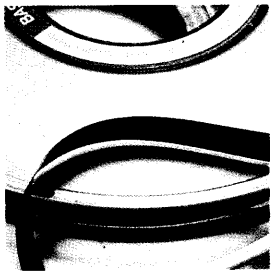
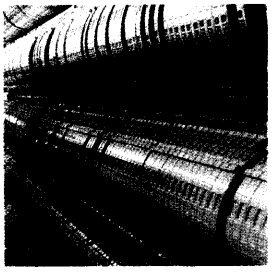
Prime Computer, Inc.
Reference Guide FDR3056-101A

COBOL

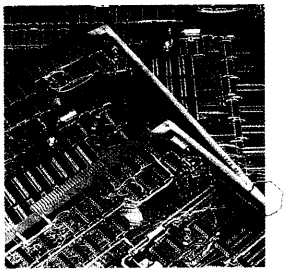













The COBOL Reference Guide

Published by Prime Computer, Inc.
Technical Publications Department
500 Old Connecticut Path
Framingham, Massachusetts 01701

First printing, January 1980

Production information: This book was composed in 10 and 11 point Melior and 10 point Helvetica by Allied Systems. The covers were printed in 6 colors by MacDonald & Evans with separations by Spectrum. The cover stock was 100# Warren LOE Gloss Cover. The text was printed in 2 colors by Federated Lithographers. The text stock was 50# Mohawk Vellum, Creme White. Layout and design was by William Agush of Prime Computer.

The COBOL Reference Guide

by Grace T. Na

CONTENTS

7 DATA DIVISION

8 PROCEDURE DIVISION

ACKNOWLEDGMENT

The following acknowledgment is a reprint from the American National Standard Programming Language COBOL, ANSI X3.23-1974:

"Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document 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 the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention 'COBOL' in acknowledgment of the source, but need not quote the acknowledgment):

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 CODASYL Programming Language 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.

The authors and copyright holders of the copyrighted material used herein

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

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

I

OVERVIEW

1

Overview of Prime's COBOL

THIS DOCUMENT

Purpose and audience

The COBOL Reference Guide is a Final Documentation Release at software revision level 17 (Rev. 17). This document and a companion document, The Prime User's Guide, replace the COBOL Programmer's Guide PDR3056 Rev. B.

This document fully describes Prime COBOL, and provides the necessary information for compiling, loading, executing and debugging COBOL programs on a Prime system. It is designed to be used as a reference guide for an experienced COBOL programmer. Users unfamiliar with the language should read one of the many commercially available instruction books; examples are:

Feingold, Carl, Fundamentals of Structured COBOL programming, WM. C. Brown Company Publishers

Stern, N. and Stern, R., COBOL Programming, John Wiley and Son, Inc.

Organization and usage

This document has three major parts:

Part one — **Overview**. Introduces Prime's COBOL, including Prime extensions to the language, supporting utilities, systems and software (Section 1).

Part two — **Language-Specific System Information**. Provides complete information on the use of the COBOL compiler (Section 2), and describes the process of loading and executing COBOL programs (Section 3).

Part three — **Language Reference**. Provides syntactical and general COBOL specifications, patterned after the ANSI standard. The three main sub-divisions are:

Fundamental Concepts of COBOL (Section 4)
Nucleus (Sections 5-8)
Functional Processing Modules (Sections 9-13)

Fundamental Concepts of COBOL defines the Nucleus and Functional Processing Modules. The Nucleus presents the structure and governing rules of COBOL's four divisions: Identification, Environment, Data and Procedure. The Functional Processing Modules include Inter-program Communication, Table Handling, Sort, Indexed I/O, Relative I/O, Sequential I/O, and Library.

Effective usage of the Language Reference sections requires knowledge of its organization:

- Fundamental Concepts begins with a generalized COBOL program summary. This is expanded in the sample listing file, SAMPLE. Fundamental COBOL concepts, including

standard format notation, punctuation rules, etc. are here set forth.

- The Nucleus expands upon the previous presentation of Fundamental Concepts. It provides detailed information related to the Identification, Environment, Data, and Procedure Divisions.

  PROCEDURE DIVISION (Section 8) presents COBOL verbs alphabetically. A quick verb index is in Appendix C.

  Each division section closes with an example of source coding for that given division. These examples form a functional program, REF2, which illustrates the interrelationship of component parts.
- Functional Processing Modules are self-contained, often restating concepts, data descriptions, and COBOL statement formats elsewhere described. The reader will find here all related data in a single location for maximum utility and efficiency. For example, the READ verb is presented in the Procedure Division. It is restated in the Indexed I/O Functional Processing Module, together with related data pertinent to Indexed I/O processing.

In addition to the body of text, the Table of Contents is a guide to content and order; the index provides the most direct access to specifics. Appendices present a capsule form of repeatedly used data as follows:

- The file organization
- Two sets of typical CREATK dialog for INDEXED and SAM files, along with examples
- Important reference tables: COBOL Verb Index, COBOL Reserved Words, ASCII Character Set, File Status Key Definitions, Permissible I/O Statements, Permissible Moves, and Numeric Conversion Tables.
- A list of COBOL punctuation, arithmetic and edit symbols
- A list of compile-time and run-time error messages and their meanings
- An introduction to expanded listing for COBOL programs
- The LABEL command for magnetic tapes
- A list of system files required by COBOL

RELATED DOCUMENT

The Prime User's Guide describes all supporting PRIMOS utilities for programming in Prime COBOL or any other Prime language. The COBOL Reference Guide and The Prime User's Guide are complementary documents: both are essential to the COBOL programmer.

LANGUAGE SPECIFICATIONS

Prime COBOL is based upon American National Standard Programming Language COBOL, X3.23-1974. Elements of the COBOL language are allocated to the following 12 different functional processing "modules": Nucleus, Table Handling, Sequential I/O, Relative I/O, Indexed I/O, Sort-Merge, Report Writer, Segmentation, Library, Debug, Inter-Program Communication, and Communication.

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

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

The following summary specifies the content of the eight modules supported by Prime COBOL with respect to the Standard.

Module	Features Available in Prime COBOL
Nucleus	Full level 1, plus these features of level 2: • Levels 01-49, 77; • Level 66 with RENAMES clause permits alternate, possibly overlapping or regrouping of elementary items; • Value series or range for level 88 conditions; • AND OR NOT = < > in conditions; • **IF** statements; • Procedure-names consisting of digits only; • **PERFORM VARYING** with **AFTER** (up to 3 indexes allowed); • Mnemonic-names for **ACCEPT** or **DISPLAY** devices; • Qualification of Names (Procedure Division); • Sign test; • **STRING**; • **UNSTRING**; • **COMPUTE** with multiple receiving fields; • **CORRESPONDING** operations for **MOVE, ADD, SUBTRACT**; • **ACCEPT** { **DAY** / **TIME** / DATE }; • **ADD** with TO identifier, and **GIVING** identifier; • **SUBTRACT** with **FROM** identifier, and **GIVING** identifier; • **MULTIPLY** with **GIVING** identifier, and **BY** identifier; • **DIVIDE** with **INTO** identifier, **BY** identifier, and **GIVING** identifier.
Sequential I/O	Full level 1, plus these features of level 2: • **RESERVE** clause and variable form of **BLOCK;** • **Multple** file-name in **OPEN** and **CLOSE** statements; • **WRITE** statement with **BEFORE/AFTER ADVANCING** identifier **LINES**; • **OPEN EXTEND** for Sequential Disk Files.
Relative I/O	Full level 1, plus these features of level 2: • **RESERVE** clause; • **DYNAMIC** access mode (with READ next); • **START** (with key relations EQUAL, GREATER, or NOT LESS).
Indexed I/O	Full level 1, plus these features of level 2: • **RESERVE** clause; • **DYNAMIC** access (with READ next); • **RANDOM** access mode with READ by KEY;

	• **START** (with key relations EQUAL, GREATER, or NOT LESS); • **ALTERNATE RECORD KEY** clause with **WITH DUPLICATES** phrase (up to 5 additional key fields supported).
Sort	Full level 1 and full level 2 excluding Collating Sequence.
Library	Full level 1, plus these features of level 2: • **COPY text-name OF/IN library-name.**
Table Handling	Full level 1, plus these features of level 2: • **SEARCH;** • **SEARCH ALL.**
Inter-program Communication	Full level 1.

PRIME EXTENSIONS TO THE LEVEL 2 STANDARD

- **ASSEMBLER** (enter assembler);
- **COMP-3**
- **COMPUTATIONAL-3** } (packed decimal format);
- **EXHIBIT NAMED** statement;
- **OWNER IS;**
- **READY TRACE;**
- **RESET TRACE;**
- **REMARKS;**
- **UNCOMPRESSED/COMPRESSED** file format;
- Comprehension Cross Reference Listing.

COBOL UNDER PRIMOS

Implementation

Prime's COBOL runs on Prime models 350 and above, operating under PRIMOS. COBOL runfiles operate in segmented mode (V-MODE or 64V). Code generated in V MODE on the Prime is pure, and is the same for all processor models.

Prime's processors with XIS (Extended Instruction Set hardware) execute an extended set of instructions directly, including decimal arithmetic and character edits. They maximize execution time efficiency. Other processors recognize the code as an unimplemented instruction trap and automatically substitute an equivalent software routine (UII, Unimplemented Instruction package).

Operation

Prime's COBOL operates on an integrated, interactive virtual memory system based on demand paging from disk. It supports up to 63 simultaneous users.

All phases of COBOL compilation can be handled through any of the interactive terminals. Therefore, source programs can be entered and modified directly at a terminal. A COBOL programmer can compile, list, execute, and save his program in a single interactive session. Features such as the interactive text editor enable simplified debugging and enhanced program handling.

