The SIGN clause is appended to a data definition statement whose USAGE IS DISPLAY. It determines the method of internal sign representation.

Details

The following summarizes the effect of the four possible forms of this clause:

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

When these forms are coded, the PICTURE must begin with an S. If no S appears, the item is unsigned and the SIGN clause is prohibited. When S appears at the front of a PICTURE but no SIGN clause is included in the definition, the default case SIGN IS TRAILING is assumed.

The SIGN clause may be written at a group level. Such a clause specifies the sign's format for any signed DISPLAY item within the group.

The SEPARATE CHARACTER phrase increases the size of the data item by one character.

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

Application

```
77 NUMERIC-VAL     PIC S999 SIGN IS TRAILING SEPARATE.
```

STOP

Syntax in PROCEDURE DIVISION

```
STOP {RUN | literal}.
```

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

Details

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

When followed by a literal, the STOP statement displays the specified literal on the screen and suspends execution. Execution resumes when you press RETURN. Generally you use the pause to perform a function suggested by the literal.

Application

```
STOP "Place data disk #2 in drive B and press RETURN."

STOP "Press RETURN when printer is ready."

STOP RUN.
```

ALPHABETICAL RESERVED WORD LIST

STRING

Syntax in PROCEDURE DIVISION

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

INTO identifier-1 [WITH POINTER identifier-2]

[ON OVERFLOW imperative-statement]
```

The STRING statement allows concatenation of one or more alphabetic or alphanumeric data items into a single receiving data item.

Details

Operand-1 and -2 may be data-names, non-numeric literals, or single-character figurative constants. Identifier-1 is the receiving field. It may not contain editing symbols or a JUSTIFIED clause. Identifier-2 is a position counter. It must be an elementary numeric integer data item of a size that allows storage of a value 1 greater than the length of identifier-1.

If the POINTER phrase is omitted, the internal pointer defaults to a value of 1. This causes concatenation to occur at the beginning of the string found in identifier-1. If the POINTER phrase is included, concatenation occurs at the position that corresponds to the value of identifier-2.

Termination of the character transfer process is controlled by the DELIMITED clause. DELIMITED BY SIZE causes the entire sending field to be moved. DELIMITED BY operand-2 terminates the move when the character string specified by operand-2 is found in operand-1. The matching characters are not transferred.

The imperative-statement will be executed anytime the position pointer assumes a value less than 1 or greater than the length of identifier-1. If no imperative-statement is provided, control is passed to the next executable statement.

There is no automatic space fill into any position of identifier-1. Unaccessed positions are unchanged at the completion of the STRING statement. The value of the pointer (identifier-2) is incremented by 1 for each character moved. Its value at completion is equal to its starting value plus the number of characters moved.

Application

Data values at start of each application:

```
05  MESSAGE-FIELD-1  PIC X(21) VALUE "ACCOUNT BALANCE DOES NOT ".
05  MESSAGE-FIELD-2  PIC X(5)  VALUE "MEETS".
05  MESSAGE-FIELD-3  PIC X(13) VALUE " REQUIREMENTS.".

05  DELIMITER-1      PIC X(4)  VALUE "DOES".

05  DISPLAY-MESSAGE  PIC X(43) VALUE SPACES.

STRING MESSAGE-FIELD-1 DELIMITED BY SIZE
       MESSAGE-FIELD-2 DELIMITED BY "S"
       MESSAGE-FIELD-3 DELIMITED BY SIZE
           INTO DISPLAY-MESSAGE.
```

Result:

```
DISPLAY-MESSAGE = "ACCOUNT BALANCE DOES NOT MEET REQUIREMENTS."

STRING MESSAGE-FIELD-1 DELIMITED BY DELIMITER-1
       MESSAGE-FIELD-2 DELIMITED BY SIZE
       MESSAGE-FIELD-3 DELIMITED BY SIZE.
```

Result:

```
DISPLAY-MESSAGE = "ACCOUNT BALANCE MEETS REQUIREMENTS.
```

ALPHABETICAL RESERVED WORD LIST

SUBTRACT

Syntax in PROCEDURE DIVISION

```
SUBTRACT  { data-name-1...       }  FROM
          { numeric-literal-1... }

{ data-name-m         [GIVING] data-name-n] }   [ROUNDED]
{ numeric-literal-m   GIVING data-name-n    }

[ON SIZE ERROR statement...]
```

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

Details

All of the values that precede the word FROM are subtracted from the value that follows FROM. The result is stored in the data item that follows FROM unless the optional GIVING clause is included. GIVING causes the result to be stored in data-name-n. If the value that follows FROM is a numeric literal, the GIVING clause is required.

All of the operands (excluding literals) must be elementary numeric data items, except that the operand following GIVING may be a numeric edited item. Decimal point alignment and proper sizing of intermediate storage fields is provided automatically by the compiler, except that any intermediate result that cannot fit in 18 digits will be left-truncated.

Inclusion of the ON SIZE ERROR option makes the SUBTRACT statement conditional rather than imperative. If the integer portion of the result cannot fit in the specified receiving field, the receiving field will be unchanged and the statement(s) in the SIZE ERROR clause will be executed. If a size error occurs and no SIZE ERROR clause is present, no assumption should be made about the contents of the receiving field.

If the number of digits to the right of the decimal point in the result exceeds the number available in the result PICTURE clause, right truncation will occur unless you include the optional ROUNDED clause. If ROUNDED is specified, the right-most digit transferred to the result field will be increased by 1 whenever the most significant digit of the truncated portion is equal to or greater than 5. Negative values are affected in a similar fashion.

SUBTRACT

If the result field is an integer containing one or more P editing characters, ROUNDED will add 1 to the right-most digit stored in the result field when the value masked by the left-most P is 5 or greater.

Application

Data values at start of each application:

```
05  DATA-1 PIC 99  VALUE 99.
05  DATA-2 PIC 999 VALUE 100.
05  DATA-3 PIC V9  VALUE .5.
```

```
SUBTRACT 1 FROM DATA-2
```

Result: DATA-2 = 99

```
SUBTRACT 1 FROM DATA-2 GIVING DATA-1
```

Result: DATA-1 = 99

```
SUBTRACT DATA-3 FROM DATA-2
```

Result: DATA-2 = 99

```
SUBTRACT DATA-3 FROM DATA-2 ROUNDED
```

Result: DATA-2 = 100

```
SUBTRACT DATA-2 FROM DATA-1 ON SIZE ERROR DISPLAY "UNDERFLOW".
```

Result: DATA-1 = 99, display statement executed

ALPHABETICAL RESERVED WORD LIST

SYNCHRONIZED

Syntax in DATA DIVISION

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

The SYNCHRONIZED clause is appended to a data definition statement and indicates that memory allocation for the associated item should begin or end on a word boundary.

Details

In COBOL-86, this statement is checked for correct syntax, but is treated only as commentary. SYNC is an acceptable abbreviation of SYNCHRONIZED.

Application

```
77  BIN-VAL    PIC 9999 COMP SYNC.
```

TRACE

Syntax in PROCEDURE DIVISION

{READY | RESET} TRACE

TRACE is a debugging function that causes the name of each section or procedure to be displayed on the screen each time it is executed.

Details

The TRACE function is activated when the READY TRACE statement is encountered in the source code. It is suspended when the RESET TRACE statement is encountered. If these statements are written on a line that includes a D in column 7, they will be ignored unless you specify WITH DEBUGGING MODE in the SOURCE-COMPUTER paragraph in the CONFIGURATION SECTION. By comparing the ordered list of section and procedure names with the performance of the program being debugged, you can easily detect the point at which the intended program flow departed from the actual program flow.

Application

Source code:

```
INITIALIZE.
    READY TRACE.

MAIN.
    MOVE 0 TO VAL.
    PERFORM INCREMENT UNTIL VAL=5.
    STOP RUN.

INCREMENT.
    ADD 1 TO VAL.
    DISPLAY VAL.
    IF VAL > 3 RESET TRACE.
```

Screen output:

```
MAIN
1
INCREMENT
2
INCREMENT
3
INCREMENT
4
5
```

ALPHABETICAL RESERVED WORD LIST

UNSTRING

Syntax in PROCEDURE DIVISION

```
UNSTRING data-name-1

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

INTO {data-name-2

[DELIMITER IN data-name-3] [COUNT INdata-name-4]}...

[WITH POINTER data-name-5] [TALLYING IN data-name-6]

[ON OVERFLOW imperative-statement]
```

The UNSTRING statement causes characters in a single alphanumeric sending field to be separated into subfields and moved to individual receiving fields.

Details

Data-name-1 and data-name-3 must be alphanumeric group or elementary data items. Operands must be non-numeric literals, single-character figurative constants, or alphanumeric data items. Data-name-2 must be an alphabetic or alphanumeric group or elementary item, or an elementary numeric item with USAGE DISPLAY and no P characters in its PICTURE. Data-name-4, -5, and -6 must be elementary numeric items.

During execution, one or more variable length character strings is developed, as determined by the length of the current receiving field or the effect of delimiting operands. When the string is completely developed, it is transferred to the receiving field in standard MOVE fashion.

If you include the DELIMITED BY clause, each operand specified is compared in order to the characters being moved from data-name-1. When a match is found, the move into the current data-name-2 is terminated and any remaining positions are either zero- or space-filled, as appropriate. The delimiter itself is not transferred. The UNSTRING function then continues, utilizing each subsequent data-name-2 until the end of data-name-1 is reached.

UNSTRING

When you specify the ALL phrase, multiple contiguous occurrences of the associated delimiter will be treated as a single occurrence. If, at any time, two contiguous delimiters are found, the current receiving field will be entirely zero- or space-filled, depending on its type.

If you include the DELIMITED BY clause, you may also include the DELIMITER IN phrase after any data-name-2. The character(s) used to delimit the associated data-name-2 will then be stored in data-name-3. If you also include the COUNT IN phrase, it will contain the number of characters moved to data-name-2.

The value found in data-name-5 determines the character position in data-name-1 where the UNSTRING function will begin. If you do not include a POINTER phrase, an internal pointer is maintained that has a starting value of 1. When execution of the statement is completed, identifier-5 will contain the final value of the POINTER.

If the POINTER should assume a value less than 1 or greater than the length of data-name-1, an overflow condition exists and control passes to the imperative statement(s) in the ON OVERFLOW clause, if you include one. An overflow condition also exists if all of the receiving fields (data-name-2) are exhausted prior to scanning the entire length of the sending field (data-name-1).

If there is a TALLYING IN phrase, 1 will be added to data-name-6 for each receiving field (data-name-2) acted upon. Thus, the final value of data-name-6 will be its starting value plus the number of receiving fields acted upon during execution of the statement.

Any subscript or index of data-name-1, -5, or -6 is evaluated only once at the beginning of the UNSTRING statement. Any subscript associated with the operands or with data-name-2, -3, or -4 is evaluated immediately before access to that item.

ALPHABETICAL RESERVED WORD LIST

UNSTRING

Application

Data values at start of each application:

```
77  STRING1   PIC X(13) VALUE "ABCDEFGGHIJKL".
77  DEST1     PIC X(5).
77  DEST2     PIC X(5).
77  DEST3     PIC X(5).
77  DEST4     PIC X(5).
77  PTR-VAL   PIC 99    VALUE 8.
```

```
UNSTRING STRING1 INTO DEST1 DEST2 DEST3
```

Result: DEST1 = "ABCDE", DEST2 = "FGGHI", DEST3 = "JKL ".

```
UNSTRING STRING1 DELIMITED BY "G" INTO DEST1 DEST2 DEST3 DEST4
```

Result: DEST1 = "ABCDE", DEST2 = "F   ", DEST3 = "   ", DEST4 = "HIJKL".

```
UNSTRING STRING1
    DELIMITED BY ALL "G" INTO DEST1 DEST2 DEST3
```

Result: DEST1 = "ABCDE", DEST2 = "F   ", DEST3 = "HIJKL".

```
UNSTRING1 STRING1 INTO DEST1 DEST2 WITH POINTER PTR-VAL
```

Result: DEST1 = "GHIJK", DEST2 = "L   ", PTR-VAL = 13.

USAGE

Syntax in DATA DIVISION

```
USAGE IS  { DISPLAY          }
          { COMPUTATIONAL-0  }
          { COMPUTATIONAL-3  }
          { INDEX            }
```

The USAGE clause is written as part of the definition of a group or elementary data item. Its function is to specify the form in which numeric data are represented internally.

Details

The USAGE clause is optional and may be written at any level. When included at the group level, the specified USAGE applies to all subordinate items. When written at the elementary level, the USAGE clause must not contradict any USAGE specified for the group.

If the clause is omitted, USAGE IS DISPLAY is assumed. DISPLAY USAGE allocates one byte of storage for each digit defined in a picture clause. The sign, if present, is normally embedded in the lowest order byte. Values are limited to a maximum of 18 digits.

COMPUTATIONAL-0 USAGE allocates a single 16-bit word for storage of a value as a true binary number. The highest order bit represents the sign. COMP-0 is a valid abbreviation.

COMPUTATIONAL-3 USAGE allocates a half-byte for each digit plus a half-byte for the sign. Each digit is stored as a binary number in the range 0-9. COMP-3 is a valid abbreviation.

Restrictions for Index Data Item

INDEX USAGE provides a means to store the value of a table index in a data item that is not itself an index. A data item containing a USAGE IS INDEX phrase is referred to as an *index data item*. Such an item has two restrictions applicable to it that do not affect other storage formats:

USAGE

- An index data item may not have a PICTURE clause. Its size and structure are determined automatically by the compiler.
- An index data item may be used only in a SET or SEARCH statement, the USING list of a CALL statement or a PROCEDURE DIVISION header, a relation test, or as the variation item in a PERFORM VARYING statement.

For a complete understanding of indexes and index data items, see OCCURS, SEARCH, and SET. Also see Numeric Items in Chapter 8 for a further discussion of numeric values in COBOL.

Application

```
01  WS-ACCUMULATORS COMP-3.
    05  PAGE-COUNT PIC S9(5).
    05  LINES-USED PIC S9(3).