The Prime operating system supporting COBOL is called PRIMOS. Only one version of PRIMOS exists for all Prime models. It features paged and segmented virtual memory.

management; it supports up to 63 simultaneous users. The system is based on demand paging from disk with 2048 bytes per page. A page-sharing feature reduces overhead time. For example, several COBOL users may share one copy of the Editor to enter, modify, or debug their programs, rather than individually having their own copy.

Prime's segmentation scheme uses a virtual address consisting of a segment number (one of 4096), a page number, and a word number. The virtual address is translated into a physical address by a series of segment tables and page maps. Paging requirements for the application program are thus met immediately and automatically.

COBOL under PRIMOS has advanced segmentation capability, expanded compilation options, sharable code, and DBMS capability.

Compatibility

Because a common operating system architecture is used throughout the Prime processor line, COBOL programs created on one Prime computer can be used on any larger or smaller Prime computer without modification. Compatibility holds true at both the source level and the memory image level.

PROGRAM ENVIRONMENTS

Under PRIMOS, COBOL programs may execute in one of three environments:

- Interactive
- Phantom user
- Batch job processing

Interactive

Program execution is initiated directly by the user. Programs run in real time and are "connected" to the terminal. Program output is printed at the terminal, as well as user- or system-generated error messages. This environment is the one most often used. Major uses are:

- Program development
- Programs requiring short execution time
- Data entry programs such as order entry, payroll, etc.
- Interactive programs such as the Editor, etc.

Phantom user

The phantom environment allows programs to be executed while "disconnected" from a terminal. This frees the terminal for other uses. Phantom users accept input from a command file instead of a terminal; output directed to a terminal is either ignored or directed to a file.

Major uses of phantoms are:

- Programs requiring long execution time (such as sorts)
- Certain system utilities (such as line printer spooler)
- Freeing terminals for interactive uses

Batch job processing

Since the number of phantom users on a system is limited, phantoms are not always available. The Batch environment allows users to submit non-interactive command files as Batch jobs at any time. The Batch monitor (itself a phantom) queues these jobs and runs them, one to six at a time, as phantoms become free.

SYSTEM RESOURCES SUPPORTING COBOL

Prime COBOL shares equally with all Prime programming languages a broad range of system and file management resources.

Such resources as system libraries, the text editor, or the SEG utility expand the scope and efficiency of Prime's interactive environment.

Compatible file management systems enhance the mixing capabilities of the system while providing standardized file management functions. Files are created and maintained separately from the applications program.

Libraries

The COBOL programmer may find system library functions and subroutines of use in some applications. A list of VCOBLB library subroutines and functions is presented in Appendix H. A complete treatment of all library and system subroutines is in The Subroutine Reference Guide.

Compiler

Prime's COBOL compiler operates on COBOL source code to generate object code. It is also possible to generate a program listing only. Since syntax checking can be achieved in a shorter period of time, this feature can produce a quick and useful reference to the source program. The user has the additional compiler options to control I/O specifications. The compiler is described in detail in Section 2.

SEG utility

SEG is the V-identity program loading and execution utility. It combines separately compiled program modules, subroutines, and libraries into an executable program. Program modules can be up to 64K words long. All memory management, symbol tables, linkages, etc. are handled by SEG's loader. Various types of loadmaps may be obtained. The SEG utility has many functions; they are described as follows:

Normal usage (Section 3)

Advanced usage (LOAD and SEG Reference Guide)

Editor

Prime's text editor is a line-oriented editor enabling the programmer to enter and modify source code and text files. Information for these purposes is in The Prime User's Guide; a complete description of the Editor is in The New User's Guide to EDITOR and.RUNOFF.

Database Management System (DBMS)

Prime's DBMS is a CODASYL-compliant system for management of large amounts of data. DBMS can be accessed from either COBOL or FORTRAN programs. It is compatible with MIDAS and FORMS.

As a system resource available to COBOL, Prime's DBMS provides generalized database management capabilities for describing, creating, manipulating, and maintaining structured databases in a diverse range of applications.

It is particularly useful for:

- Interactive business data processing applications with complex relationships among data.
- On-line transaction processing
- Standardization of data meaning and usage
- A high degree of protected, concurrent usage

- Minimized data redundancy
- Integrity, backup and automatic recovery

Complete information on using DBMS in the COBOL environment is in DBMS Administrator's Guide, DBMS SCHEMA Reference Guide, and DBMS COBOL Reference Guide.

Multiple Index Data Access System (MIDAS)

MIDAS is a management software system of utilities and subroutines for creating and maintaining keyed-index/direct-access files.

MIDAS provides the COBOL programmer with a transparent multi-level file structure. All housekeeping functions on the index and data sub-files are performed by MIDAS subroutines called from COBOL programs.

Prime programming files created by programs written in one language may be accessed and manipulated by programs written in other languages, insuring compatability.

MIDAS Access Manager is reentrant. All active programs on Prime models 350 and above share a single copy of the manager, minimizing redundancy.

- There can be up to 5 alternate record keys for a COBOL MIDAS file
- Duplicate keys let MIDAS retrieve multiple records for a single key value
- LOCK prevents concurrent usage conflicts
- KEYS can be constructed from concatenated information
- A single program can make segmented and random accesses to a single file

Basic MIDAS template construction information is presented in Appendix B. The complete documentation is The MIDAS Reference Guide.

Forms Management Systems (FORMS)

FORMS is a system for creation, maintenance, and use of screen forms for interactive file maintenance. These screen forms are an extremely useful tool for the applications programmer writing data entry programs, where data fields are to be displayed in one or more formats.

FORMS keeps application programs, the forms and devices they use separated until run time. Thus, changes can be effected in one area without necessarily affecting the other two.

FORMS is compatible with DBMS and MIDAS; it is available to up to 63 concurrent users. It facilitates making accurate data available at widely dispersed locations for inquiry and/or update by transactions which can represent all elements of a business.

Details are in The FORMS Programmer's Guide.

Language interfaces

Since all Prime high-level languages are alike at the object-code level, and since all use the same calling conventions, object modules produced by the COBOL compiler can call and be called by modules produced by the F77, FTN, or PL1G compilers, provided that certain restrictions are observed:

- All I/O routines must be written in the same language.
- There must be no conflict of data types for variables being passed as arguments.
- Modules in 64V or 32I may call each other if they are otherwise compatible.

COBOL programs can also call PMA (Prime Macro Assembler) routines, and vice versa. For information, see Section 9 of this manual and **The Assembly Language Programmer's Guide.**

II

LANGUAGE-SPECIFIC SYSTEM INFORMATION

Compiling the program

INTRODUCTION

There is one COBOL compiler for all Prime computers and PRIMOS levels.

Source programs must meet the requirements of Prime's COBOL as specified in this manual.

The COBOL compiler generates object code in the segmented-addressing (64V) mode suitable for processing by Prime's segmented-addressing loader (SEG) utility on Prime models 350 and up.

USING THE COMPILER

The COBOL compiler is invoked by the COBOL command to PRIMOS:

COBOL pathname [-parameter-1 -parameter-2 ... -parameter-n]

where **pathname** is the pathname of the COBOL source program file

parameter-1 etc. are the mnemonics for the options controlling compiler functions such as I/O device specification, listings, and others.

For example:

```
COBOL MYPROG -L PRGLST
```

The mnemonics are explained in COMPILER FUNCTIONS in this section. All mnemonic parameters must be preceded by a hyphen (-). The name of the source program file must be specified as the above expression following the command COBOL.

Compilation messages

The Prime COBOL compiler flags milestones during compilation: Phases I through VI. Phase markers are output to the user's terminal in the following manner:

```
OK, COBOL pathname
Phase I      Environment Division
Phase II     Data Division
Phase III    Procedure Division
Phase IV     Intermediate code generation
Phase V      File Control Block generation
Phase VI     Final code generation
No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.1 <program>
```

If errors occur during compilation, an appropriate message will be output to the console, the listing file, or both. For example:

On the terminal:

```
OK, COBOL SAMPLE.SORT
 Phase I
 Phase II
 Phase III
 Phase IV
```

```
  Phase V
  Phase VI

1 Error, No Warnings, Prime V-Mode COBOL, Rev 17.1 <SORTIT>
```

In the listing file:

```
     .
     .
     .
ØØØ9  AREA-A VIOLATION; RESUMES AT  NEXT PARAGRAPH/SECTION/DIVISION/VERB.
     .
     .
     .
1 Error, No Warnings, Prime V-Mode COBOL, Rev 17.1 <SORTIT>
```

Note

If there are compiler errors, the object file is unusable.

End of compilation messages: After the compiler has completed a pass at the specified input file, generated code and listing output as specified by the mnemonic parameters, it prints a message at the user's terminal (see examples above). The message format is:

where

```
xxxx Errors, yyyy Warnings, Prime V-Mode COBOL, Rev 17.1 <program>
```

xxxx	is the number of errors encountered during compilation.
yyyy	is the number of warnings.
program	is the name of the program (ID) compiled.

An error is a mistake in a statement which makes execution of the program impossible.

A warning occurs when a statement is encountered which, although legal, may cause unexpected and/or undesirable results.

Note that the compiler does not support a SYSTEM READ/WRITE LOCK OF 5. Consequently, this will cause miscellaneous compiler aborts.

After compilation, control returns to PRIMOS.

Compiler error messages: The general format of the error message is:

```
n message.[]
```

where

n	Is the line reference number.
message	Is the standard COBOL compiler error message. A complete list is given in the Error Reference Section, Appendix E.
[ ]	Is a variable describing the problem.

For example:

```
     .
     .
     .
ØØ82  UNRESOLVED PROCEDURE-NAME; STATEMENT DELETED.  [SORT-DATA    ]

?Unsuccessful Compilation; Terminal Error On Line 82.
```

```
***** Compilation Terminated (Internal Error 1Ø6). Object File Unusable. *****
      .
      .
      .
```

Note

The example above is a semantic compiler error message accompanied by an internal error message.

If an internal error occurs on a line containing a semantic error, the semantic error, not the internal error, is probably the sole source of the problem. Correct the semantic error and recompile.

If an internal error occurs by itself, correct all previous semantic errors and recompile. If the internal error persists, report the internal error code number to your local field analyst.

An in-line error message takes the format:

****SYNTAXX ERROR** variable - in-line-message**

For example:

```
** SYNTAX ERROR **    PIC = X
```

Compiler Warning Messages: The general format of the message is:

n /W/ message. [ ]

where

n	is the line reference number.
/W/	indicates WARNING.
message	is the standard COBOL compiler warning message. A complete list is given in the Error Reference Section, Appendix E.
[ ]	is a variable describing the problem.

For example:

```
ØØ23  /W/ LITERAL TRUNCATED TO ITEM SIZE.  [FILE-RECORD-IN  ]
```

Program statistics

When programs or modules are compiled, program statistics are appended to the listing. These statistics relate to fixed storage allocations for specified aspects of the program as compiled.

These can be useful in determining the number of segments a program may require, or in setting up shared procedures.

It should be noted that while the storage allocation descriptions which follow are fixed for a given compilation, they do not include storage allocations for required libraries, subroutines, etc.

All sizes are stated in words.

Executable Code Size:	The number of words of code generated from the Procedure Division of the source program.
Constant Pool Size:	The size of any non-changing information required at run-time (e.g., quoted literals or decimal and binary constants).

Total Pure Procedure Size:	The sum of the two values above. This is the size of the sharable portion of a program.
Working-Storage Size:	The size of the user-defined Working-Storage.
Total Linkframe Size:	The total size of static storage needed by the program (i.e., file buffers, File Control Block's, etc.).
Stack Size:	The total size of stack needed. It is comprised of: • Standard stack header. • Arguments to this routine *(if any)*. • Compiler-generated temporaries.
Trace Mode:	The trace mode status given by on or off.

The number of arguments expected is given by:

xxx Arguments Expected.

where **xxx** is the number of arguments expected. If xxx=0, then the message is: No Arguments Expected.

The source program length is given by:

yyy Source lines.

where **yyy** is the number of lines in the source program.

COMPILER FUNCTIONS

The compiler functions enabled by the mnemonic parameters fall into three groups:

- **Specify I/O Devices**
  BINARY
  INPUT
  LISTING

- **Addressing Mode**
  64V

- **Enable Expanded Listings/Cross References**
  EXPLIST
  NOEXPLIST
  XREF
  NOXREF

The defaults listed in this section are those supplied by PRIME and are preceded by the "•" symbol. The system manager may change these at any particular installation. The programmer should check with the system manager to determine if defaults have been changed and, if so, which parameters are the new defaults.

Rust colored letters indicates minimum permissible abbreviations.

Specify input/output devices

The parameters below allow the user to inform the compiler of the input source filename and to specify the listing and binary object files.

-INPUT	Specifies input file/device (example -I TEST).
-I pathname	Specifies the name of the input source program.

(See Table 2-1.) This parameter must not be used if the source filename immediately follows the COBOL command; otherwise, it must be included in the parameter list.

-BINARY To override default, specifies binary (object) output file/device.

-B pathname The binary file will be created with the pathname specified (example: -B OUTPUT > TEST, where the binary file is created on the UFD OUTPUT under the filename TEST).

-B NO No binary file will be created; only a syntax check will occur.

• **-B YES** The binary file is created with the default name B–filename, where filename is the name of the source program file in the UFD in which the source program file resides. The binary file, however, is created in the UFD to which the user is attached when invoking the compiler.
If the BINARY parameter is not included in the command line, it is equivalent to -B YES. (See Table 2-1.)

-LISTING To override default, specifies listing file/device.

-L pathname The listing file will be created with the pathname specified (example: -L ELM > LTEST).

-L NO No listing file will be created. At later stages in program development or when minor modifications are made to programs, it may not be considered necessary to get a source program listing.

• **-L YES** The listing file is created with the default name L–filename, where filename is the name of the source program file in the UFD in which the source program file resides. The listing file, however, is created in the UFD to which the user is attached when invoking the compiler.

-L TTY The listing file is printed on the user's terminal.

-L SPOOL The listing file is spooled directly to the line printer.
If the LISTING parameter is not included in the command line, it is equivalent to -L YES.

-SOURCE Same as -INPUT. See -INPUT.

Addressing mode

• **-64V** Generates segmented-addressed code which must be loaded with the SEG loader. It provides a user area up to 32 megabytes (256 segments of 128K bytes each). It may be run on any Prime model 350 or above under PRIMOS.

Enable expanded listings/cross references

Expanded listing: The expanded listing is a combination of a regular listing (source code with line number appended) and machine-generated code.

Table 2-1. Compiler File Specifications

COMPILER MNEMONICS	INPUT	LISTING	BINARY
pathname	Looks for file named pathname as source file	Opens file named pathname as listing file	Opens file named pathname as binary (object) file.
YES		Uses default file-name for listing file L–PROGRM.	Uses default file-name for binary file B–PROGRM.
NO		No listing file.	No binary file.
TTY		Print listing on user terminal.	
SPOOL		Spool listing directly to line printer.	
Option not invoked	Source filename must be first option after COBOL command.	Same as YES	Same as YES

- **-NOEXPLIST** Suppresses generation of the expanded listing. This is the normal default.

  -EXPLIST Generates an expanded listing at the end of the listing file. User defined names are NOT used, machine-generated labels are placed in the listing.

An expanded listing example for SAMPLE appears in Appendix F. To fully utilize the listing, a knowledge of PMA is necessary. The reader is referred to The Assembly Language Programmer's Guide.

Cross-reference listing: The Cross Reference has two compile-time options, -NOXREF or -XREF.

- **-NOXREF** Suppresses generation of any cross-reference listing. This is the normal default.

  -XREF Generates a cross-reference listing at the end of the listing file. A line number with a suffix 'D' indicates a paragraph or section name in the Procedure Division.

For example:

```
OK, COBOL SAMPLE.SORT -XREF -L TTY
                .
                .
                .
 (0073)           PROCEDURE DIVISION.
 (0074)           START-PARA.
                .
                .
                .
 (0115)           GET-TOTALS SECTION.
 (0116)           GET-TOTAL.
```

```
(Ø117)              DISPLAY 'ENTER MONTH XX (Ø1-12) OR ENTER 99 TO QUIT'.
(Ø118)              ACCEPT MONTH-ACCEPT.
(Ø119)              IF MONTH-ACCEPT = 99
(Ø12Ø)                  GO TO DONE-PARA.
(Ø121)              IF NOT VALID-MONTH
(Ø122)                  GO TO GET-TOTAL.
                .
                .
                .
(Ø148)          DONE-PARA.
(Ø149)              STOP RUN.
 ADD-TOTALS                    Ø129  Ø139D
                .
                .
                .
 DONE-PARA                     Ø12Ø  Ø148D
                .
                .
                .
 GET-TOTAL                     Ø116D Ø122  Ø135
 GET-TOTALS                    Ø093  Ø115D
                .
                .
                .
 START-PARA                    Ø074D
 SUM-DEPT                      Ø041  Ø13Ø  Ø132  Ø14Ø
 TABLE-AREA                    Ø062
 TABLE-VALUE                   Ø044  Ø057
 VALID-MONTH                   Ø043  Ø121


P R O G R A M    S T A T I S T I C S

                .
                .
                .
```

An actual listing for SAMPLE.SORT is shown in Section 11.

COMPILER GENERATED FILES

File types

Three types of files may be involved during compilation. They are: source file, listing file, object file. Of these, the listing and object files are compiler-generated. Corresponding PRIMOS file units are given below.

File Type	PRIMOS file unit
Source	1
Listing	2
Object	3

File names

If disk is specified as the device for the listing and/or object file, the COBOL compiler causes these files to be opened under the filename specified in the compile command. The default convention for a listing file is L–filename. The default convention for an object file is B–filename. Thus, for a source file named SAM, following the compile command COBOL SAM, the listing and object files would exist in the current UFD as L–SAM and B–SAM, respectively.

If the source file is given as a pathname, e.g., [MFD] > UFD1 ... > SAM, where the file SAM does not reside in the current UFD (that in which compilation is occuring), the listing and object files will still be opened as L–SAM and B–SAM, respectively. Although the source exists in another UFD, L–SAM and B–SAM will, nevertheless, be opened in the current UFD.

If the user desires the listing or object files to have other than default names as outlined above, the PRIMOS command, LISTING, must be invoked prior to compilation.

File manipulation

LISTING filename-2 opens a listing file in the current UFD, on PRIMOS file unit 2, under the specified name filename-2. This inhibits the compiler instruction COBOL from opening a default listing file.

The listing output(s) of more than one source file can be concatenated if all listings are generated prior to closing the listing file.

For example,

```
LISTING filename-2
    .
    .
    .
COBOL source-1 mnemonics
    .
    .
    .
COBOL source-n mnemonics
    .
    .
    .
CLOSE ALL
```

Note

System responses are not printed in the example above. Filename-2 will contain the concatenation of all listing outputs from source-1, ..., source-n (for those compilations wherein listings were specified).

BINARY filename-3 opens a binary (object) file with the specified name (in the current UFD) on PRIMOS file unit 3. This inhibits the compiler instruction COBOL from opening a default object file.

If the BINARY or LISTING commands are used prior to the COBOL command to establish non-default files, then COBOL does not close these files upon completion.