77  RECORDS-READ PIC S9(4) COMP-0.

01  WS-INDEX-ITEMS.
    05  TABLE-1-POINTER INDEX.
    05  TABLE-2-POINTER INDEX.
```

VALUE (to Define Truth Set of Condition-name)

Syntax in DATA DIVISION

Format 1:

```
VALUE IS literal-1 [literal-2...]
```

Format 2:

```
VALUES ARE literal-1 THRU literal-2
```

A VALUE clause is required in a level 88 condition entry to specify the value(s) or value range that results in the condition-name being true.

Details

A level 88 entry must be preceded by either another level 88 entry (as in the case of multiple condition-names associated with a single data item) or by an elementary data item (which may be FILLER). INDEX data items may not be followed by level 88 items.

A nonunique condition-name may be qualified by the name of the elementary item associated with it and by the elementary item's qualifiers (see OF, IN).

When used in the PROCEDURE DIVISION, a condition-name replaces a simple relational condition. A condition-name may be associated with an elementary item requiring subscripts. In this case, the condition-name, when written in the PROCEDURE DIVISION, must be subscripted according to the same requirements as the associated elementary item.

The type and size of the literal that follows VALUE IS must be consistent with the PIC clause of the associated elementary item. If the PIC clause contains editing characters, the literal must be non-numeric. A VALUE clause may contain either a series or a range of literals, but not both.

VALUE (to Define Truth Set of Condition-name)

Application

Used with VALUE range:

```
05  FILLER PIC 99.
    88  UNDER-AGE VALUES ARE 0 THRU 20.
```

Used with VALUE series:

```
05  FILLER PIC X(5).
    88  LEGAL-HOLIDAY VALUE "1/1  " "7/4  " "12/25".
```

Multiple conditions used with an edited field:

```
05  FILLER PIC Z,ZZ9.
    88  MAXIMUM-LEVEL VALUE "8,750".
    88  AVERAGE-LEVEL VALUE "4,320".
    88  MINIMUM-LEVEL VALUE "  120".
```

Qualified condition-name:

```
      In DATA DIVISION:

05  SYSTEM-PRESSURE PIC 999.
    88  INSUFFICIENT VALUES ARE 0 THRU 124.

      In PROCEDURE DIVISION reference:

IF INSUFFICIENT IN SYSTEM-PRESSURE
```

VALUE (to Initialize Data Value)

Syntax in DATA DIVISION

Format 1:

```
VALUE IS "ABCD"
```

Format 2:

```
VALUE IS -1234
```

A VALUE clause specifies the initial value of a data field defined in the Working-Storage Section.

Details

Use of a VALUE clause is optional. If it is omitted, no assumption should be made about the initial contents of a data item. A VALUE clause may not be used outside the Working-Storage Section, nor may it be used with a data item that includes or is subordinate to an OCCURS or REDEFINES clause.

The size of the specified VALUE must fit in the field size defined by the PICTURE clause. If the PIC clause defines a larger field, an alphanumeric value will be left-justified with space fill and a numeric value will be decimal point aligned with zero padding. A figurative constant may replace a literal value.

A VALUE clause ignores editing characters in the PIC definition, as well as the BLANK WHEN ZERO and JUSTIFIED RIGHT clauses. However, an alphanumeric edited PICTURE clause can properly align an alphanumeric VALUE entry if the VALUE entry includes the editing characters (see the Application).

VALUE may be used at the group level provided the group does not contain a PICTURE clause with any 9 characters. You may specify either a non-numeric figurative constant or a non-numeric literal of a size not greater than the size of the group. Do not use this option if the reserved words JUSTIFIED, SYNCHRONIZED, or USAGE (other than DISPLAY) occur anywhere within the group. If you specify a VALUE at a group level, it may not be specified at a higher level number.

VALUE (to Initialize Data Value)

Application

Use of figurative constant:

```
PIC X(5) VALUE SPACES
```

```
PIC 9(5) VALUE ZERO
```

Left justification with space fill:

```
PIC X(5) VALUE "YES"
```

```
PIC X(5) VALUE "YES" JUSTIFIED RIGHT
```

Decimal point alignment with zero padding:

```
PIC 99V99 VALUE 1.2
```

Only legal method to accommodate edited field:

```
PIC XX/XX/XX VALUE "10/20/48"
```

Illegal, PIC too small:

```
PIC 99V99 VALUE 100.2
```

```
PIC X(5) VALUE "RESULT"
```

Use at group level:

```
01 WS-SWITCHES VALUE " NO NO".
    05  ERROR-SWITCH PIC XXX.
    05  EOF-SWITCH PIC XXX.
```

VALUE (to Specify a Disk Filename)

Syntax in DATA DIVISION

```
VALUE OF FILE-ID IS {data-name | literal}
```

The VALUE OF clause is required in the FD entry of any file assigned to DISK. Its function is to designate the specific file to be opened.

Details

The value of the file ID may be specified as either a data-name or a quoted literal not longer than 16 characters. This clause may not be used for a file assigned to PRINTER. Any time you use the VALUE OF clause, it is also necessary to include the LABEL RECORDS ARE STANDARD clause.

Application

```
VALUE OF FILE-ID IS "A:MASTER.DAT"
```

WORKING-STORAGE

Syntax in DATA DIVISION

```
WORKING-STORAGE SECTION.
```

The WORKING-STORAGE SECTION is the section in which internally stored data items are defined. If you include this section, it must immediately follow the FILE SECTION.

Details

Data definitions in this section may employ level numbers 01-49, as in the FILE SECTION, as well as level 77. Value clauses, including 88-level condition-names, are permitted throughout the WORKING-STORAGE SECTION.

WRITE (to Perform Sequential Output)

Syntax in PROCEDURE DIVISION

```
WRITE record-name [FROM data-name-1]

[ {AFTER | BEFORE} ADVANCING {PAGE |  {integer  }  {LINE | LINES}} ]
                                      {data-name}

[AT {END-OF-PAGE | EOP} imperative-statement]
```

The WRITE statement causes a record of a file opened in the output mode to be sent to the assigned storage or output device. WRITE with indexed and relative files is discussed in Chapters 10 and 11.

Details

LINE may be coded in the singular or plural. END-OF-PAGE and EOP are identical in effect. Record-name must be one of the 01 level data items in the file being written. It may be qualified by the filename. Execution of a WRITE statement updates the FILE STATUS data item (if defined) for the associated file.

If you attempt to WRITE beyond the physical boundary of a disk file, a declarative procedure will be executed if you have provided one. If no declarative procedure exists and you have not defined a FILE STATUS item, a runtime error will occur. If a FILE STATUS item has been specified, it will be set to indicate a boundary violation. Execution will then proceed normally on the assumption that the FILE STATUS will be tested to determine a course of action.

Inclusion of the FROM clause will cause the data to be moved from another storage area (defined in either the File or Working-Storage Section) to the specified record area before the data are written. It functions exactly as if a MOVE statement were coded immediately ahead of the WRITE statement.

The ADVANCING clause is allowed only for records being output to a printer. Between 0 and 120 line feeds, as specified by the value of integer or data-name, are sent either BEFORE or AFTER the line is printed. If you omit the ADVANCING clause, a one-line advance is assumed. Use of the PAGE option causes the printer to advance to the next top-of-form, rather than a specified number of lines. If you include a LINAGE clause in the associated file description, PAGE will advance to the top of the *next logical* page as defined in the *LINAGE* clause.

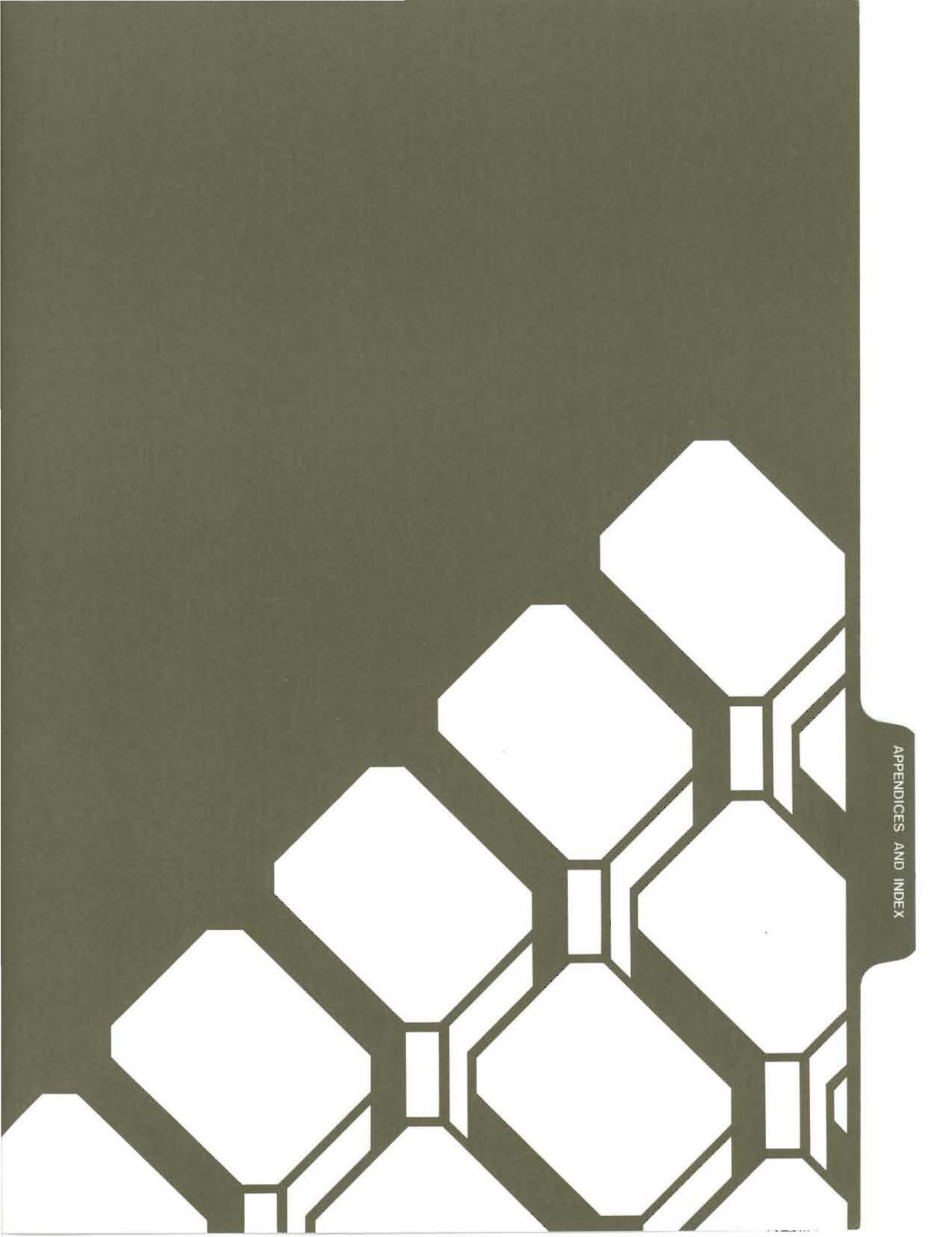
WRITE (to Perform Sequential Output)

Use of the AT clause is also restricted to printed records. It may be included only if a LINAGE clause is included in the associated file description. Imperative-statement is executed when the linage counter indicates an END-OF-PAGE condition. The END-OF-PAGE condition exists whenever execution of a WRITE statement causes printing in the FOOTING area of the page.

Application

```
WRITE record-line FROM ws-report-image
    AFTER ADVANCING 2 LINES

WRITE record-line AFTER ADVANCING lf-counter LINES
    AT END PERFORM select-page-format
```


APPENDICES AND INDEX

Part IV
Appendices and Index

APPENDIX A INTERPROGRAM COMMUNICATION

Overview

Interprogram communication is accomplished by using the CALL or CHAIN statement. CALL temporarily transfers control to another program or assembly language subroutine, and CHAIN permanently transfers control to another program. In linking, the calling and called programs or subroutines are linked together, while chained programs are linked separately. The various communications possible with CALL and CHAIN are:

1. Temporary transfer of control from one COBOL-86 program to another (CALL).

2. Temporary transfer of control from a COBOL-86 program to an assembly language subroutine (CALL).

3. Permanent transfer of control from one COBOL-86 program to another (CHAIN).

4. Permanent transfer of control from a COBOL-86 program to an assembly language program (CHAIN).

In addition to transferring program control, these statements can transfer data between programs. This is done with the USING and chaining clauses. In a CALL statement, the USING clause lists parameters that give the addresses of data to be acted on within the called program. These data are specified in a corresponding USING clause in the PROCEDURE DIVISION statement of the called program. The called program makes any necessary changes and then returns control to the calling program.

When a program is chained, the USING clause of the CHAIN statement also contains parameters, but in this case the actual values of the parameters in the chaining program are substituted for those of the chained program. This happens because the runtime system copies the data values listed in the chaining program to high memory, loads the chained program into memory, and copies the data values into their corresponding parameters in the chained program. These parameters are specified by a chaining clause in the PROCEDURE DIVISION statement of the chained program.

Note that COBOL-86 programs may pass no more than 12 parameters, and the maximum number of files that may be open in one run unit (a program linked together with other programs or subroutines) is 14.

INTERPROGRAM COMMUNICATION

Calling COBOL Programs

Syntax