After COBOL returns command to PRIMOS, these files should be closed by the user by typing:

CLOSE { **2** / **filename-2** } { **3** / **filename-3** }

or

CLOSE ALL

3

Loading and executing programs

LOADING AND EXECUTING PROGRAMS

The PRIMOS SEG utility loads and executes all COBOL programs. This section describes normal loading and execution, and specifies some techniques required for COBOL programs. The loading concept is described in more detail in the Prime User's Guide. For extended loading features, as well as a complete description of all SEG commands, including those for advanced system-level programming, refer to the LOAD and SEG Reference Guide.

LOADING PROGRAMS

Normal Loading

Most loads can be accomplished by the following basic procedure:

1. Invoke the SEG loader with the SEG command. (A '#' sign will be the prompt symbol.)
2. Enter the SEG-level LOAD command to start the load subprocessor and to set up the runfile (LO #filename). (A '$' sign will appear as the next prompt symbol.)
3. Use the load subprocessor's LOAD command to load the object files in the following order:
   - The object file of the main program (LO B_filename)
   - The object files of any separately compiled subroutines (preferably in order to frequency of use)
4. Use the load subprocessor's LIBRARY command to load subroutines called from libraries in the following order:
   - Shared COBOL library (LI VCOBLB)
   - Non-shared COBOL library, if user written subroutines are loaded (LI NVCOBLB)
   - Other Prime libraries, if required (LI filename)
   - Standard FORTRAN library (LI)

At this point, you should receive a LOAD COMPLETE message. If the message is absent, do a MAP 3 to identify the unsatisfied references and load them. In the unlikely event some other SEG error message appears, refer to the LOAD and SEG Reference Guide for the probable cause and correction.

5. SAVE the runfile.
6. The QUIT command exits from the utility.

As an example of loading, assume that the user has compiled a main program, MAIN, and a subroutine in a separate source file, SUBR. Both have been compiled using the default object filenames. They could be loaded as follows:

```
OK, SEG              brings SEG into memory
[SEG rev 17.1]
# LO #MAIN           invokes the loader and establish a runfile
$ LO B MAIN          loads the main program
$ LO B SUBR          loads any separately compiled subroutine
$ LI VCOBLB          loads the shared COBOL library
$ LI NVCOBLB         loads the non-shared COBOL library
$ LI                 loads the FORTRAN library
LOAD COMPLETE        Loader indicates all references are satisfied
$ SA                 user saves runfile
$ Q                  returns to PRIMOS level

OK,
```

EXECUTING LOADED PROGRAMS

Execution of Runfiles

For programs loaded and saved by SEG, execution is performed at the PRIMOS level using the SEG command:

SEG #filename

where **#filename** is the filename (or pathname) of a SEG runfile. SEG loads the runfile into segmented memory and begins execution of the program after a dialog with C$IN (see below).

A shortcut to saving and executing a loaded program is available. Immediaely after receiving the LOAD COMPLETE message, enter the load subprocessor's EXECUTE command. This command will then save the loaded program and start executing the program after a dialog with the C$IN utility program (see below). EXECUTE will also work if a SAVE has been given explicitly.

Upon completion of program execution, control returns to PRIMOS command level.

C$IN utility program

Immediately following the execute command of SEG or EXECUTE, a series of questions will be asked concerning run-time file assignments. These questions are prompted by the utility program C$IN.

The utility programs will ask on the terminal:

```
ENTER FILE ASSIGNMENTS:
>
```

The proper response to the request above is to give the name of the file (as stated in the VALUE OF FILE-ID clause of the File Description), followed by the pathname of the actual fiile to be associated with the ID. The pathname can be a filename if the file resides in the current UFD. For example, suppose that in a COBOL program the following statements existed:

```
FD  TEST-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF FILE-ID IS 'FILE1'.
```

then the proper dialog with C$IN would be:

```
ENTER FILE ASSIGNMENTS:
>FILE1=REED>T1

          OR

>FILE1=$MT1, S, T1, 000001
```

The first statement would go to a UFD called REED and use a disk file called T1 as input to TEST-FILE in the program.

The second statement requires MAG TAPE unit 1 to be assigned, with the tape mounted to contain a TAPE-ID of T1 and a volume serial of 000001.

The utility program C$IN will do all prescreening of the files and display the prompt character while waiting for user input. There should be one entry for each FD if its FILE-ID is to be reassigned. When no files remain to be entered, the single slash character /L will conclude the session. Execution of the application program will then begin, using the file assignments which were just entered.

If there are no files in the program or if the main program contains an EXIT PROGRAM statement, C$IN will not ask for file assignment. In the latter case, C$IN will take the default VALUE OF FILE-ID values as defined in the FDs. If there are no VALUE OF FILE-ID clauses in the program, the compiler will generate files with the name F1 F2, F3, etc.

Note that when using standard magnetic tape labels the tape will automatically rewind after a CLOSE statement. With non-standard labels the tape will stay positioned to the end of the file.

Disk formats (filenames and pathnames): A pathname in a disk format entry is an extended form of the filename which describes the location of the file in the directory structure.

Pathnames specified as parameters to external commands should not contain spaces. The space or comma is used to separate one parameter from another. If a space must be specified due to a password, enclose the entire pathname in single quotes.

For example:

```
UPCASE    UFD1>FILE UFD2>FILE

UPCASE    'UFD1 PASSWORD>FILE' UFD2>FILE
```

```
          FILE-ID=MAGTAPE, LABEL, TAPE-ID, TAPE-NUMBER
```

MAGTAPE: $MT(X) being a 9-track drive number
LABEL: N: for no label information
S: specifies the tape contains standard labels and is pre-numbered.

TAPE-ID: is up to a 17-character field which is written in the label of the tape being created; or is used for comparison if the ape is being read. Label must have been specified as S.

TAPE NUMBER: is a 16-character field which is checked at open-time when reading a tape, but is not needed when creating a tape.

Note

Appendix G explains how to create and read a VOL1 label on a magtape.

C$IN error messages: The following are error messages which may be output by the C$IN utility program:

```
FILENAME TOO LONG (no equal sign found)
INVALID TREE SYNTAX (see allowable format)
NO FILENAME ENTERED (equal sign with no filename)
INVALID TAPE UNIT (format did not contain MTx)
NO TAPE NAME ENTERED (standard label specified)
TAPE NAME GREATER THAN 17
TAPE NUMBER GREATER THAN 6
```

Run-time error messages

Alphabetic list of the run-time error messages are provided in Appendix E of this document.

III

COBOL LANGUAGE REFERENCE

4

Fundamental concepts of COBOL

DIVISIONS OF A COBOL PROGRAM: A SUMMARY

Every COBOL program consists of four divisions:

- Identification Division
- Environment Division
- Data Division
- Procedure Division

Identification Division

The Identification Division (ID Division) assigns a name to the program and allows the programmer to enter other documentary information, such as the programmer's name, the date the program was written, and so on.

Environment Division

The Environment Division specfies a standard method of expressing those aspects of a data-processing problem which depend upon the physical characteristics of a specific computer.

Two sections make up the Environment Division; the Configuration Section and the Input-Output Section.

Configuration section: describes the computer configuration on which the source program is compiled, and the configuration on which the compiled program is to be run. It also relates system names used by the compiler to names introduced by the programmer in the source program.

Input-output section: contains the information needed to control transmission and handling of data between external media and the program. This section describes the name, type of organization, and access mode of each data file, and associates the file with a peripheral device.

Data Division

The Data Division provides the compiler with a detailed description of the characteristics of every data item used within the program. There are three sections of the Data Division; the File Section, the Working-Storage Section and the Linkage Section.

File section: describes the structure of data files. Each file is defined by a File Description entry and one or more Record Description entries.

Working-storage section: describes reecords and noncontiguous data items which are not part of external files, but are developed and processed internally. It also defines data items whose values do not change during the execution of the program (i.e., constants).

Linkage section: of a COBOL program is meaningful only in a called program. This section, appearing in the called program, describes data items which may be referred to by both the called and calling programs.

Procedure Division

The Procedure Division contains instructions (COBOL statements) required to solve a data processing problem.

This division contains two types of sections: declarative sections and procedural sections.

Declarative sections: are optional. When used, they must be grouped at the beginning of the Procedure Division. Declarative sections permit the execution of instructions which are not performed in the regular sequence of coding. Such out-of-sequence procedures are usually initiated by a condition which the program does not test directly.

Procedural sections: follow declaratives in a logical sequence. Each procedural section comprises one or more paragraphs. Each paragraph consists of one or more COBOL sentences. Sentences, in turn, are comprised of one or more COBOL statements.

Execution of the instructions in the Procedure Division begins with the first statement in the division, excluding declaratives. Statements are executed in the order in which they are presented for compilation, unless the rules indicate otherwise.

The Procedure Division ends at that point in the source program after which no further procedures appear. This coincides with the physical end in the program.

The following skeletal coding defines the program format and order:

```
 ID DIVISION.
 PROGRAM-ID. program-name.
[AUTHOR. [comment-entry] ... ]
[INSTALLATION. [comment-entry] ... ]
[DATE-WRITTEN. [comment-entry] ... ]
[DATE-COMPILED. [comment-entry] ... ]
[SECURITY. [comment-entry] ... ]
[REMARKS. [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] ... ] ...
[sort-file-description-entry.
{record-description-entry} ... ] ... ]
[WORKING-STORAGE SECTION.
[77-level-description-entry] ...
[record-description-entry] ... ]
[LINKAGE SECTION.
[77-level-description-entry] ...
[record-description-entry] ... ]
 PROCEDURE DIVISION [USING identifier-1 ... ].
[DECLARATIVES.
{section-name SECTION. use-sentence.
[paragraph-name. [sentence] ... ] ... } ...
 END DECLARATIVES.
{section-name SECTION.
[paragraph-name. [sentence] ... ] ... } ...
```

The following listing file for sample program SAMPLE, illustrates the program format and order. SAMPLE creates and reads a relative file sequentially.

```
Rev 17.2  COBOL     Source File:  SAMPLE                    11/05/79  13:55
 (0001)          ID DIVISION.
 (0002)          PROGRAM-ID. SAMPLE.
 (0003)          INSTALLATION. PRIME COMPUTER TECHNICAL PUBLICATIONS DIVISION.
 (0004)          DATE-WRITTEN. OCT 26, 1979.
 (0005)          SECURITY. NONE.
 (0006)          REMARKS. THIS PROGRAM CREATES AND READS A RELATIVE FILE
 (0007)                   SEQUENTIALLY.
 (0008)          ENVIRONMENT DIVISION.
 (0009)          CONFIGURATION SECTION.
 (0010)          SOURCE-COMPUTER. PRIME-750.
 (0011)          OBJECT-COMPUTER. PRIME-750.
 (0012)          INPUT-OUTPUT SECTION.
 (0013)          FILE-CONTROL.
 (0014)              SELECT PRINT-FILE ASSIGN TO PRINTER.
 (0015)              SELECT CARD-FILE ASSIGN TO PFMS.
 (0016)              SELECT DIRECTORY-FILE ASSIGN TO PFMS,
 (0017)                     ORGANIZATION IS RELATIVE,
 (0018)                     RELATIVE KEY IS RELATIVE-KEY,
 (0019)                     ACCESS MODE IS SEQUENTIAL,
 (0020)                     FILE STATUS IS FILE-STATUS.
 (0021)          DATA DIVISION.
 (0022)          FILE SECTION.
 (0023)          FD  PRINT-FILE, LABEL RECORDS ARE OMITTED,
 (0024)                          DATA RECORD IS PRINT-LINE,
 (0025)                          RECORD CONTAINS 132 CHARACTERS.
 (0026)          01  PRINT-LINE PIC X(132).
 (0027)          FD  CARD-FILE, LABEL RECORDS ARE STANDARD,
 (0028)              VALUE OF FILE-ID IS 'INDATA'.
 (0029)          01  CARD-IMAGE PIC X(80).
 (0030)          FD  DIRECTORY-FILE, LABEL RECORDS ARE STANDARD,
 (0031)              VALUE OF FILE-ID IS 'D-FILE'.
 (0032)          01  DIRECTORY-RECORD.
 (0033)              05  CARRIAGE-CONTROL PIC X.
 (0034)              05  NAME.
 (0035)                  10  LAST-NAME PIC X(15).
 (0036)                  10  FIRST-NAME PIC X(15).
 (0037)              05  FILLER PIC X(1).
 (0038)              05  ADDRESS PIC X(25).
 (0039)              05  FILLER PIC X(1).
 (0040)              05  CITY PIC X(4).
 (0041)              05  FILLER PIC X(3).
 (0042)              05  PHONE-NO PIC 9(7).
 (0043)              05  FILLER PIC X(8).
 (0044)          WORKING-STORAGE SECTION.
 (0045)          01  RELATIVE-KEY PIC XX.
 (0046)          77  FILE-STATUS PIC XX VALUE SPACES.
 (0047)          01  HEADER.
 (0048)              05  H1 PIC X(5), VALUE IS ' NAME'.
 (0049)              05  FILLER PIC X(27), VALUE IS SPACE.
 (0050)              05  H2 PIC X(6), VALUE IS 'STREET'.
```

```
(0051)              05  FILLER PIC X(19), VALUE IS SPACE.
(0052)              05  H3 PIC X(4), VALUE IS 'CITY'.
(0053)              05  FILLER PIC X(4), VALUE IS SPACE.
(0054)              05  H4 PIC X(5), VALUE IS 'PHONE'.
(0055)          PROCEDURE DIVISION.
(0056)          BEGIN SECTION.
(0057)          CREATE-FILE.
(0058)              OPEN INPUT CARD-FILE.
(0059)              OPEN OUTPUT PRINT-FILE, DIRECTORY-FILE.
(0060)              WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
(0061)          READ-NEXT.
(0062)              READ CARD-FILE AT END GO TO LIST-DIRECTORY.
(0063)              MOVE CARD-IMAGE TO PRINT-LINE.
(0064)              MOVE CARD-IMAGE TO DIRECTORY-RECORD.
(0065)              WRITE PRINT-LINE.
(0066)              WRITE DIRECTORY-RECORD INVALID KEY DISPLAY 'INVALID KEY'.
(0067)              GO TO READ-NEXT.
(0068)          LIST-DIRECTORY.
(0069)              CLOSE CARD-FILE, DIRECTORY-FILE.
(0070)              DISPLAY 'END TEST TO CREATE FILE'.
(0071)              OPEN INPUT DIRECTORY-FILE.
(0072)          LAST-SECTION SECTION.
(0073)          LIST.
(0074)              WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
(0075)          READ-NEXT-DIRECTORY-RECORD.
(0076)              READ DIRECTORY-FILE NEXT RECORD AT END GO TO CLOSE-ALL.
(0077)              MOVE DIRECTORY-RECORD TO PRINT-LINE.
(0078)              WRITE PRINT-LINE.
(0079)              GO TO READ-NEXT-DIRECTORY-RECORD.
(0080)          CLOSE-ALL.
(0081)              CLOSE DIRECTORY-FILE, PRINT-FILE.
(0082)              DISPLAY 'END TEST SEQUENTIAL READ AFTER A START'.
(0083)              STOP RUN.
                        .
                        .
                        .
```

```
No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.2 <SAMPLE>
```

LANGUAGE CONSIDERATIONS

Format notation

Throughout the Reference portion of this document, basic formats are prescribed for various clauses or statements. These generalized descriptions guide the programmer in writing his (or her) own statements. They are presented in a uniform system of notation:

- All words printed entirely in capital letters are Reserved Words. These are words which have preassigned meanings. In all formats, words in capital letters represent an actual occurrence of those words.
- All underlined Reserved Words are required unless the portion of the format containing them is itself optional. Such underlined Reserved Words are Key Words. If any Key Word is missing or is incorrectly spelled, it is considered an error in the program. Reserved Words not underlined may be included or omitted at the option of the programmer. These words are

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

- The characters < , > , and = when appearing in formats, although not underlined, are required when such formats are used.
- All punctuation and other special characters represent the actual occurrence of those characters. Punctuation is essential where it is shown. Additional punctuation can be inserted, according to the rules for punctuation specified in this publication. In general, terminal periods are shown in formats in the manual because they are required; semicolons and commas are not shown generally because they are optional.
- Words printed in lower-case letters in formats represent programmer defined variables.
- Parts of a statement or Data Description entry which are enclosed in brackets [ ] are optional. Parts between matching braces { } represent a choice of mutually exclusive options, of which one must be chosen. When brackets or braces enclose a portion of a format, but only one possibility is shown, the function of the brackets or braces is to delimit that portion of the format to which a following ellipsis applies.
- Certain entries in the formats consist of a capitalized word(s) followed by the word Clause or Statement . These designate clauses or statements which are described in other formats in appropriate sections of the text.
- In order to facilitate reference to them in the text, some lower case words are followed by a hyphen and a digit or letter. This modification does not change the syntactical definition of the word.
- The ellipsis (...) indicates that the immediately preceding unit may occur once, or any number of times in succession. A unit means either a single lower-case word, or a group of lower-case words and one or more Reserved Words enclosed in brackets or braces. If a term is enclosed in brackets or braces, the entire unit of which it is part must be repeated when repetition is specified.
- Comments, restrictions, and clarifications on the use and meaning of every format are contained in the appropriate portions of the manual.
- Multiple formats for a given COBOL verb are mutually exclusive options, of which only one may be chosen.

Punctuation rules

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

- A period, semicolon, or comma, when used, cannot be preceded by a space, but must be followed by space.
- Left and right parentheses must appear in balanced pairs. They are used to delimit subscripts, indexes, arithemetic expressions, or conditions.
- At least one space must appear between two successive words and/or literals. Two or more successive spaces are treated as a single space, except in non-numeric literals.
- Relation characters should always be preceded by a space and followed by another space.
- When the period, comma, plus, or minus characters are used in the PICTURE clause, they are governed solely by rules for numeric edited items.
- A comma may be used as a separator between successive operands of a statement, or between two subscripts.

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

Coding rules

Since Prime COBOL is a subset of American National Standards Institute (ANSI) COBOL, programs are written on standard COBOL coding sheets (Figure 4-1). The following rules are applicable:

- Each line of code should have a six-digit sequence number in positions 1-6, such that the source statements are in ascending order. Blanks are also permitted in positions 1-6.
- Reserved Words for division, section, and paragraph headers must begin in the A Area (positions 8-11). Procedure-names must also appear in the A Area (at the point where they are defined). Level numbers may appear in the A Area.
- All other program elements must be confined to positions 12-72, governed by the other rules of statement punctuation.
- Positions 73-80 are ignored by the compiler. Frequently, these positions are used to contain the program identification.
- Position 7 is used for special coding symbols. Explanatory comments may be inserted on any line within a source program by placing an asterisk (*) in position 7 of the line. Any combination of characters may be included in the A and B Areas of that line. The asterisk and the characters will be produced on the source listing but serve no other purpose. If a slash (/) appears in position 7, the next line will be printed at the top of a new page when the compiler lists the program. A hyphen (-) is used to continue a non-numeric literal from one line to another. Refer to **Non-numeric literals** (later on in this section) for coding rules.