```
CALL literal [USING data-name...]
```

literal is the PROGRAM-ID defined in the IDENTIFICATION DIVISION of a COBOL program. The literal must be non-numeric and enclosed in quotation marks. Data-name(s) are references whose addresses are passed to the called program.

Purpose

CALL temporarily transfers control to another COBOL-86 program. The two programs are compiled separately and then linked together (see Chapter 3). Control will be returned to the calling program by an EXIT PROGRAM statement in the called program.

Details

The USING clause specifies data items in the calling program that can be used by the called program. For example, a program that needed inventory totals could CALL another program to calculate the totals and place them into designated data-names in the calling program. When this clause is used, the following requirements must be met:

1. Within the calling program: The data-names listed in the USING clause must be declared in the WORKING-STORAGE SECTION of the DATA DIVISION.

2. Within the called program: The data-names corresponding to those in the USING clause of the calling program must be declared in the LINKAGE SECTION of the DATA DIVISION and in a USING clause after the PROCEDURE DIVISION header. The names in the LINKAGE SECTION and in the PROCEDURE DIVISION header must be in the same order.

   Control is returned to the calling program by an EXIT PROGRAM statement in the PROCEDURE DIVISION.

You must make sure that the data items listed in the calling program and in the called program are equivalent. See CALL and CHAIN in Part III, "Reference Guide," for more detailed information on data items.

Sample Program Structure

```
Calling Program
 .
 .
 .
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA-NAME PIC 99.
 .
 .
 .
PROCEDURE DIVISION.
 .
 .
 .
    CALL PROG2 USING DATA-NAME.
 .
 .
 .
Called Program

IDENTIFICATION DIVISION.
PROGRAM-ID. PROG2.
 .
 .
 .
DATA DIVISION.
LINKAGE SECTION.
01  LOCAL-REFERENCE PIC 99.
 .
 .
 .
PROCEDURE DIVISION USING LOCAL-REFERENCE.
 .
 .
 .
    EXIT PROGRAM.
```

INTERPROGRAM COMMUNICATION

Calling Assembly Language Subroutines

A COBOL-86 program may call assembler subroutines. (See the MACRO-86 Assembler portion of the *Z-DOS* manual for instructions on writing assembly language programs.) The runtime system transfers execution to a subroutine by means of a machine language FAR CALL instruction. Execution should return via the MACRO-86 RET instruction.

Parameters are passed by reference (i.e., by passing the address of the parameter). Parameter addresses are passed on the stack (see Figure A.1).

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. You must determine that the type and length of arguments passed by the calling program are acceptable to the called subroutine; neither the compiler nor the common runtime system checks for the correct number of parameters. Numeric values to be passed should be declared as binary (i.e., USAGE IS COMP-0 in the WORKING-STORAGE SECTION of the calling program).

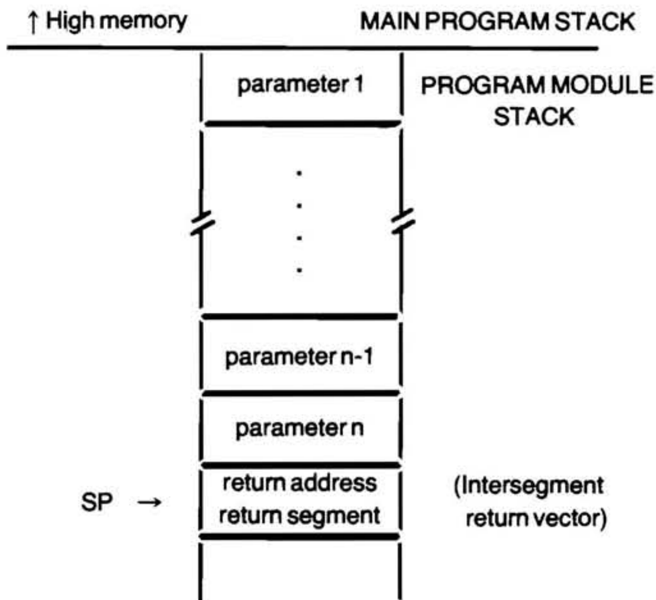


Figure A.1. Contents of Stack at Entry to a Routine

INTERPROGRAM COMMUNICATION

Calling Assembly Language Subroutines

Because the stack space used by a COBOL-86 program is contained within the program boundaries, assembler programs that use the stack must not overflow or underflow the stack. The best way to assure safety is to save the COBOL-86 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-86 stack pointer before returning to the main program.

To call an assembler program module, use the name of the module in the CALL statement. The name of an assembler program module is defined by a PUBLIC directive and is declared as PROC FAR. Compile and/or assemble the program(s) and assembly language subroutine(s). Then link the called program module to the calling program using the linker, as described in Chapter 3 and in the *Z-DOS* manual.

Sample Program Structure

```
 COBOL Program

 IDENTIFICATION DIVISION.
 PROGRAM-ID.EXAMPLE.
*DEMONSTRATE CALLING AN ASSEMBLY LANGUAGE PROGRAM
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  PARM1      PIC 99 COMP-0 VALUE 50.
 77  PARM2      PIC 99 COMP-0 VALUE 45.
 77  PARM3      PIC 99 COMP-0 VALUE 0.
 77  PAR1       PIC 99.
 77  PAR2       PIC 99.
 77  PAR3-DIF   PIC 99.
 PROCEDURE DIVISION.
 MAIN.
     CALL 'SUBIT' USING PARM1, PARM2, PARM3
     MOVE PARM1 to PAR1.
     MOVE PARM2 to PAR2.
     MOVE PARM3 to PAR3-DIF.
     DISPLAY PAR1 ' - ' PAR2 ' = ' PAR3-DIF.
     STOP RUN
```

INTERPROGRAM COMMUNICATION

Calling Assembly Language Subroutines

Assembly Language Program

```
        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  subit           ;entry point
subit   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
        xchg    ah,al           ;swap bytes (COBOL stores
                                ;high-order byte first)
        mov     dx,[bx]         ;get 2nd value
        xchg    dh,dl           ;swap bytes
        sub     ax,dx           ;compute difference
        mov     bx,[bp].parm2   ;get addr of parm2
        mov     di,[bp].parm3   ;get addr of parm3
        xchg    ch,al           ;swap bytes back for COBOL
        mov     [di],ax         ;put result into parm3
        pop     bp              ;restore caller's bp
        ret     6               ;restore stack
subit   endp
codeseg ends
        end
```

Chaining COBOL Programs

Syntax

<u>CHAIN</u> {literal | data-name-1} [<u>USING</u> data-name-2...]

`Literal` or `data-name-1` is the filename of an executable program. The only difference between them is that the literal must be enclosed in quotation marks, while the data-name does not use quotation marks. Either must

be alphanumeric. Data-name-2 is a data item identified in the WORKING-STORAGE SECTION of the chaining program.

For more details about CHAIN format, see Part III of the manual.

Purpose

CHAIN permanently transfers control to a separately compiled and separately linked program, which is loaded into memory and executed. The chained program can issue its own CHAIN statement or may even issue a CHAIN statement to its original chaining program, but it cannot issue an actual return to the original program.

Details

If you include the USING clause, the values of the data items listed there will be copied to high memory, and when the chained program is loaded and run, they will be substituted for the equivalent values in the chained program. This allows you to run a new program using values established in an earlier program. When you use this clause, the following requirements must be met:

1. Chaining program: The data items listed in the USING clause must be declared in the WORKING-STORAGE SECTION of the DATA DIVISION.

2. Within the chained program: The data items corresponding to those in the USING clause of the chaining program must be declared in the WORKING-STORAGE SECTION of the DATA DIVISION and in a chaining clause after the PROCEDURE DIVISION.

Sample Program Structure

```
Chaining Program:
 .
 .
 .
DATA DIVISION.
WORKING-STORAGE SECTION.
01  READ DATA-NAME PIC 99.
 .
 .
 .
```

INTERPROGRAM COMMUNICATION

Chaining COBOL Programs

```
PROCEDURE DIVISION.
  .
  .
  .
    CHAIN PROG2 USING DATA-NAME.
Chained Program
  .
  .
  .
DATA DIVISION.
LINKAGE SECTION.
01  LOCAL-REFERENCE PIC 99.
  .
  .
  .
PROCEDURE DIVISION CHAINING LOCAL-REFERENCE.
  .
  .
  .
```

Chaining Assembly Language Programs

Assembly language programs are chained exactly as are COBOL-86 programs (see Chaining COBOL Programs of this appendix). The following additional information will be useful when you are writing assembly language programs that will be chained.

When the USING clause is included in the CHAIN statement, the parameters passed between programs are stored at the highest available memory address. This address is determined from byte 2 of the program header (see Z-DOS system documentation for more information).

The memory layout is as follows, starting at the highest available address and proceeding toward location zero (see Figure A.2):

1. First, 256 bytes are reserved for stack space.

Chaining Assembly Language

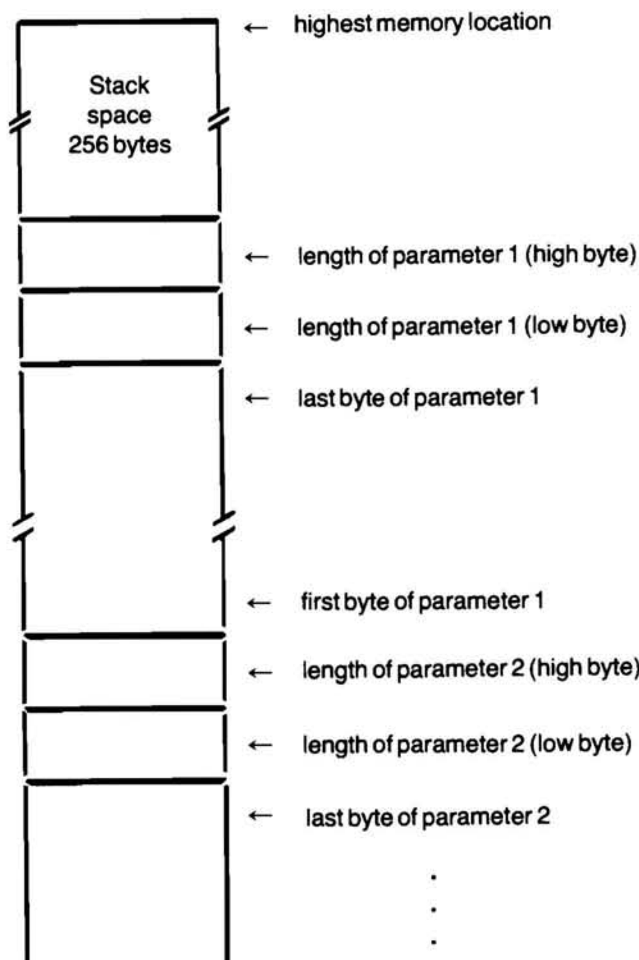


Figure A.2 Memory Layout for Chained Programs

2. Then the first parameter in the USING list follows, preceded by its length in bytes. The parameter length is stored in two bytes, high-order byte first. The parameter itself is stored as a string of bytes in the same order as the bytes were stored in the DATA DIVISION, beginning at the address of the length minus the length itself (see Figure A.2).

3. 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. No checking will be done by the compiler or the common runtime system.

APPENDIX B CUSTOMIZATIONS

Source Program Tab Stops

This appendix is intended for those who are proficient with a debugger and/or assembly language and would like to change some of the built-in parameters of COBOL-86.

If tab characters (hex 09) are used in the COBOL-86 source program, the compiler converts them into enough spaces to reach the next tab stop as defined in its internal TAB table. The table originally defines ten stops at the following columns (counting from column 1):

8, 12, 20, 28, 36, 44, 52, 60, 68, and 73

These may be changed by patching the table, whose address is 15 bytes from the start of COBOL.COM. There is one byte in the table for each tab stop. You may supply any values you like, provided that: (1) the numbers are in ascending order; (2) no more than ten stops are defined; (3) the last tab stop is 73.

Compiler Listing Page Length

One byte in the compiler defines the page length of the listing as 55 (hex 37) lines. Its location is 14 bytes from the start of COBOL.COM, and it may be patched to any value between 1 and 255.

APPENDIX C COMPILER PHASES

The COBOL-86 Compiler creates an object code program from your source program. This is done in five phases, consisting of the root portion of the compiler (COBOL.COM) and four overlays, COBOL1.OVR through COBOL4.OVR. These are the phases referenced by an error message such as `?Compiler error in phase n`. They are used as follows.

Compilation is performed in two passes: The first pass creates an intermediate version of the program, which is stored in a binary file called COBIBF.TMP. This is done in three steps:

- The root portion of the compiler (Phase 0) compiles the IDENTIFICATION and ENVIRONMENT DIVISIONS of the source program
- Phase 1 (COBOL1.OVR) compiles the DATA DIVISION of the source program.
- Phase 2 (COBOL2.OVR) compiles the PROCEDURE DIVISION of the source program.

The compiler's second pass reads the intermediate file and creates the object code:

- Phase 3 (COBOL3.OVR) reads the intermediate file and creates the object code.
- Phase 4 (COBOL4.OVR) allocates file control blocks and finalizes the object code.

APPENDIX D REBUILD: INDEXED FILE RECOVERY UTILITY

Overview

The Indexed File Recovery Utility (REBUILD) can be used to recover or restore information contained within indexed files. The indexed files that are compatible with this utility are those that have been created by a program compiled under COBOL-86 Version 1.00 or later.

REBUILD works by reading the data file portion of an indexed file and generating new key and data files for that indexed file. The new indexed file has the same structure as the old one. The utility will skip over all deleted records and any other control records within the data file.

Use of REBUILD is recommended in the following situations:

1. When disk space is exhausted during a WRITE operation (file status = 24).

2. When electrical power to the computer system is interrupted or the operating system is rebooted while an indexed file is OPEN in I-O or OUTPUT mode.

3. When the data file portion of the indexed file contains large areas of unused space, usually as a result of numerous record DELETE and REWRITE operations, and especially when records within the file have varying lengths.

Situation 1 occurs when WRITE produces a boundary error (file status "24"), indicating that the disk is full. When this happens, you should perform a CLOSE in order to write as much information as possible to disk. It is likely, however, that the CLOSE will also return with a boundary error. As in the case of a system failure during the addition of records, the last 256 bytes of information will not be present within the data file, and is therefore not recoverable by REBUILD.

Recovery from situation 2 may also be limited, because without a transaction file to rebuild the indexed file, recovery from some types of system failure is a problem. Because of the high degree of disk file buffering in memory, a system failure may leave the data file with partially-written data records. This may cause REBUILD to fail to completely recover an indexed file for two reasons:

REBUILD: INDEXED FILE RECOVERY UTILITY

Overview

a. Because a good deal of information is kept in memory, if the system failure occurred during a file update job, the file may contain records with both original and new information. The recovery utility cannot determine which part of the data was written during the aborted job, and therefore cannot exclude the new, incomplete data from the rebuilt file. Adding a current date field to data records may help discriminate between original and new data.

b. If the system failure occurred while records were being added to the indexed file, the last 256 bytes of data will not be written to disk. The recovery utility will detect that information is missing from the end of the file but cannot add it to the recovered file.

Running Rebuild

Since REBUILD is itself a COBOL-86 program, COBRUN.EXE must be present on a disk in the default drive or drive A when you are running REBUILD. Invoke the recovery utility by entering:

```
REBUILD
```

in response to the operating system prompt.

The utility will respond with the following header information:

```
REBUILD by Microsoft Corporation
Indexed File Recovery Utility
V. 1.0
```

Use this utility to recover indexed files when they are damaged, or to reorganize indexed files by removing unused space. Compatible indexed files are those generated by COBOL-86 for versions 1.00 and later.

Running Rebuild

The recovery utility will then ask a series of questions. Your answers will provide the information necessary for rebuilding a new indexed file from the original data file. The steps you follow while using REBUILD are diagramed in Figure D.1. Following the diagram are detailed descriptions of the individual recovery steps and a sample REBUILD session.

1. Input Key Length: Enter the key length in reply to the prompt:

```
Input the key length (in bytes)
or <RETURN> to terminate program ---->
```

   Enter a key length or press the RETURN key to immediately terminate the program. If you enter a key length, the program will proceed to the next prompt.

   The key length should be a positive integer that represents the number of bytes contained in the item specified by the RECORD KEY clause of a COBOL-86 program. Failure to enter the correct key length may not hamper the execution of REBUILD, but programs will not be able to access the generated indexed file.

2. Input Key Position: Enter the key position in reply to the prompt:

```
Input the byte position of the key field,
starting at 1,
or <RETURN> to return to the Key Length prompt ---->
```

   Enter the position of the key data item within the record; or press the RETURN key to move back to the Input Key Length prompt in order to correct information or terminate the program. If you enter a key position, the program will proceed to the next prompt.

   The key position should be a positive integer that represents the position within the record of the data item specified by the RECORD KEY clause of a COBOL-86 program. As with the key length, REBUILD does not check whether an incorrect response has been entered; but the result of an incorrect response will be that programs will not be able to access the generated indexed file.

REBUILD: INDEXED FILE RECOVERY UTILITY

Running Rebuild

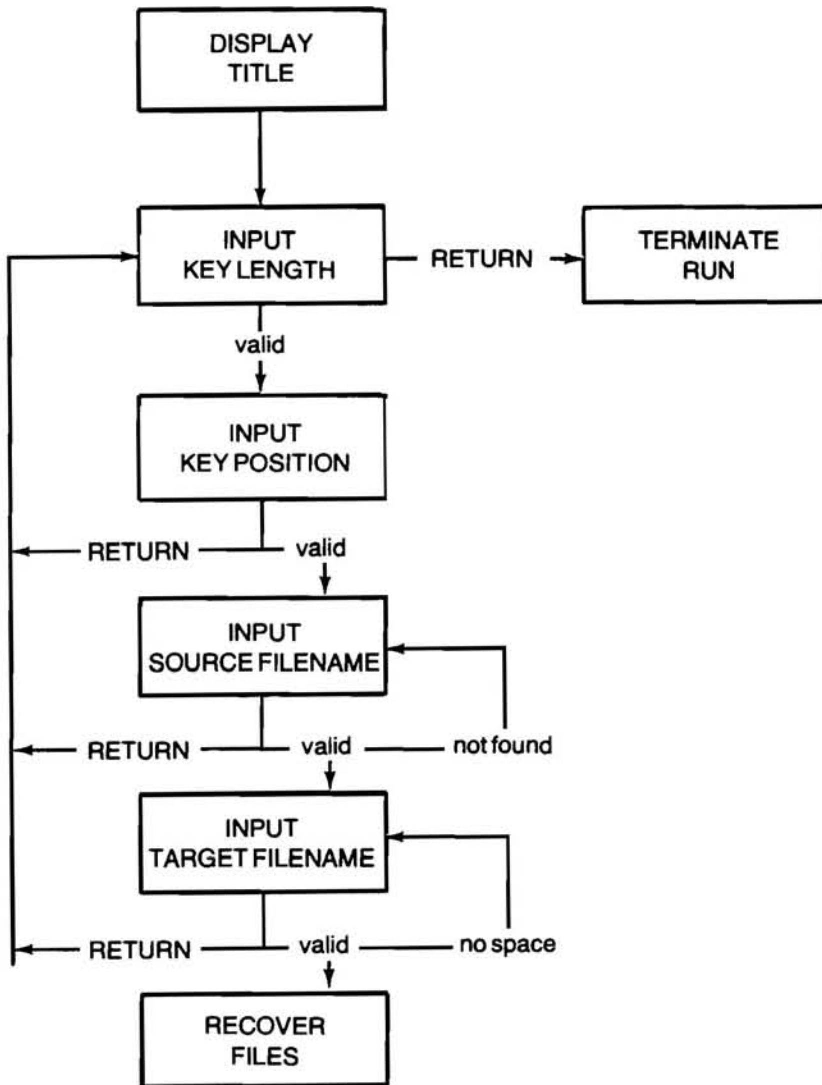


Figure D.1. Control Flow within REBUILD

REBUILD: INDEXED FILE RECOVERY UTILITY

Running Rebuild

3. Input Source Filename: Enter the filename of the source file in reply to the prompt:

```
Input the filename of the source data file
(should not have extension of .KEY)
or <RETURN> to return to the Key Length prompt ---->
```

Enter a filename or press the RETURN key to move back to the Input Key Length prompt so that you can correct and re-enter previous information or terminate the program.

The source filename should be the name that is used in the VALUE OF FILE-ID clause in COBOL-86 programs that refer to the indexed file. The filename used here should be the name of the data file. The key file, which has the same name but an extension of .KEY, will not be used in the recovery operation and should not be entered in response to this prompt.

The source filename may contain a drive specifier.

After the source filename is entered, REBUILD will check for the presence of the file. If it is not present, the following message will be displayed:

```
***Source file not found
```

and the Input Source Filename prompt will be redisplayed.

4. Input Target Filename: Enter the filename of the indexed file to be generated in reply to the prompt:

```
Input the filename of the target data file
(should not have extension of .KEY)
or <RETURN> to return to the Key Length prompt ---->
```

Enter a filename or press the RETURN key. As usual, RETURN moves you back to the Input Key Length prompt so that you can re-enter information or terminate the program.

As with the source file, this name is the name of the data file. Do not enter the key file, which has the same name but the .KEY extension.

REBUILD: INDEXED FILE RECOVERY UTILITY

Running Rebuild

The target filename should be unique within a directory. Therefore, if you wish to use a name identical to the source filename, you should send the target file to a different disk by including a drive specifier in the filename. The target file can be generated on the same disk as the source file, but you will have to use a different name. Once the recovery operation is complete, you can then rename the target filename to the source filename.

If the recovery utility cannot successfully create a new indexed file, either because the disk directory is full or because of insufficient space on the disk, the program will display the message:

```
*** No space for target file
```

and will redisplay the Input Target Filename prompt.

5. Recover File: After you have answered all questions, the recovery utility will display:

```
Now reading <source-file>
and creating <target-file>
```

The program will begin building the new indexed file from the old data file. When this process is finished, the following message will be displayed:

```
Conversion successfully completed.
Source records read:    xxx,xxx
Target records read:    xxx,xxx
```

The record counts should match. If they do not, some type of input-output error occurred during the recovery operation.

Regardless of whether the record counts match, REBUILD will then display another Input Key Length prompt. You can begin another file recovery operation (or redo the one that had an input-output error) or terminate the program.

REBUILD: INDEXED FILE RECOVERY UTILITY

Sample Rebuild Session

The following program fragment accesses the indexed file IXFILE.DAT:

```
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).
```

For this program fragment, the responses to the REBUILD utility would be:

```
Input Key Length:   52
Input Key Position:   13
Input Source Filename:   IXFILE.DAT
Input Target Filename:   NEWIX.DAT
```

The result of the recovery operation would be to generate a new indexed file with the key filename NEWIX.KEY and the data filename NEWIX.DAT.

APPENDIX E COBOL-86 ERROR MESSAGES

Overview

This appendix lists all the error messages you may encounter while compiling and executing a COBOL-86 program. Included here are:

1. Compile time errors, which can be:

   a. Command input errors and operating system input/output errors. These errors will be displayed as the errors occur during the compile. When you receive one of these messages, correct the problem and recompile.

   b. Program syntax errors in the COBOL-86 source program. These messages are placed at the end of the listing file and are also shown on the screen. They consist of:

      (1) The source program line number, which is four digits followed by a colon (:).

      (2) An explanation of the error. If the explanation begins with an /F/ (inconsistent file usage) or a /W/ (warning), then the message is only a warning; if not, the error is severe enough to prevent you from linking and executing the object file.

   Whether or not a listing has been requested, the syntax error messages will always be listed on your screen at the end of compilation. A message displaying the total number of errors or warnings is also displayed. This feature allows you to make a simple change to a program, recompile it without a listing, and still receive any error messages.

   Program syntax error messages in this manual are listed in alphabetical order, with /F/ and /W/ warnings placed at the end of the list. The line number found with an /F/ warning represents the order of files in the FILE SECTION of the COBOL-86 program.

2. Runtime errors, which can be:

   a. COBOL-86 execution errors. Some programming errors cannot be detected by the compiler but cause the program to end prematurely during execution. These runtime errors are displayed in the format:

COBOL-86 ERROR MESSAGES

Overview

```
**RUN-TIME ERR:
reason
line number (see Using Compiler Switches in Chapter 2)
program-id
```

b. COBOL-86 program load errors: chained programs, independent segments (i.e., overlays), and the common runtime executor need to be loaded by the COBOL-86 runtime system. During loading, the normal mechanism for reporting runtime errors may have been overlaid by the new program. Therefore, the COBOL-86 loader generates its own error messages. The syntax is:

```
**COBOL:  problem
```

Command Input and Operating System I/O Errors

```
?Bad filename
```

A filename is not constructed according to the rules of the operating system.

```
?Bad switch:/X
```

You have entered a switch parameter that the compiler does not recognize.

```
?Can't create file
```

An output file cannot be opened. For example, the output disk is write protected.

```
?Command error:'X'
```

You have an invalid character (X) in the command line. For example, a file name contains an @.

COBOL-86 ERROR MESSAGES

Command Input and Operating System I/O Errors

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

This is caused by a damaged source program or damaged compiler or overlay file. In the latter case, try your backup copy. If this does not work, you can sometimes determine the cause of the error by compiling increasingly larger portions of the program, starting with only a few lines, until the error recurs.

See Appendix C for a discussion of compiler phases.

```
?Disk X full
```

The disk in the specified drive is full. If X is blank, it refers to the default drive.

```
?File not found
```

You have specified a filename for input that does not exist.

```
?Memory full
```

This occurs when there is insufficient memory for all the symbols and other information obtained from the source program. It indicates that the program is too large and must be decreased in size or split into modules and compiled separately.

The symbol table of data-name(s) and procedure-names usually occupies the most space during compilation. All names require as many bytes as there are characters in the name, with 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 uses about 14 bytes of memory during compilation, and each line in the PROCEDURE DIVISION uses about 3 1/4 bytes.

```
?Overlay n not found
```

One of the COBOL-86 Compiler overlay files (COBOLn.OVR) is not on the disk. It may have been written to another disk or destroyed. Recompiling and relinking may eliminate the problem.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

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, the numeric literal is over 18 digits, or the identifier/name is over 30 characters.

ERRONEOUS FILENAME IS IGNORED.

An entry that 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.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

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 to more than three levels.

EXCESSIVE SEGMENT NUMBER.

A section header contains a segment number greater than 99.

EXCESSIVE SEGMENT NUMBER IN DECLARATIVES.

A section header in the DECLARATIVES region contains a segment number greater than 49.

FILE NOT SELECTED; ENTRY BYPASSED.

An FD is given for a filename that does not appear in any SELECT sentence.

FILL CHARACTER CONFLICT.

In a Format 3 ACCEPT statement, SPACE-FILL and ZERO-FILL are both specified.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

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 right-most digit.

GROUP ITEM, THEREFORE PIC/JUST/BLANK/SYNC IS IGNORED.

A phrase that is only allowed for elementary data items is used in the description of an item that 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 invalid.

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.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

```
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 ended a word with a hyphen or 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.

```
INDEXED/RELATIVE REQUIRES DISK ASSIGNMENT.
```

A file assigned to PRINTER is described as having indexed or relative organization.

```
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. This message is issued when the OPEN statement is processed.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

INVALID QUOTED LITERAL.

A literal of zero length, improper construction, or missing end quotes has occurred.

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.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

```
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 to a section-name or procedure-name 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.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

```
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 that 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 RESTORE or AFTER phrase of a WRITE statement is not numeric, or it is outside the range 0-120.

```
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 has not been declared as a filename.

```
STATEMENT DELETED DUE TO ERRONEOUS SYNTAX.
```

A syntax error, to which no more specific message applies, is present.

```
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 longer than 18 digits.

```
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 that requires qualification is used as a subscript.

```
SYNTAX ERROR IN SCREEN DESCRIPTION.
```

A screen item description contains a clause that is unrecognizable, improperly constructed, or redundant.

```
UNRECOGNIZABLE ELEMENT IS IGNORED.
```

A required keyword is missing, or a data-name or procedure-name is unidentified.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

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.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

```
/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/ ERRONEOUS RERUN-ENTRY IS IGNORED.
```

A RERUN clause of the I-O-CONTROL paragraph contains a syntax error.

```
/W/ FD-VALUE IGNORED SINCE LABELS ARE OMITTED.
```

The VALUE OF FILE-ID clause is used in the description of a file that 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/ INVALID RECORD SIZE(S) IGNORED.
```

The RECORD 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.

COBOL-86 ERROR MESSAGES

Program Syntax Errors

```
/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 in a file description.

```
/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.

COBOL-86 ERROR MESSAGES

Runtime Errors

`DATA UNAVAILABLE.`

You tried to reference data in a record of a file that is not open or has reached the AT END condition.

`DELETE; NO READ.`

You tried to DELETE a record of a sequential access mode file when the last operation was not a successful READ.

`FILE LOCKED.`

You tried to OPEN after an earlier CLOSE WITH LOCK.

`GO TO (NOT SET).`

You tried to execute a null GO statement that has never been altered to refer to a destination.

`ILLEGAL DELETE.`

Relative or indexed file not opened for I-O.

`ILLEGAL READ.`

You tried to READ a file that is not open in the INPUT or I-O mode.

`ILLEGAL REWRITE.`

You tried to REWRITE a record in a file not open in the I-O mode.

`ILLEGAL START.`

File not opened for INPUT or I-O.

`ILLEGAL WRITE.`

You tried 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.

COBOL-86 ERROR MESSAGES

Runtime Errors

INPUT/OUTPUT.

Unrecoverable I/O error, with no provision in the user's COBOL-86 program for acting upon the situation by way of an AT END clause, INVALID KEY clause, FILE STATUS item, or DECLARATIVES SECTION.

NEED MORE MEMORY.

The indexed file manager has ended abnormally because of insufficient dynamically allocatable memory.

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 they are subject to error (because input editing has not yet been done) by use of the NUMERIC test.

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.

PERFORM OVERLAP.

An illegal sequence of PERFORMs, as, for example, when paragraph A is performed and another PERFORM A is initiated prior to exiting from the first.

REDUNDANT OPEN.

You tried to open a file that is already open.

REWRITE; NO READ.

You tried to REWRITE a record of a sequential access mode file when the last operation was not a successful READ.

COBOL-86 ERROR MESSAGES

Runtime Errors

```
SEG nn LOAD ERR.
```

An error occurred while you were attempting to load an overlaid segment. nn is 31 hex (49 decimal) less than your overlay segment number.

```
SUBSCRIPT FAULT.
```

A subscript has an illegal value. This error may be caused by an index reference whose value is less than 1.

Program Load Errors

```
**COBOL: ATTEMPT TO USE NON-UPDATED RUNTIME MODULE (COBRUN.EXE).
```

This message appears when the version number in the runtime libraries is not the same as that in the runtime interpreter (COBRUN.EXE).

```
**COBOL: ERROR IN EXE FILE.
```

Error in loading chained or common runtime EXE file.

```
**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 not enough memory available to load a chained program or common runtime file.

APPENDIX F DEMONSTRATION PROGRAMS

COBOL-86 Programs

This is a brief description of demonstration programs included with COBOL-86. See also Key Designations in Chapter 12.

CRTEST

CRTEST is a test program for the terminal interface. CRTEST must be compiled and linked before it can be run. (See Program Development in Chapter 1.) CRTEST will prompt you for input.

CENTER

CENTER is a program that takes a line of text and centers it or aligns it with the left or right margin. It is a simple COBOL program that does not use sophisticated screen handling features. Like CRTEST, it also must be compiled and linked before running it. It will also prompt you for input.

COBOL-86 Demonstration System

The COBOL-86 Demonstration System consists of three COBOL programs:

DEMO.COB
BUILD.COB
UPDATE.COB

Linked versions of these programs are also included (DEMO.EXE, DEMO_01.OVL, and UPDATE.EXE).

DEMO is the executive program of the system. It asks if you would like a demonstration of the COBOL-86 SCREEN SECTION, or whether you would like to create or update an indexed (ISAM) file of name, addresses, and phone numbers.

Use the following procedure to run the COBOL-86 Demonstration System.

DEMONSTRATION PROGRAMS

COBOL-86 Demonstration System

1. Either copy COBRUN.EXE onto the disk containing the files DEMO.EXE, UPDATE.EXE, and DEMO_01.OVL or insert a disk containing COBRUN.EXE into drive A.

2. Insert the disk containing the files DEMO.EXE, DEMO_01.OVL, and UPDATE.EXE into drive B.

3. Type

   B:

   and press RETURN to make drive B the default drive.

4. Now type

   DEMO

   and press RETURN. When DEMO has been loaded, it will prompt you for input by providing menus and information screens to guide you through the demonstration.

   The COBOL source files for DEMO, BUILD, and UPDATE are included to allow you to see the code that produces screens and files of the system. To recreate the system from the source files, you perform the following steps:

1. Insert a disk containing the compiler (COBOL.COM) and COBOL overlays (COBOL1.OVR_COBOL4.OVR) into drive A. Insert the disk containing DEMO.COB, BUILD.COB, and UPDATE.COB into drive B. We recommend that you copy these files onto a blank disk to allow room for object (.OBJ) and executable (.EXE) files on the disk.

   Make drive B the default drive by typing

   B:

   and press RETURN.

2. Now type

   ```
   A:COBOL DEMO,,CON;
   ```

   and press RETURN. This compiles DEMO.COB and produces DEMO.OBJ. The use of "CON" in the command line directs the compile listing to the terminal screen; this allows you to watch the program compile. You should receive the message No errors or warnings when the compilation process is finished.

3. Type

   ```
   A:COBOL BUILD,,CON;
   ```

   and press RETURN to compile BUILD.COB.

4. When the compilation process is finished, type

   ```
   A:COBOL UPDATE,,CON;
   ```

   and press RETURN to compile UPDATE.COB. When the compilation process is finished, type

   ```
   DIR *.OBJ.
   ```

   You should find the files DEMO.OBJ, BUILD.OBJ, and UPDATE in the directory listing.

5. Replace the disk in drive A with your utility disk (see Program Development in Chapter 1) containing LINK.EXE, COBOL1.LIB, COBOL2.LIB, and COBRUN.EXE.

   Link DEMO.OBJ and BUILD.OBJ together by typing

   ```
   A:LINK DEMO+BUILD,,A:;
   ```

   Note that DEMO01.OVL is produced in addition to DEMO.EXE.

6. Link UPDATE.OBJ by typing

   ```
   A:LINK UPDATE,,A:;
   ```

   You may now run the system as before.

APPENDIX G ASCII CHARACTER SET FOR ANSI-74 COBOL

CHARACTER	HEXADECIMAL VALUE	CHARACTER	HEXADECIMAL VALUE
A	41	0	30
B	42	1	31
C	43	2	32
D	44	3	33
E	45	4	34
F	46	5	35
G	47	6	36
H	48	7	37
I	49	8	38
J	4A	9	39
K	4B	(SPACE)	20
L	4C	"	22
M	4D	$	24
N	4E	' (non-ANSI)	27
O	4F	(	28
P	50	)	29
Q	51	*	2A
R	52	+	2B
S	53	,	2C
T	54	-	2D
U	55	.	2E
V	56	/	2F
W	57	;	3B
X	58	<	3C
Y	59	=	3D
Z	5A	>	3E

Plus-zero (zero with embedded positive sign);	7B
Minus-zero (zero with embedded negative sign);	7D

APPENDIX H ADDITIONAL ANSI RESERVED WORDS

The following words are reserved in the ANSI 1974 standard, but are not used by COBOL-86. To ensure program portability, you should not use these words as names.

ALSO	ALTERNATE	CANCEL
CD	CF	CH
CLOCK-UNITS	COBOL	CODE
COMMUNICATION	CONTROL(S)	CORRESPONDING
DE	DEBUG-CONTENTS	DEBUG-ITEM
DEBUG-LINE	DEBUG-NAME	DEBUG-SUB-1
DEBUG-SUB-2	DEBUG-SUB-3	DESTINATION
DETAIL	DISABLE	DUPLICATES
EGI	EMI	ENABLE
ENTER	ESI	EVERY
FINAL	GENERATE	GROUP
HEADING	INITIATE	LAST
LENGTH	LIMIT(S)	LINE-COUNTER MERGE
MESSAGE	MULTIPLE	NAMES(non-ANSI)
NO	OFF	OPTIONAL
PAGE-COUNTER	PF	PH
POSITION	PRINTING	QUEUE
RD	RECEIVE	REEL
REFERENCES	RELEASE	REMAINDER
REMOVAL	RENAMES	REPORT(S)
REPORTING	RERUN	RETURN
REVERSED	REWIND	RF
RH	ROUND	SD
SEGMENT	SEGMENT-LIMIT	SEND
SOURCE	SUB-QUEUE-1	SUB-QUEUE-2
SUB-QUEUE-3	SUM	SUPPRESS
SYMBOLIC	TABLE	TAPE
TERMINAL	TERMINATE	TEXT
TYPE	UNIT	

INDEX

A

ACCEPT
- characteristics of the data input field with, 12.13
- character validity rules with, 12.11
- editing characters with, 12.18
- key designations with, 12.10
- as reserved word, 12.2, 12.8–12.24
- terminator keys with, 12.14

ACCESS as reserved word. *See* INPUT-OUTPUT
ADD as reserved word, 12.2, 12.25–12.26
AFTER as reserved word. *See* DECLARATIVES, INSPECT, PERFORM, WRITE
ALL as reserved word. *See* INSPECT, SEARCH, UNSTRING
ALPHABETIC as reserved word. *See* IF
Alphanumeric items
- in DATA DIVISION, 8.2, 8.8
- with PICTURE, 12.89

Alphanumeric-edited items in DATA DIVISION, 8.2, 8.8
ALTER as reserved word, 12.2, 12.27
AND as reserved word. *See* IF
ANSI language
- additional reserved words in, H.1
- ASCII character set for, G.1
- elements of, 7.1–7.2
- processing modules of, 7.1–7.2

AREA as reserved word. *See* INPUT-OUTPUT
ASCENDING as reserved word. *See* OCCURS, SEARCH
ASCII as reserved word. *See* CODE-SET, CONFIGURATION
ASCII character set for ANSI-74 COBOL, G.1
ASSIGN as reserved word. *See* INPUT-OUTPUT
* prompt to indicate debug facility, 6.2
AUTHOR as reserved word. *See* IDENTIFICATION
AUTO as reserved word. *See* SCREEN
AUTO-SKIP as reserved word. *See* ACCEPT

B

BACKGROUND-COLOR as reserved word. *See* SCREEN
Backing up disks, 1.4
Batch command files, 4.1
BEEP as reserved word. *See* ACCEPT
BEFORE as reserved word. *See* INSPECT, WRITE
BELL as reserved word. *See* SCREEN
Binary item in DATA DIVISION, 8.4
BLANK
- as reserved word, 12.2, 12.28
- with SCREEN, 12.102–12.106

BLOCK as reserved word, 12.2, 12.29
BOTTOM as reserved word. *See* LINAGE
BY as reserved word. *See* DIVIDE, INSPECT, MULTIPLY, PERFORM, SET

C

CALL
- as reserved word, 12.2, 12.30–12.31
- to temporarily transfer control of a program, A.1, A.2–A.6

CENTER as a COBOL-86 demonstration program, F.1
CHAIN
- to permanently transfer control of a program, A.1, A.6–A.9
- as reserved word, 12.2, 12.32