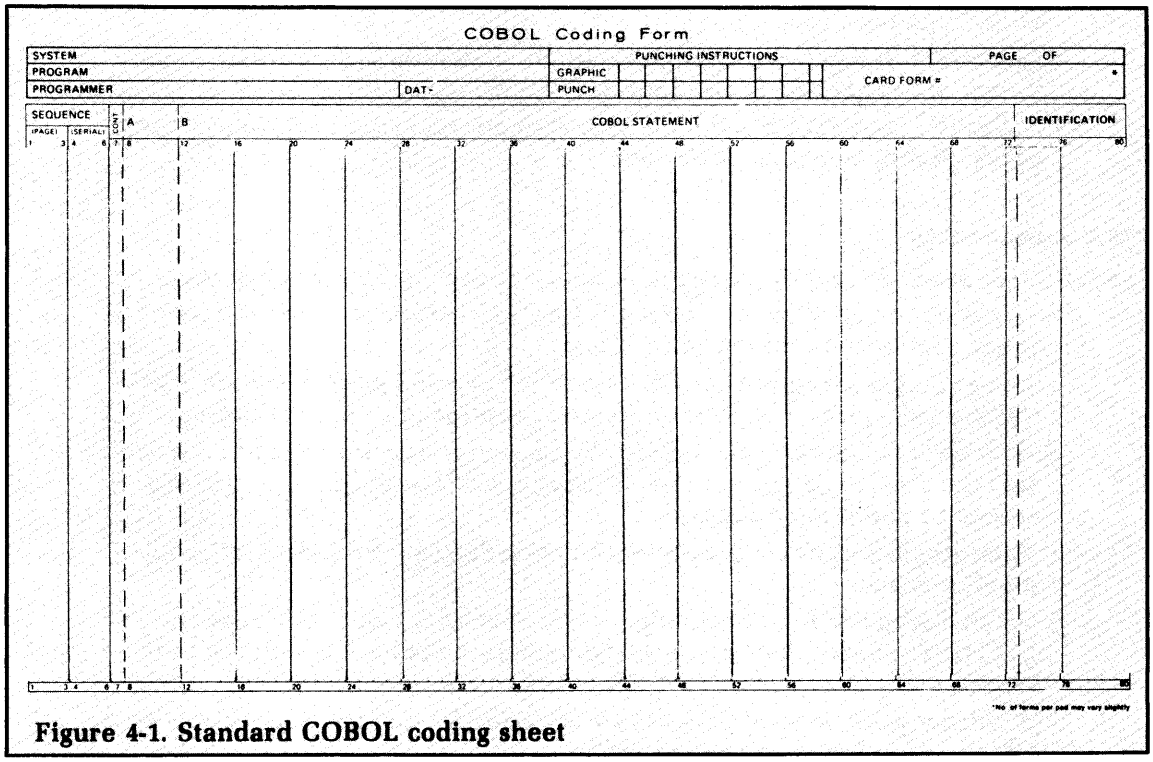


Figure 4-1. Standard COBOL coding sheet

Prime character set

The standard character set utilitized by Prime is the ANSI, ASCII, 7-bit character set. The entire set of characters, with octal, hexadecimal, and punched card equivalents, is presented in Appendix C.

Collating sequence

Each character in the Prime character set has a unique octal value which establishes the collating sequence for the character set. This sequence conforms to the American Standards Code for Information Interchange (ASCII). The characters in Appendix C, the ASCII Character Set, are arranged in ascending order from top to bottom.

LANGUAGE SPECIFICATIONS

COBOL character set

The standard COBOL language character set utilizes 52 characters as follows: The numbers 0 through 9, the 26 uppercase letters of the English alphabet, the space (blank), and 14 special characters. (A fifteenth special character, the apostrophe, is used by Prime COBOL as an alternate for the quotation mark). The complete COBOL character set is illustrated in Figure 4-2.

The individual characters of the COBOL language are the basic units used to form the major elements of COBOL, i.e., character-string, separators, words, statements, sentences, paragraphs, and sections.

Character strings

A character-string is a character or a seqence of contiguous characters which forms a COBOL word, a literal, a PICTURE character-string, or a comment-entry. A character-string is delimited by separators.

Picture character-strings

A PICTURE character-string (picture-string) consists of certain combinations of characters in the COBOL character set used as symbols. See DATA DIVISION, PICTURE, for a description of the PICTURE character-string and the rules governing its use. A punctuation character which is part of the specification of a PICTURE character-string is not considered as a punctuation character, but as a symbol in that PICTURE character-string.

Word formation

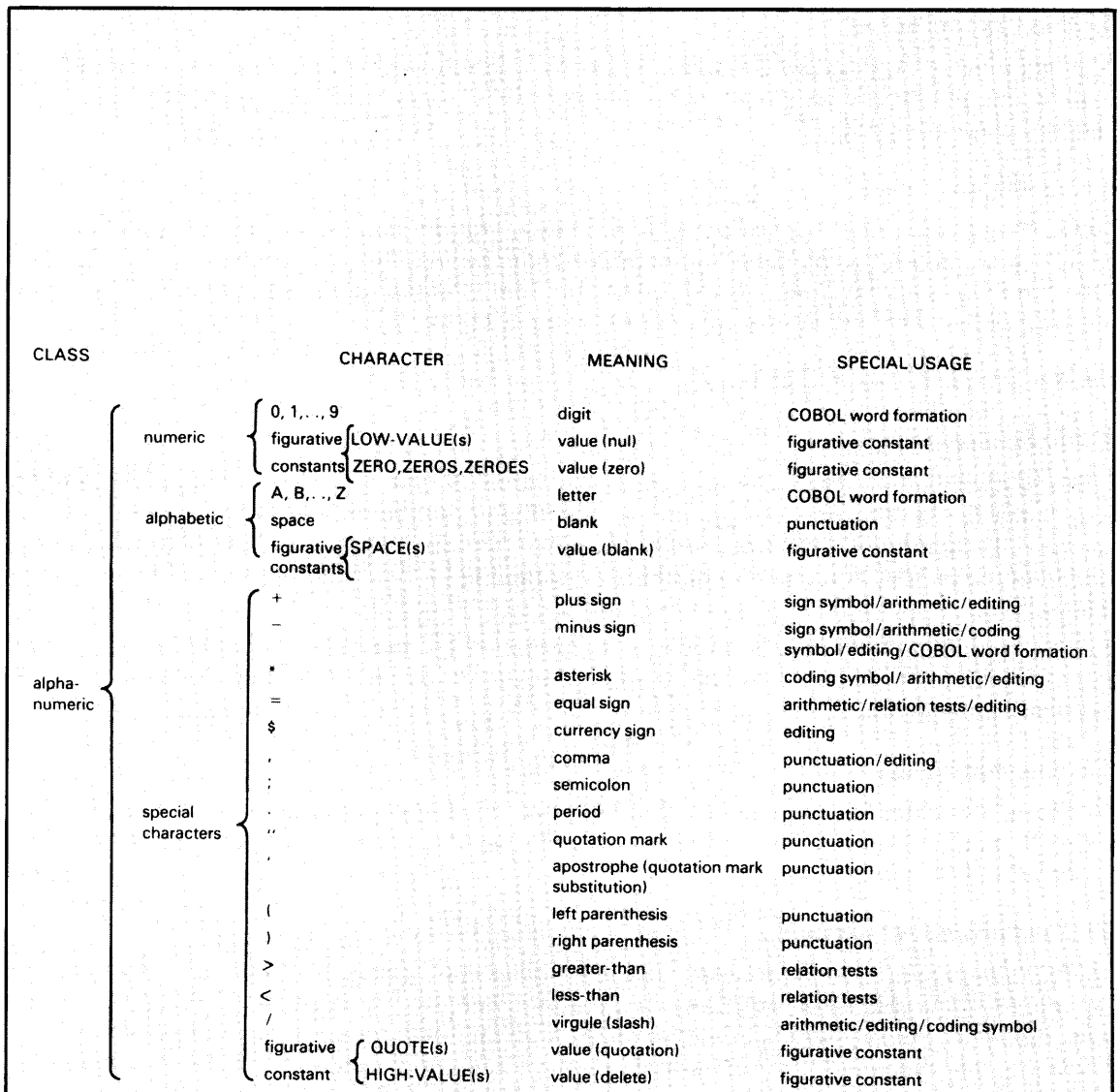
A COBOL word is a character-string of not more than 30 characters chosen from the following set of 37 characters:

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

A word must 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 or programmer-defined words.

If a programmer-defined word is not unique, there must be an unique method of referencing it by using name qualifiers, e.g., TAX-RATE IN STATE-TABLE. Primarily, a programmer-defined word identifies a data item or field, and is called a data-name. Other cases of programmer-defined words are file-names, condition-names, and mnemonic-names.



CLASS		CHARACTER	MEANING	SPECIAL USAGE
alpha-numeric	numeric	0, 1, . . , 9	digit	COBOL word formation
		figurative constants: LOW-VALUE(s)	value (nul)	figurative constant
		figurative constants: ZERO,ZEROS,ZEROES	value (zero)	figurative constant
	alphabetic	A, B, . . , Z	letter	COBOL word formation
		space	blank	punctuation
		figurative constants: SPACE(s)	value (blank)	figurative constant
	special characters	+	plus sign	sign symbol/arithmetic/editing
		−	minus sign	sign symbol/arithmetic/coding symbol/editing/COBOL word formation
		*	asterisk	coding symbol/ arithmetic/editing
		=	equal sign	arithmetic/relation tests/editing
		$	currency sign	editing
		,	comma	punctuation/editing
		;	semicolon	punctuation
		.	period	punctuation
		"	quotation mark	punctuation
		'	apostrophe (quotation mark substitution)	punctuation
		(	left parenthesis	punctuation
		)	right parenthesis	punctuation
		>	greater-than	relation tests
		<	less-than	relation tests
		/	virgule (slash)	arithmetic/editing/coding symbol
	figurative constant	QUOTE(s)	value (quotation)	figurative constant
	figurative constant	HIGH-VALUE(s)	value (delete)	figurative constant

Note

When the figurative constant LOW-VALUES is used with binary data, it is interpreted as numeric. In all other instances, it is interpreted as alphanumeric.

Figure 4-2 COBOL Character Set

With the exception of paragraph-name and section-name, all programmer-defined words must contain at least one alphabetical character.