CHAINING as reserved word. *See* PROCEDURE
Characteristics of the data input field with ACCEPT, 12.13
CHARACTERS as reserved word. *See* CONFIGURATION, INSPECT, BLOCK, RECORD
Character validity rules with ACCEPT, 12.11
Class-test with IF, 12.61
CLOSE
- as reserved word, 12.2, 12.33
- statement with indexed files, 10.3–10.4

COBOL-86 programs
- with ANSI standard, 7.1–7.3
- coding rules for, 7.6–7.7
- compile time errors in, E.1, E.2–E.3
- compiling, 2.1–2.8
- character set in, 7.4–7.5
- customizing, B.1
- demonstration programs with, F.1– F.3
- error messages in, E.1–E.17
- executing, 3.8
- expediting compilation with, 4.1
- linking, 3.1–3.7
- program load errors in, E.2, E.17
- program syntax errors in, E.1, E.4–E.14
- punctuation in, 7.5
- runtime errors in, E.1–E.2, E.15–E.17
- structure of, 7.7–7.8
- word formation, 7.6

CODE-SET as reserved word, 12.2, 12.34

INDEX

COL as reserved word. *See* ACCEPT, DISPLAY, EXHIBIT
COLUMN as reserved word. *See* SCREEN
COMMA as reserved word. *See* CONFIGURATION
Command input errors, E.1, E.2–E.3
Command strings with operating the compiler, 2.3–2.4
Compiler
  defining page length with, B.1
  operating the, 2.1–2.4
  phases of, C.1
  switches, 2.5–2.6, 2.7
Compile time errors, E.1, E.2–E.3
Compiling
  COBOL programs, 1.1–1.2, 2.1–2.8
  large programs, 2.7–2.8
COMPUTATIONAL-0 as reserved word. *See* USAGE
COMPUTATIONAL-3 as reserved word. *See* USAGE
COMPUTE as reserved word, 12.3, 12.35–12.37
Conditional statements in PROCEDURE DIVISION, 9.1
Condition-name with IF, 12.61
CONFIGURATION as reserved word, 12.3, 12.38–12.39
CONTAINS as reserved word. *See* BLOCK, RECORD
COPY as reserved word, 12.3, 12.40
COPY utility used to back up disks, 1.4
COUNT as reserved word. *See* IF, UNSTRING
CRTEST as a COBOL-86 demonstration program, F.1
CURRENCY as reserved word. *See* CONFIGURATION
Currency symbols with CONFIGURATION, 12.39

D
DATA
  as reserved word with DIVISION header, 12.3, 12.41
  as reserved word with RECORD clause, 12.3, 12.42
Data description entry
  components of, 8.6–8.7
  syntax in, 8.7–8.9
DATA DIVISION, 8.1–8.12
  data description entry in, 8.6–8.9
  data items in, 8.1–8.4
  level numbers in, 8.4–8.6
  literals and figurative constants in, 8.9–8.11
  size limitations in, 8.12
  structures and types in, 8.1–8.4
Data files, 5.3
Data input and output, 5.1–5.4
  types of disk files in, 5.2
  using disk files with, 5.1–5.3
  using Z-DOS and nondisk files with, 5.3–5.4
Data items
  passing, in PROCEDURE DIVISION header, 9.3
  structures and types of, 8.1–8.4
Data-name, use of, 12.46
DATE as reserved word. *See* ACCEPT
DATE-COMPILED as reserved word. *See* IDENTIFICATION
DATE-WRITTEN as reserved word. *See* IDENTIFICATION
DAY as reserved word. *See* ACCEPT
Debug facility
  * prompt used with, 6.2
  commands of, 6.2–6.3
  procedure for, 6.1–6.2
DEBUGGING as reserved word. *See* CONFIGURATION
DECIMAL-POINT as reserved word. *See* CONFIGURATION
DECLARATIVES
  in PROCEDURE DIVISION, 9.4
  as reserved word, 12.3, 12.43–12.44. *See* also PROCEDURE
DELETE
  as reserved word, 12.3
  statement with indexed files, 10.6–10.7
  statement with relative files, 11.4
DELIMITED as reserved word. *See* STRING, UNSTRING
DELIMITER as reserved word. *See* UNSTRING
Demonstration programs, F.1–F.3
DEPENDING as reserved word. *See* GO
DESCENDING as reserved word. *See* OCCURS, SEARCH
Design of the PROCEDURE DIVISION, 9.2–9.6
DISK as reserved word. *See* INPUT-OUTPUT
Disk files
  granules in, 5.2
  types of, 5.1, 5.2–5.3
  using, 5.1–5.3
DISPLAY
  as reserved word, 12.3, 12.45–12.47. *See* also USAGE
  use of data-name, literal, and ERASE with, 12.46
  use of screen-name with, 12.47
DIVIDE as reserved word, 12.3, 12.48–12.49
DIVISION as reserved word. *See* DATA (division), ENVIRONMENT, IDENTIFICATION, PROCEDURE
Division header with PROCEDURE DIVISION, 9.3–9.4
DOWN as reserved word. *See* SET
DYNAMIC as reserved word. *See* INPUT-OUTPUT

INDEX

E

Editing
   characters with ACCEPT, 12.18
   data with PICTURE 12.95
Effects of optional clauses in SCREEN, 12.103
Elementary items
   in DATA DIVISION, 8.2
   syntax in, 8.8
ELSE as reserved word. *See* IF
END as reserved word. *See* DECLARATIVES, READ, SEARCH
END-OF-PAGE as reserved word. *See* WRITE
ENVIRONMENT as reserved word, 12.3, 12.50
ENVIRONMENT DIVISION
   syntax in, with indexed files, 10.1–10.3
   syntax in, with relative files, 11.1–11.2
EOP as reserved word. *See* WRITE
EQUAL as reserved word. *See* IF
ERASE
   as reserved word. *See* DISPLAY, EXHIBIT
   use of, 12.46
ERROR as reserved word. *See* ADD, COMPUTE, DECLARATIVES, DIVIDE, MULTIPLY, SUBTRACT
Error messages with COBOL-86 programs, E.1–E.17
Error reporting with indexed files, 10.2
ESCAPE as reserved word. *See* ACCEPT
ESC key, effects of, 12.21
EXCEPTION as reserved word. *See* DECLARATIVES
Executing COBOL programs, 3.8
EXHIBIT as reserved word, 12.3, 12.51
EXIT as reserved word, 12.3, 12.52
EXIT PROGRAM as reserved word, 12.3, 12.53
Expediting compilation, 4.1
Expression evaluation, precedence for, 12.36
EXTEND as reserved word. *See* DECLARATIVES, OPEN
External decimal item in DATA DIVISION, 8.3

F

FD as reserved word. *See* FILE
Figurative constants in DATA DIVISION, 8.9, 8.11
FILE as reserved word, 12.3, 12.54–12.55. *See also* DATA (division)
FILE-CONTROL as reserved word. *See* INPUT-OUTPUT
FILE-CONTROL paragraph, 12.65
FILE-ID as reserved word. *See* VALUE (OF FILE-ID)
Files
   batch command, 4.1
   source listing, 2.7
FILE STATUS
   clause with indexed files, 10.2
   table of, 10.3
FILLER as reserved word, 12.4
FIRST as reserved word. *See* INSPECT
Fixed segments, 9.5
Floating string with PICTURE, 12.92–12.93
FOOTING as reserved word. *See* LINAGE
FOR as reserved word. *See* INSPECT
FOREGROUND-COLOR as reserved word. *See* SCREEN
FORMAT utility used to back up disks, 1.4
FROM as reserved word. *See* ACCEPT, PERFORM, REWRITE, SCREEN SUBTRACT, WRITE

G

GIVING as reserved word. *See* ADD, SUBTRACT, MULTIPLY, DIVIDE
GO TO as reserved word, 12.4, 12.56
Granules, types of, 5.2
GREATER as reserved word. *See* IF
Group items
   in DATA DIVISION, 8.2
   syntax in, 8.7

H

HIGH-VALUE(S) as reserved word, 12.4

I

I-O
   error handling, 9.4
   as reserved word. *See* DECLARATIVES, OPEN
I-O-CONTROL as reserved word. *See* INPUT-OUTPUT
I-O-CONTROL paragraph in INPUT-OUTPUT, 12.66
IDENTIFICATION as reserved word, 12.4, 12.57
IF
   class-test with, 12.61
   condition-name with, 12.61
   nesting ELSE statements with, 12.59
   relation-test with, 12.60
   as reserved word, 12.4, 12.58–12.62
   sign-test with, 12.61
Imperative statements in PROCEDURE DIVISION, 9.1

INDEX

IN, OF as reserved word, 12.4, 12.63
Independent segments, 9.5
INDEX as reserved word. *See* USAGE
Index data item, restrictions for. with, USAGE 12.123
INDEXED as reserved word. *See* INPUT-OUTPUT, OCCURS
Indexed file recovery utility, D.1–D.7
Indexed files, 10.1–10.7
  DELETE statement with, 10.6–10.7
  FILE STATUS clause with, 10.2
  OPEN statement with, 10.3
  READ statement with, 10.4
  RECORD KEY clause with, 10.1-10.2
  REWRITE statement with, 10.6
  SELECT clause with, 10.1
  START statement with, 10.7
  syntax in ENVIRONMENT DIVISION with 10.1-10.3
  syntax in PROCEDURE DIVISION with 10.3–10.7
  WRITE statement with, 10.5–10.6
Indexes
  rules applied to values of, with SEARCH, 12.108
  use of, with OCCURS, 12.80
INITIAL as reserved word, 12.4
Interactive debug facility, 6.1–6.3
Internal storage formats with numeric items, 8.3
Interprogram communication with CALL or CHAIN, A.1–A.9
INPUT as reserved word. *See* DECLARATIVES, OPEN
INPUT-OUTPUT
  FILE-CONTROL paragraph in, 12.66
  I-O-CONTROL paragraph in, 12.66
  as reserved word, 12.4, 12.64–12.66. *See also* ENVIRONMENT
INSPECT as reserved word, 12.4, 12.67–12.69
INSTALLATION as reserved word. *See* IDENTIFICATION
Internal decimal item in DATA DIVISION, 8.3–8.4
INTO as reserved word. *See* DIVIDE, READ, STRING, UNSTRING
INVALID as reserved word, 12.4
IS as reserved word. *See* CONFIGURATION

J
JUST as reserved word. *See* SCREEN
JUSTIFIED as reserved word, 12.4, 12.70. *See also* SCREEN

K
KEY as reserved word. *See* ACCEPT, OCCURS
Key files, 5.2

L
LABEL as reserved word. *See* FILE
LEADING as reserved word. *See* INSPECT, SIGN
LEFT-JUSTIFY as reserved word. *See* ACCEPT
LENGTH-CHECK as reserved word. *See* ACCEPT
LESS as reserved word. *See* IF
Level numbers, subdividing records of, 8.4–8.6
LIN as reserved word. *See* ACCEPT, DISPLAY, EXHIBIT
LINAGE as reserved word, 12.4, 12.71–12.72. *See* also DATA (division)
LINAGE-COUNTER as reserved word. *See* LINAGE
LINE as reserved word. *See* ACCEPT, INPUT-OUTPUT, SCREEN, WRITE
LINES as reserved word. *See* WRITE, LINAGE
LINK, using, 3.1–3.5
LINKAGE as reserved word, 12.4, 12.73. *See* also DATA (division)
Linking
  large programs, 3.7
  program modules, 3.6
  programs that use overlays, 3.6
Literals
  in DATA DIVISION, 8.8–8.11
  use of, 12.46 LOCK as reserved word. *See* CLOSE
LOW-VALUE(S) as reserved word, 12.4

M
Methods for separating files with COBOL programs, 3.7
MEMORY as reserved word. *See* CONFIGURATION
MODE as reserved word. *See* CONFIGURATION
MOVE
  as reserved word, 12.5, 12.74–12.76
  rules for using, 12.74–12.75
  tables of, 12.76
MULTIPLY as reserved word, 12.5, 12.77–12.78

N
NATIVE as reserved word. *See* CONFIGURATION
NEGATIVE as reserved word. *See* IF
Nesting IF-ELSE statements with IF, 12.59
NEXT as reserved word. *See* IF, SEARCH
Non-numeric literals
  on continuation lines, 8.10
  in DATA DIVISION, 8.9–8.10

INDEX

NOT as reserved word. *See* IF
NUMBER as reserved word. *See* ACCEPT, SCREEN
NUMERIC as reserved word. *See* IF
Numeric items
  in DATA DIVISION, 8.3, 8.8
  with PICTURE, 12.89
Numeric-edited items in DATA DIVISION, 8.3, 8.9
Numeric literals in DATA DIVISION, 8.10–8.11

O

OBJECT-COMPUTER as reserved word. *See* CONFIGURATION
OCCURS
  as reserved word, 12.5, 12.79–12.82
  use of indexes with, 12.80
  use of subscripts with, 12.79
OF as reserved word. *See* IF
OMITTED as reserved word. *See* FILE
OPEN
  as reserved word, 12.5, 12.83–12.84
  statement with indexed files, 10.3–10.4
Operating system I/O errors, E.1, E.2–E.3
Operating the compiler, 2.1–2.4
  partial command strings in, 2.3–2.4
Optional clauses, effects of, 12.103
OR as reserved word. *See* UNSTRING
ORGANIZATION as reserved word. *See* INPUT-OUTPUT
Organization of the PROCEDURE DIVISION, 9.2–9.3
OUTPUT as reserved word. *See* DECLARATIVES, OPEN
OVERFLOW as reserved word. *See* STRING, UNSTRING

P

PAGE as reserved word. *See* WRITE
Paragraphs in statement structures, 9.2
Passing data items in PROCEDURE DIVISION header, 9.3
PERFORM as reserved word, 12.5, 12.85–12.88
Phases of the compiler, C.1
PIC as reserved word. *See* SCREEN
PICTURE
  alphanumeric items with, 12.89
  editing DATA with, 12.95
  numeric items with, 12.89
  floating string with, 12.92
  report-form option with, 12.91–12.94
  as reserved word, 12.5, 12.89–12.95. *See* also SCREEN
PLUS as reserved word. *See* SCREEN
POINTER as reserved word. *See* STRING, UNSTRING
POSITIVE as reserved word. *See* IF
Precedence for evaluating the expression with COMPUTE, 12.36
PRINTER as reserved word. *See* CONFIGURATION, INPUT-OUTPUT
PROCEDURE as reserved word, 12.5, 12.96. *See* also DECLARATIVES
PROCEDURE DIVISION, 9.1–9.6
  design of, 9.2–9.6
  segmentation facility in, 9.5
  syntax in, with indexed files, 10.3–10.7
  syntax in, with relative files, 11.2–11.5
  types of statements in, 9.1–9.2
Program
  load errors, E.2, E.17
  syntax errors, E.1, E.4–E.14
PROGRAM as reserved word. *See* EXIT (PROGRAM)
PROGRAM-ID as reserved word. *See* IDENTIFICATION
PROMPT as reserved word. *See* ACCEPT
Punctuation of source code, 7.5

Q

QUOTE(S) as reserved word, 12.5

R

RANDOM as reserved word. *See* INPUT-OUTPUT
READ
  as reserved word (to perform sequential input), 12.5, 12.97
  statement with indexed files, 10.4–10.5
  statement with relative files, 11.2–11.3
READY as reserved word. *See* TRACE
REBUILD utility
  to recover or restore information, D.1–D.7
  running, D.2–D.6
RECORD
  KEY clause with indexed files, 10.1–10.2
  as reserved word, 12.5, 12.98. *See* also DATA (RECORD clause), FILE, INPUT-OUTPUT, READ
RECORDS as reserved word. *See* BLOCK, DATA (RECORD clause), FILE
Recovering information with REBUILD, D.1–D.7
REDEFINES as reserved word, 12.5, 12.99–12.100
Relation-test to evaluate operands, 12.60

INDEX

RELATIVE
  KEY clause with relative files, 11.2
  as reserved word. *See* INPUT-OUTPUT
Relative files, 11.1–11.5
  DELETE statement with, 11.4
  READ statement with, 11.2–11.3
  RELATIVE KEY clause with, 11.2
  REWRITE statement with, 11.4
  SELECT clause with, 11.1
  START statement with, 11.5
  syntax in ENVIRONMENT DIVISION with, 11.1–11.2
  syntax in PROCEDURE DIVISION with, 11.2–11.5
  WRITE statement with, 11.3
REPLACING as reserved word. *See* INSPECT
Report-form option with PICTURE, 12.91–12.94
RESERVE as reserved word. *See* INPUT-OUTPUT
Reserved words, 12.1–12.132
  ACCEPT as, 12.2, 12.8–12.24
  ACCESS as, 12.2. *See also* INPUT-OUTPUT
  ADD as, 12.2, 12.25–12.26
  AFTER as, 12.2. *See also* DECLARATIVES, INSPECT, PERFORM, WRITE
  ALL as, 12.2. *See also* INSPECT, SEARCH, UNSTRING,
  ALPHABETIC as, 12.2. *See also* IF
  ALTER as, 12.2, 12.27
  AND as, 12.2. *See also* IF
  AREA as, 12.2. *See also* INPUT/OUTPUT
  ASCENDING as, 12.2. *See also* OCCURS, SEARCH
  ASCII as, 12.2. *See also* CODE-SET, CONFIGURATION
  ASSIGN as, 12.2. *See also* INPUT-OUTPUT
  AUTHOR as, 12.2. *See also* IDENTIFICATION
  AUTO as, 12.2. *See also* SCREEN
  AUTO-SKIP as, 12.2. *See also* ACCEPT
  BACKGROUND-COLOR as, 12.2. *See also* SCREEN
  BEEP as, 12.2. *See also* ACCEPT
  BEFORE as, 12.2. *See also* INSPECT, WRITE
  BELL as, 12.2. *See also* SCREEN
  BLANK as, 12.2, 12.28. *See also* SCREEN
  BLOCK as, 12.2, 12.29
  BOTTOM as, 12.2. *See also* LINAGE
  BY as, 12.2. *See also* DIVIDE, INSPECT, MULTIPLY, PERFORM, SET
  CALL as, 12.2, 12.30–12.31
  CHAIN as, 12.2, 12.32
  CHAINING as, 12.2. *See also* PROCEDURE
  CHARACTERS as, 12.2. *See also* CONFIGURATION, INSPECT, BLOCK, RECORD
  CLOSE as, 12.2, 12.33
  CODE-SET as, 12.2, 12.34
  COL as, 12.2. *See also* ACCEPT, DISPLAY, EXHIBIT
  COLUMN as, 12.2. *See also*, SCREEN
  COMMA as, 12.2. *See also* CONFIGURATION
  COMPUTATIONAL as, 12.2. *See also* USAGE
  COMPUTATIONAL-3 as, 12.2. *See also* USAGE
  COMPUTE as, 12.3, 12.35–12.37
  CONFIGURATION as, 12.3, 12.38–12.39. *See also* ENVIRONMENT
  CONTAINS as, 12.3. *See also* BLOCK, RECORD
  COPY as, 12.3, 12.40
  COUNT as, 12.3. *See also* IF, UNSTRING
  CURRENCY as, 12.3. *See also* CONFIGURATION
  DATA (in DATA DIVISION header) as, 12.3, 12.41
  DATA (in DATA RECORD clause) as, 12.3, 12.42
  DATE as, 12.3. *See also* ACCEPT
  DATE-COMPILED as, 12.3. *See also* IDENTIFICATION
  DATE-WRITTEN as, 12.3. *See also* IDENTIFICATION
  DAY as, 12.3. *See also* ACCEPT
  DEBUGGING as, 12.3. *See also* CONFIGURATION
  DECIMAL-POINT as, 12.3. *See also* CONFIGURATION
  DECLARATIVES as, 12.3, 12.43–12.44. *See also* PROCEDURE
  DELETE as, 12.3
  DELIMITED as, 12.3. *See also* STRING, UNSTRING
  DELIMITER as, 12.3. *See also* UNSTRING
  DEPENDING as, 12.3. *See also* GO
  DESCENDING as, 12.3. *See also* OCCURS, SEARCH
  DISK as, 12.3. *See also* INPUT-OUTPUT
  DISPLAY as, 12.3, 12.45–12.47. *See also* USAGE
  DIVIDE as, 12.3, 12.48–12.49
  DIVISION as, 12.3. *See also*
  DIVISION as, 12.3. *See also* DATA (division), ENVIRONMENT, IDENTIFICATION, PROCEDURE
  DOWN as, 12.3. *See also* SET
  DYNAMIC as, 12.3. *See also* INPUT-OUTPUT
  ELSE as, 12.3. *See also* IF
  END as, 12.3. *See also* DECLARATIVES, READ, SEARCH
  END-OF-PAGE as, 12.3. *See also* WRITE

INDEX

ENVIRONMENT as, 12.3, 12.50
EOP as, 12.3. *See also* WRITE
EQUAL as, 12.3. *See also* IF
ERASE as, 12.3. *See also* DISPLAY, EXHIBIT
ERROR as, 12.3. *See also* ADD, COMPUTE, DECLARATIVES, DIVIDE, MULTIPLY, SUBTRACT
ESCAPE as, 12.3. *See also* ACCEPT
EXCEPTION as, 12.3. *See also* DECLARATIVES
EXHIBIT as, 12.3, 12.51
EXIT as, 12.3, 12.52
EXIT PROGRAM as, 12.3, 12.53
EXTEND as, 12.3. *See also* DECLARATIVES, OPEN
FD as, 12.3. *See also* FILE
FILE as, 12.3, 12.54-12.55. *See also* DATA (division)
FILE-CONTROL as, 12.3. *See also* INPUT-OUTPUT
FILE-ID as, 12.4. *See also* VALUE (OF FILE-ID)
FILLER as, 12.4
FIRST as, 12.4. *See also* INSPECT
FOOTING as, 12.4. *See also* LINAGE
FOR as, 12.4. *See also* INSPECT
FOREGROUND-COLOR as, 12.4. *See also* SCREEN
FROM as, 12.4. *See also* ACCEPT, PERFORM, REWRITE, SCREEN, SUBTRACT, WRITE
GIVING as, 12.4. *See also* ADD, SUBTRACT, MULTIPLY, DIVIDE
GO as, 12.4, 12.56
GREATER as, 12.4. *See also* IF
HIGH-VALUE(S) as, 12.4.
I-O as, 12.4. *See also* DECLARATIVES, OPEN
I-O-CONTROL as, 12.4. *See also* INPUT-OUTPUT
IDENTIFICATION as, 12.4, 12.57
IF as, 12.4, 12.58–12.62
IN, OF as, 12.4, 12.63
INDEX as, 12.4. *See also* USAGE
INDEXED as, 12.4. *See also* INPUT-OUTPUT, OCCURS
INITIAL as, 12.4
INPUT as, 12.4. *See also* DECLARATIVES, OPEN
INPUT-OUTPUT as, 12.4, 12.64–12.66. *See also* ENVIRONMENT INSPECT as, 12.4, 12.67–12.69
INSTALLATION as, 12.4. *See also* IDENTIFICATION
INTO as, 12.4. *See also* DIVIDE, READ, STRING, UNSTRING
INVALID as, 12.4
IS as, 12.4. *See also* CONFIGURATION
JUST as, 12.4. *See also* SCREEN
JUSTIFIED as, 12.4, 12.70. *See also* SCREEN
KEY as, 12.4. *See also* ACCEPT, OCCURS
LABEL as, 12.4. *See also* FILE
LEADING as, 12.4. *See also* INSPECT, SIGN
LEFT-JUSTIFY as, 12.4. *See also* ACCEPT
LENGTH-CHECK as, 12.4. *See also* ACCEPT
LESS as, 12.4. *See also* IF
LIN as, 12.4. *See also* ACCEPT, DISPLAY, EXHIBIT
LINAGE as, 12.4, 12.71-12.72. *See also* DATA (division)
LINAGE-COUNTER as, 12.4. *See also* LINAGE
LINE as, 12.4. *See also* ACCEPT, INPUT-OUTPUT, SCREEN, WRITE
LINES as, 12.4. *See also* WRITE, LINAGE
LINKAGE as, 12.4, 12.73. *See also* DATA (division)
LOCK as, 12.4. *See also* CLOSE
LOW-VALUE(S) as, 12.4
MEMORY as, 12.4. *See also* CONFIGURATION
MODE as, 12.5. *See also* CONFIGURATION
MOVE as, 12.5, 12.74–12.76
MULTIPLY as, 12.5, 12.77–12.78
NATIVE as, 12.5. *See also*
NEGATIVE as, 12.5. *See also* IF
NEXT as, 12.5. *See also* IF, SEARCH
NOT as, 12.5. *See also* IF
NUMBER as, 12.5. *See also* ACCEPT, SCREEN
NUMERIC as, 12.5. *See also* IF
OBJECT-COMPUTER as, 12.5. *See also* CONFIGURATION
OCCURS as, 12.5, 12.79–12.82
OF as, 12.5. *See also* IF
OMITTED as, 12.5. *See also* FILE
OPEN as, 12.5, 12.83–12.84
OR as, 12.5. *See also*UNSTRING
ORGANIZATION as, 12.5. *See also* INPUT-OUTPUT
OUTPUT as, 12.5. *See also* DECLARATIVES, OPEN
OVERFLOW as, 12.5. *See also* STRING, UNSTRING
PAGE as, 12.5. *See also* WRITE
PERFORM as, 12.5, 12.85–12.88
PIC as, 12.5. *See also* SCREEN
PICTURE as, 12.5, 12.89–12.95. *See also* SCREEN
PLUS as, 12.5. *See also* SCREEN
POINTER as, 12.5. *See also* STRING, UNSTRING
POSITIVE as, 12.5. *See also* IF
PRINTER as, 12.5. *See also* CONFIGURATION, INPUT-OUTPUT
PROCEDURE as, 12.5, 12.96. *See also* DECLARATIVES
PROGRAM as, 12.5. *See also* EXIT (PROGRAM)

INDEX

PROGRAM-ID as, 12.5. *See also* IDENTIFICATION
PROMPT as, 12.5. *See also* ACCEPT
QUOTE(S) as, 12.5.
RANDOM as, 12.5. *See also* INPUT-OUTPUT
READ (to Perform Sequential Input) as, 12.5, 12.97
READY as, 12.5. *See also* TRACE
RECORD as, 12.5, 12.98. *See also* DATA (RECORD clause), FILE, INPUT-OUTPUT, READ
RECORDS as, 12.5. *See also* BLOCK, DATA (RECORD clause), FILE
REDEFINES as, 12.5, 12.99–12.100
RELATIVE as, 12.5. *See also* INPUT-OUTPUT
REPLACING as, 12.5. *See also* INSPECT
RESERVE as, 12.5. *See also* INPUT OUTPUT
RESET as, 12.5. *See also* TRACE
REVERSE-VIDEO as, 12.5. *See also* SCREEN
REWRITE (to Perform Sequential I/O) as, 12.5, 12.101
RIGHT-JUSTIFY as, 12.5. *See also* ACCEPT
ROUNDED as, 12.6. *See also* ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT
RUN as, 12.6. *See also* STOP
SAME as, 12.6. *See also* INPUT-OUTPUT
SCREEN as, 12.6, 12.102–12.106. *See also* DATA (division)
SEARCH as, 12.6, 12.107–12.110
SECTION as, 12.6. *See also* CONFIGURATION, DATA (division), DECLARATIVES, ENVIRONMENT, FILE, INPUT-OUTPUT, LINKAGE, PROCEDURE, SCREEN, WORKING-STORAGE
SECURE as, 12.6. *See also* SCREEN
SECURITY as, 12.6. *See also* IDENTIFICATION
SELECT as, 12.6. *See also* INPUT-OUTPUT
SENTENCE as, 12.6. *See also* IF, SEARCH
SEPARATE as, 12.6. *See also* SIGN
SEQUENCE as, 12.6. *See also* CONFIGURATION
SEQUENTIAL as, 12.6. *See also* INPUT-OUTPUT
SET as, 12.6, 12.111
SIGN as, 12.6, 12.112
SIZE as, 12.6. *See also* ADD, COMPUTE, DIVIDE, MULTIPLY, STRING, SUBTRACT
SORT as, 12.6. *See also* INPUT-OUTPUT
SORT-MERGE as, 12.6. *See also* INPUT-OUTPUT
SOURCE-COMPUTER as, 12.6. *See also* CONFIGURATION
SPACE(S) as, 12.6
SPACE-FILL as, 12.6. *See also* ACCEPT
SPECIAL-NAMES as, 12.6. *See also* CONFIGURATION
STANDARD as, 12.6. *See also* FILE
STANDARD 1 as, 12.6. *See also* CONFIGURATION
START as, 12.6
STATUS as, 12.6. *See also* INPUT-OUTPUT
STOP as, 12.6, 12.113
STRING as, 12.6, 12.114–12.115
SUBTRACT as, 12.6, 12.116–12.117
SYNC as, 12.6
SYNCHRONIZED as, 12.6, 12.118
table of, 12.2–12.7
TALLYING as, 12.6. *See also* INSPECT, UNSTRING
THROUGH as, 12.6. *See also* PERFORM, VALUE (in condition-names)
THRU as, 12.6. *See also* PERFORM, VALUE (in condition-names)
TIME as, 12.6. *See also* ACCEPT
TIMES as, 12.6. *See also* PERFORM
TO as, 12.6. *See also* ADD, ALTER, MOVE, RECORD, SCREEN, SET
TOP as, 12.6. *See also* LINAGE
TRACE as, 12.6, 12.119
TRAILING as, 12.6. *See also* SIGN
TRAILING-SIGN as, 12.6. *See also* ACCEPT
UNSTRING as, 12.6, 12.120–12.122
UNTIL as, 12.6. *See also* PERFORM
UP as, 12.7. *See also* SET UPDATE, ACCEPT
USAGE as, 12.7, 12.123–12.124
USE as, 12.7. *See also* DECLARATIVES
USING as, 12.7. *See also* CALL, CHAIN, PROCEDURE, SCREEN
VALUE (to define truth set of condition-name) as, 12.7, 12.125–12.126
VALUE (to initialize data value) as, 12.7, 12.127–12.128. *See also* SCREEN
VALUE (to specify a disk filename) as, 12.7, 12.129
VALUES as, 12.7. *See also* VALUE (in condition-names)
VARYING as, 12.7. *See also* PERFORM, SEARCH
WHEN as, 12.7. *See also* SEARCH
WITH as, 12.7. *See also* ACCEPT, CLOSE, CONFIGURATION
WORDS as, 12.7. *See also* CONFIGURATION
WORKING-STORAGE as, 12.7, 12.130. *See also* DATA (division)
WRITE (to Perform Sequential Output) as, 12.7, 12.131–12.132
ZERO (ZEROS, ZEROES) as, 12.7. *See also* BLANK, IF, SCREEN

INDEX

ZERO-FILL as, 12.7. *See also* ACCEPT
RESET as reserved word. *See* TRACE
Restoring information with REBUILD, D.1–D.7
Restrictions for index data item with USAGE, 12.123
REVERSE-VIDEO as reserved word. *See* SCREEN
REWRITE
as reserved word (to perform sequential I/O), 12.5, 12.101
statement with indexed files, 10.6
statement with relative files, 11.4
RIGHT-JUSTIFY as reserved word. *See* ACCEPT
ROUNDED as reserved word. *See* ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT
Rules for using MOVE, 12.74–12.75
RUN as reserved word. *See* STOP
Runtime errors, E.1, E.15–E.17
Runtime executor with COBOL programs, 3.8

S

SAME as reserved word. *See* INPUT-OUTPUT SCREEN
effects of optional clauses with, 12.103–12.105
as reserved word, 12.6, 12.102–12.106. *See* also DATA (division)
use of levels in, 12.103
Screen-name, use of, 12.47
SEARCH
as reserved word, 12.6, 12.107–12.110
rules of index values with, 12.108
rules of WHEN condition with, 12.108–12.109
SECTION as reserved word. *See* CONFIGURATION, DATA (division), DECLARATIVES, ENVIRONMENT, FILE, INPUT-OUTPUT, LINKAGE, PROCEDURE, SCREEN, WORKING-STORAGE
Sections in statement structures, 9.2
SECURE as reserved word. *See* SCREEN
SECURITY as reserved word. *See* IDENTIFICATION
Segmentation facility
to enable execution, 9.5
range for segment-number with, 9.5
SELECT
clause with indexed files, 10.1
clause with relative files, 11.1
as reserved word, 12.6. *See also* INPUT-OUTPUT
SENTENCE as reserved word. *See* IF, SEARCH
Sentences in statement structures, 9.2
SEPARATE as reserved word. *See* SIGN
SEQUENCE as reserved word. *See* CONFIGURATION
SEQUENTIAL as reserved word. *See* INPUT-OUTPUT
SET as reserved word, 12.6, 12.111
SIGN as reserved word, 12.6, 12.112
Sign-test with IF, 12.61
SIZE as reserved word. *See* ADD, COMPUTE, DIVIDE, MULTIPLY, STRING, SUBTRACT
Size limitations in DATA DIVISION, 8.12
/ used to indicate switch, 2.5–2.6
SORT as reserved word. *See* INPUT-OUTPUT
SORT-MERGE as reserved word. *See* INPUT-OUTPUT
SOURCE-COMPUTER as reserved word. *See* CONFIGURATION
Source listing file, 2.7
Source program, changing tab stops in, B.1
SPACE(S) as reserved word, 12.6
SPACE-FILL as reserved word. *See* ACCEPT
SPECIAL-NAMES as reserved word. *See* CONFIGURATION
Specifying the file organization with data input and output, 5.1
Stand-alone items in DATA DIVISION, 8.2
STANDARD as reserved word. *See* FILE
STANDARD-1 as reserved word. *See* CONFIGURATION
START
as reserved word, 12.6
statement with indexed files, 10.7
statement with relative files, 11.5
Statements, types of, 9.1–9.2
STATUS as reserved word. *See* INPUT-OUTPUT
Steps in the compilation process, 1.1–1.2
STOP as reserved word, 12.6, 12.113
STRING as reserved word, 12.6, 12.114–12.115
Subscripts, use of, 12.79. *See also* OCCURS
SUBTRACT as reserved word, 12.6, 12.116–12.117
Switches, 2.5–2.6, 2.7
SYNC as reserved word, 12.6
SYNCHRONIZED as reserved word, 12.6, 12.118

T

Tab stops changed in source program, B.1
TALLYING as reserved word. *See* INSPECT, UNSTRING
Terminator keys, 12.14
THROUGH as reserved word. *See* PERFORM, VALUE (in condition-names)
THRU as reserved word. *See* PERFORM, VALUE (in condition-names)
TIME as reserved word. *See* ACCEPT
TIMES as reserved word. *See* PERFORM
TO as reserved word. *See* ADD, ALTER, MOVE, RECORD, SCREEN, SET

INDEX

TOP as reserved word. *See* LINAGE
TRACE as reserved word, 12.6, 12.119
TRAILING as reserved word. *See* SIGN
TRAILING-SIGN as reserved word. *See* ACCEPT

U
UNSTRING as reserved word, 12.6, 12.120–12.122
UNTIL as reserved word. *See* PERFORM
UP as reserved word. *See* SET UPDATE, ACCEPT
USAGE
  as reserved word, 12.7, 12.123–12.124
  restrictions for index data item with, 12.123
USE as reserved word. *See* DECLARATIVES
USING as reserved word. *See* CALL, CHAIN, PROCEDURE, SCREEN

V
VALUE as reserved word
  (in condition-names), 12.7, 12.125–12.126
  (of file ID), 12.7, 12.129
  (to initialize data value), 12.7, 12.127–12.128
VALUES as reserved word. *See* VALUE (in condition-names)
VARYING as reserved word. *See* PERFORM, SEARCH

W
WHEN
  as reserved word. *See* SEARCH
  rules of, with SEARCH, 12.108–12.109
WITH as reserved word. *See* ACCEPT, CLOSE, CONFIGURATION
WORDS as reserved word. *See* CONFIGURATION
WORKING-STORAGE as reserved word, 12.7, 12.130. *See* also DATA (division)
WRITE
  as reserved word (to perform sequential output), 12.7, 12.131–12.132
  statement with indexed files, 10.5–10.6
  statement with relative files, 11.3

Z
Z-DOS
  backing up disks with, 1.4
  used with data input and output, 5.4
ZERO (ZEROS, ZEROES) as reserved word. *See* BLANK, IF, SCREEN
ZERO-FILL as reserved word. *See* ACCEPT

ZENITH data systems

HEATH

HILLTOP ROAD, ST. JOSEPH, MICHIGAN 49085

Dear Customer,

The second distribution disk contains several additional files that are not described in the COBOL-86 manual.

MKDEMO.BAT is a batch file that will compile and link the demonstration program set. The batch routine assumes that you have a two-drive 5.25-inch floppy disk system where the COBOL-86 system disk is in drive A and the program disk is in drive B. If you are using higher capacity disks (i.e., 8-inch or the Winchester hard disk), the COBOL compiler, linker, and runtime files may fit on one disk. If this is the case, you may use the following procedure to copy the distribution disk(s) and then compile, link, and run the demonstration program set.

1. Select a formatted Winchester partition (or other high capacity disk) that contains Z-DOS as the default disk by entering the drive name and then pressing the RETURN key. For instance, if you selected the first Winchester partition, enter **E:** and then press **RETURN**.

2. Place the COBOL distribution disks one at a time in drive A and copy each by entering **COPY A:*.***.

3. If the program, LINK.EXE, is not on the default disk, copy it from the Z-DOS distribution disk by replacing the COBOL distribution disk with the Z-DOS distribution disk I and entering **COPY A:LINK.EXE** and pressing **RETURN**.

4. Compile the demonstration program set by entering each of the following commands and pressing **RETURN**.

```
COBOL DEMO,,CON;
COBOL BUILD,,CON;
COBOL UPDATE,,CON;
LINK DEMO+BUILD;
LINK UPDATE;
```

5. To run the program, enter **DEMO** and press **RETURN**.

The second disk also contains five sample programs that demonstrate the five types of file organization supported by COBOL-86. You should study these examples to become familiar with the sequence of operations that may be used for each file type. The programs are:

EX-I1.COB—Indexed files
EX-R1.COB—Relative files
EX-S1.COB—Sequential files
EX-L1.COB—Line sequential files
EX-P1.COB—Printer output samples

Note to Users of Previous Versions of Microsoft™ COBOL

There are three differences between COBOL-86 and previous versions of Microsoft COBOL. They are described in the following paragraphs along with revisions necessary so that the source code may be modified to be compatible with the current version of Microsoft COBOL.

First, in previous versions, data items described as COMP or COMPUTATIONAL were treated as 16-bit binary fields. In the present version, these must be described as COMP-0 or COMPUTATIONAL-0. Items described as COMP or COMPUTATIONAL will be stored as ASCII characters in the present version.

Second, there is a change in the default tab stops. Previous versions used 7, 17, 25, 33, 41, 49, 57, 65, and 73. The current version uses 8, 12, 20, 28, 36, 44, 52, 60, 68, and 73. Either you must modify the source programs that used different tab stops and recompile them or modify the COBOL-86 tab stop table. To modify the tab stop table, see Appendix B, "Customizations."

The third difference is in the format of index files which was changed between versions 4.01 and 4.60 of COBOL-80. If you have programs that were written under COBOL-80 version 4.01 and want to convert those indexed files to the format used by COBOL-86, you must write two programs for each index file. The first program, which must be written in COBOL-80 under CP/M, converts the COBOL-80 (version 4.01) index file into a sequential file. It does this by reading all the records of the index file and then writing them as a sequential file. Once that is done, you can move the sequential file to Z-DOS (MS-DOS) with the RDCPM utility (described in your Z-DOS manual). The second program, written in COBOL-86, converts the transferred sequential file back into an index file. It does this by reading the sequential file information into memory and then writing it out as an indexed file.

™ Microsoft is a trademark of Microsoft Corporation.