Reserved words

A Reserved Word is one of a specified list of words which may be used in COBOL source programs, but which may not appear as programmers-defined words. They may only be used as specified in the general formats. The types of Reserved Words are:

- Key words
- Optional words
- Connectives
- Figurative constants
- Special-character words

Key words: A key word is one whose presence is required when the statement in which the word appears is used in a source program. Within each statement, such words are uppercase and underlined.

Optional words: Within each format, uppercase words which are not underlined are called optional words; they may appear at the user's option. The presence or absence of an optional word does not alter the meaning of the COBOL program in which it appears, but is required as written when used.

Connectives: The three types of connectives are:

1. Qualifier-connectives used to associate a data-name, condition-name, text-name, or paragraph-name with its qualifier: OF, IN
2. Series connectives which may be used to link two or more consecutive operands: , (comma) or ; (semicolon)
3. Logical connectives used in the formation of conditions: AND, OR

Figurative constants: Figurative constants are Reserved Words used to name and reference specific constant values. A figurative constant represents as many instances of the associated character as required in the context of the statement.

The singular and plural forms are equivalent and may be used interchangeably.

A figurative constant may be used wherever **literal** appears in a format description; except that, whenever the literal is restricted to numeric characters, the only figurative constant permitted is ZERO (ZEROS, ZEROES). A figurative constant must not be bounded by quotation marks.

Values, and the Reserved Words used to reference them are:

ZERO **ZEROS** **ZEROES**	=	The ASCII character represented by Octal 260
LOW-VALUE **LOW-VALUES**	=	The character whose Octal representation is 200
HIGH-VALUE **HIGH-VALUES**	=	The character whose Octal representation is 377
QUOTE **QUOTES**	=	The quotation mark, whose Octal representation is 242
SPACE **SPACES**	=	The blank character represented by Octal 240
All literal	=	The literal is a single character, used in MOVE statements; the receiving field is filled with the given character

Special character words: The arithmetic operators and relation characters are Reserved Words. They comprise the following:

Operators	Meaning
Arithmetic	
+	Addition
–	Subtraction
*	Multiplication
/	Division
Relation	
=	is equal to
<	is less than
>	is greater than

Programmer-defined words

A programmer-defined word is one supplied by the user to satisfy the format of a clause or statement. Each is constructed according to the rules for Word Formation. The categories for programmer-defined words include:

- Level-numbers
- Data-names
- File-names
- Condtion-names
- Mnemonic-names
- Paragraph-names
- Section-names

Level numbers: For the purposes of processing, the contents of a file are divided into logical records. The level concept is inherent in the structure of a logical record, in that it allows the specification of record subdivisions for the purpose of data reference.

Once a subdivision is specified, it may be further subdivided to permit more detailed data referral. The most basic subdivision of a record, that which cannot be futher subdivided, is an elementary item. Data items which contain subdivisions are known as group items.

Level-numbers are one or two character, programmer-defined words. All level-numbers are numeric. They group items within the data hierarchy of the Record Description. Since records are the most inclusive data items, level-numbers for records begin at 01.

Less inclusive groups are assigned numerically higher level-numbers. Level-numbers of items within groups need not be consecutive. A group whose level is 02 includes all groups and elementary items described under it until a level number less than or equal to 02 is encountered.

Separate entries are written in the source program for each level. The range of levels is 01 through 49. 1 through 9 may be written as single numbers.

Level numbers 66, 77 and 88 are used in certain applications and are defined together with additional level-number information in Section 7, DATA DIVISION.

A weekly time card record illustrates the level concept. It is divided into four major items: name, employee-number, date, and hours, with more specific information appearing for name and date.

```
                                LAST-NAME
          NAME                  FIRST-INIT
                                MIDDLE-INIT

          EMPLOYEE-NUM

.         TIME-CARD
```

```
                              MONTH
DATE                          DAY
                              YEAR

HOURS-WORKED
```

The time card record might be described (in part) by Data Division entries having the following level-numbers, data-names, and picture definitions.

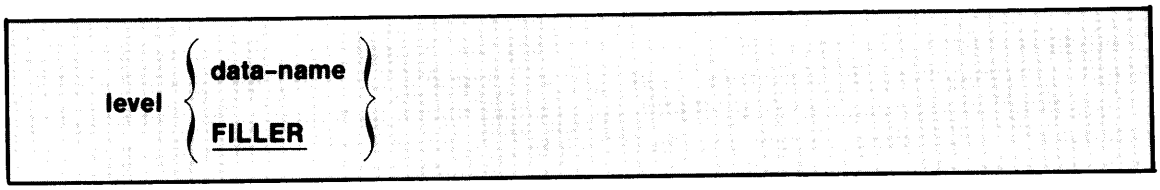
```
01  TIME-CARD.
    05  NAME.
        10   LAST-NAME          PICTURE X (18).
        10   FIRST-INIT         PICTURE X.
        10   MIDDLE-INIT        PICTURE X.
    05  EMPLOYEE-NUM  PICTURE 99999.
    05  DATE.
        10   MONTH     PIC 99.
        10   DAY       PIC 99.
        10   YEAR      PIC 99.
    05  HOURS-WORKED  PICTURE 99V9.
```

Data names: In the preceding time card example, TIME-CARD, NAME, LAST-NAME FIRST-INIT, etc., are data-names supplied by the programmer.

A data-name is a word assigned by the user to identify a data item used in a program. A data-name always refers to a field of data, not a particular value.

A data-name is formulated according to the rules for Word Formation; it must begin with an alphabetic character.

A data-name or the Key Word FILLER must be the first word following the level-number in each Record Description entry, as shown in the following general format:



This data-name is the defining name of the entry. It is the means by which references to the associated data area (containing the value of a data item) are made.

If some of the characters in a record are not used in the processing steps of a program, then the data description for these characters need not include a data-name. In this case, FILLER is written in lieu of a data-name after the level number. Note that FILLER can be used only at the elementary level; ANSI standards do not permit its use at a group level.

File-names: A file is a collection of data records containing individual records of a similar class or application. A file-name is defined by an FD entry in the Data Division's File Section. FD is a Reserved Word which must be followed by an unique programmer-supplied word called the file-name. Rules for composition of the file-name word are identical to those for data-names (see Word Formation). References to a file-name appear in Procedure statements OPEN, CLOSE and READ, as well as in the Environment Division.

Condition-names: A condition-name is a name assigned to a specific value, set of values, or range of values, within a complete set of values which a data item may assume.

A condition-name is defined within the Data Division in level 88 entries. Rules for the formation of condition-name words are the same as those specified in Word Formation. Additional information concerning condition-names, and those procedural statements employing them, is given in the sections on the Data and Procedure Divisions.

Mnemonic-names: A mnemonic-name is assigned in the Enviornment Division under SPECIAL-NAMES for reference in ACCEPT or DISPLAY statements. A mnemonic-name is composed according to the rules for Word Formation.

Procedure names: Procedure-names in the form of paragraph-names and section-names are words which identify paragraphs and sections, respectively, in the Procedure Division.

They may be up to 30 characters long, and may be all alphabetic, all numeric, or some combination of the two.

Literals

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

Non-numeric literals: A non-numeric literal must be bounded by matching quotation marks or apostrophes and may consist of any combination of characters in the ASCII set, except apostrophe or quotation marks,, 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'
"123"
'1001'
"3.1414"
'- 6'
"DO'S & DON'TS"
'PLEASE DON''T SQUEEZE THE CHARMIN'
```

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; length minimum is one.

A succession of two delimiters ('') within a literal is interpreted as a single representation of the delimiter within the literal. The last example above illustrates this point.

Only 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 conventions apply to the next line of coding (continuation line):

- A hyphen is placed in position 7 of the continuation line.
- A delimiter is placed in B Area preceding the continuation of the literal.

In the absence of continuation characters and delimeters, the non-numeric literal is required to continue for five lines. On any continuation line, A Area 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 (digits) 0 through 9 (optionally preceded by a sign) and/or the decimal point. It may contain only one sign character and only one decimal point. The sign, if present, must appear as the leftmost character of the numeric literal. If a numeric literal is unsigned, it is assumed to be positive.

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

If a literal conforms to the rules for the formation of numeric literals, but is enclosed in quotation marks, it is a nonnumeric literal and it is treated as such by the compiler.

The following are examples of numeric literals:

```
72   + 1Ø11   3.14159   - 6   - .333   Ø.5
```

By use of the Environment specfication DECIMAL-POINT IS COMMA, the functions of the period and comma characters 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.

Qualification of names

The user must be able to identify, uniquely, every name which defines an element in a COBOL source program. The name may be made unique in its spelling or hyphenation; or, procedural reference may be accomplished by use of qualifier names.

In the following example, the data-name, YEAR, will require qualification for procedural reference:

```
Ø1  EMPLOYEE-RECORD
    Ø5  NAME
    Ø5  ADDRESS
    Ø5  HIRE-DATE
        1Ø  YEAR
        1Ø  MONTH
        1Ø  DAY
    Ø5  TERMINATION-DATE
        1Ø  YEAR
        1Ø  MONTH
        1Ø  DAY
```

YEAR OF HIRE-DATE is a qualified reference which would differentiate between year fields in HIRE-DATE and TERMINATION-DATE.

Qualifiers are preceded by the word OF or IN. Successive data-name or condition-name qualifiers must designate lesser level-numbered groups which contain all preceding names in the composite reference. That is, HIRE-DATE must be a group item (or file-name) containing an item called YEAR. Paragraph-names may be qualified by their containing section-name. Therefore, two identical paragraph-names cannot appear in the same section.

The rules for qualification are:

- Each qualifier must be of a successively more inclusive level within the same hierarchy as the name it qualifies.
- The same name must not appear at two levels in a hierarchy.
- If a data-name or a condition-name is assigned to more than one item in a source program, the data-name or condition-name must be qualified each time it is referred to in the Procedure, Environment, and Data Divisions (except in the REDEFINES clause where qualification must not be used).
- A data-name cannot be subscripted when it is being used as a qualifier.
- A name can be qualified even though it does not need qualification. If more than one combination of qualifiers can make a name unique, any combination can be used. The complete set of qualifiers for a data name must not be the same as any partial set of qualifiers for another data-name.
- A qualified name may only be written in the Procedure Division.

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

Classes of data

The five categories of data-items (alphabetic, numeric, numeric edited, alphanumeric, and alphanumeric edited), as specified in the PICTURE clause, are grouped into three classes: alphabetic, numeric, and alphanumeric. For alphabetic and numeric data items, classes and categories are the same. The alphanumeric class includes the categories of alphanumeric edited, numeric edited and alphanumeric (without editing). Every elementary item except for an index data item belongs to one of the classes and, further, to one of the categories. The class of a group item is treated at object time as alphanumeric regardless of the class of elementary items subordinate to that group item. The following chart depicts the relationship of the class and categories of data items.

Level of data	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
		Numeric Edited
	Alphanumeric	Alphanumeric Edited
		Alphanumeric
Nonelementary (Group)	Alphanumeric	Alphabetic
		Numeric
		Numeric Edited
		Alphanumeric Edited
		Alphanumeric

Data levels

The two major levels of data are group and elementary:

Group item: A group item is defined as one having further subdivisions, so that it contains one or more elementary items. In addition, a group item may contain other groups. An item is a group item if, and only if, its level number is less than the level number of the immediately succeeding item. If an item is not a group item, then it is an elementary item. The maximum size of a group is 32,767 characters.

Elementary item: An elementary item is a data item containing no subordinate items. An elementary item must contain a PICTURE clause, except when usage is described as COMPUTATIONAL (binary), or INDEX.

Categories of data

The classes of data are: alphabetic, numeric, alphanumeric. Within these, the categories of data are: alphabetic, numeric, numeric edited, alphanumeric edited and alphanumeric.

Alphabetic item: An alphabetic item consists of any combination of the 26 characters of the English alphabet and the space character.

Numeric item: A maximum number of 18 digits is permitted; the exact number of digit positions is defined by the specification of 9's in the picture-string. For example, PICTURE 999 defines a 3-digit item whose maximum decimal value is nine hundred and ninety-nine.

Numeric edited item: An edited numeric item contains only digits and/or special editing characters. It must not exceed 30 characters in length. A numeric edited item can be used only as a receiving field for numeric data.

Alphanumeric edited item: This is an alphanumeric item with editing characters contained in the PICTURE description.

Alphanumeric item: An alphanumeric item consists of any combination of characters, making a character string.

Data representation

Data is further categorized by the format in which it is stored in the computer. The formats are: external decimal, internal decimal, binary and index. These formats are directly related to usage, as outlined below.

Usage is	Machine description
DISPLAY	External decimal
COMPUTATIONAL	Binary
INDEX	Binary
COMPUTATIONAL-3	Internal decimal

External decimal item: An external decimal item is one in which one byte (8 binary bits) is employed to represent one digit as well as the sign. It can be a group or an elementary item. The USAGE for an external decimal item is always DISPLAY.

Internal decimal item: An internal decimal item is packed decimal format. It is defined by inclusion of the COMPUTATIONAL-3 USAGE clause.

A packed decimal item defined by n 9's in its PICTURE occupies n/2+1 bytes in memory. All bytes, except the rightmost, contain a pair of digits, each digit being represented by the binary equivalent of a valid digit value from 0 to 9. For this reason, when using packed decimal, the optimum space allocation should be an odd size field.

In the rightmost byte of a packed item, the left half contains the item's low-order digit, while the right half contains a representation of the sign. An operational sign capability is always present for a packed field, even if the picture lacks the leading character S.

Binary item: A binary item uses the base 2 system to represent an integer not in excess of 32,767. It occupies one 16-bit word. The leftmost bit of the reserved area is the operational sign. No PICTURE clause is required; usage is COMPUTATIONAL. If a PICTURE clause is specified, and a decimal point is included, DISPLAY usage is assumed and a warning message is printed out.

Note that the user is responsible for aligning binary items on word boundaries.

Index item: An index item may not have a PICTURE clause. It also uses a 16-bit binary representation.

Standard aligment rules

The standard rules for positioning data within an elementary item depend on the category of the receiving item. These rules are:

1. If the receiving data item is described as numeric:
   - The data is aligned by decimal point and is moved to the receiving digit positions with zero fill or truncation at either end, as required.
   - When an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately following its rightmost digit. It is aligned as in the rule directly above.
2. If the receiving data item is numeric edited, the data moved to the edited data item is aligned by decimal point. Zero filling or truncation, at either

end, occurs as required within the receiving character postions of the data item, except where editing requirements cause replacement of the leading zeros.

3. If the receiving data item is alphanumeric (other than a numeric edited data item), alphanumeric edited or alphabetic, the sending data is moved to the receiving character positions and aligned at the leftmost character position in the data item. Space fill or truncation occurs to the right, as required.

If the JUSTIFIED clause is specified for the receiving item, these standard rules are modified as described under JUSTIFIED, Data Division. Examples:

(b =blank, ^ =implied decimal)

Data to be stored	Receiving field before transfer	Receiving field after transfer
ABC	PQRSTUVWXYZ	ABCbbbbbbbb
ABCDEF1234	PQRSTUVWXYZ	ABCDEF1234b
AAABBBCCCDD	PQRSTUVWXYZ	AAABBBCCCDD
AAABBBCCCDDDE	PQRSTUVWXYZ	AAABBBCCCDD

The examples above show the results of moving various length alphabetic and alphanumeric items into an eleven-character field.

Data to be stored	Receiving field before transfer	Receiving field after transfer
3^4	987^654	003^400
345^678	987^654	345^678
12345^67890	987^654	345^678
34^	987^654	034^000
3^4	ABC234	34bbbb
1234567890	987^654	890^000
1234567890	9876^54	7890^00

The examples above show the results of moving various length numeric items into a six-character field.

Algebraic signs

Algebraic signs fall into two categories: operational signs and editing signs. Operational signs are associated with signed numeric data items and signed numeric literals to indicate their algebraic properties. Editing signs appear on edited reports to identify the sign of the item.

The SIGN clause permits the programmer to state explicitly the location of the operational sign. Editing signs are inserted into a data item through the use of the control symbols of the PICTURE clause.

Subscripting

Subscripts can be used only when reference is made to an individual element within a list or table of like elements which have not been assigned individual data-names (see the OCCURS clause in DATA DIVISION and TABLE HANDLING).

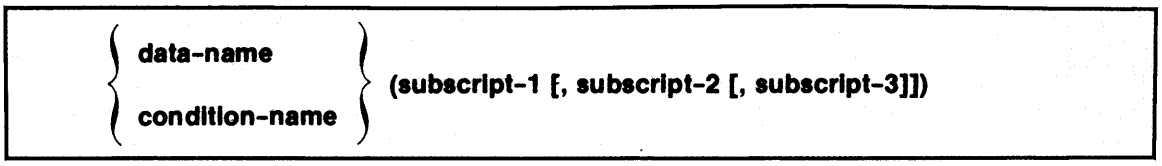
The subscript can be represented either by a numeric literal which is an integer, or by a data-name which may be qualified but not subscripted.

The subscript may be signed and, if signed, it must be positive. The lowest possible subscript value is 1. This value points to the first element of the table. The next sequential elements

of the table are pointed to by subscripts whose values are 2, 3, ... The highest permissible subscript value, in any particular case, is the maximum number of occurrences of the item as specified in the OCCURS clause.

The subscript which identifies the table element is delimited by the balanced pair of separators, left parenthesis and right parenthesis, following the table element data-name. When more than one subscipt is required, they are written in the order of successively less inclusive dimensions of the data-organization.

The format is:



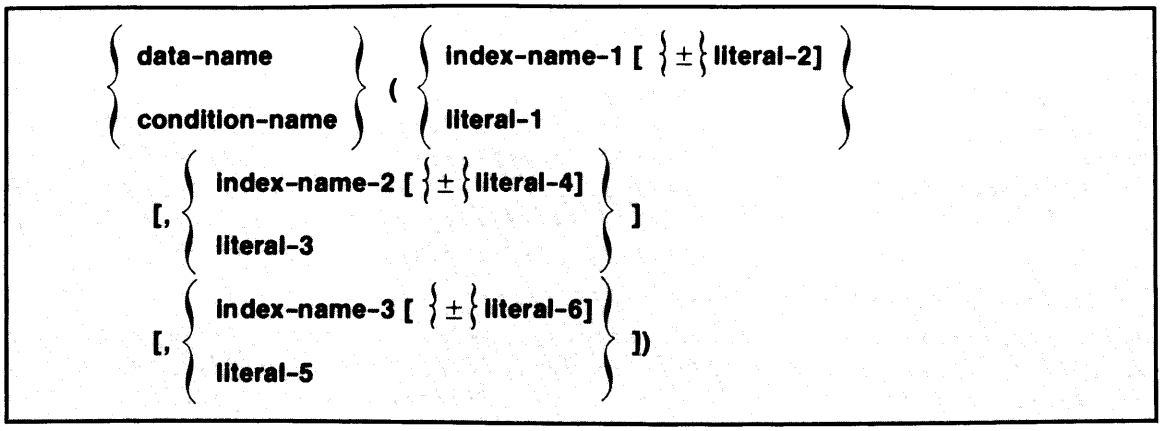
Indexing

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

Prime COBOL supports two types of indexing: direct and relative. Direct indexing is specified by using an index-name in the form of a subscript. Relative indexing is specified when the index-name is followed by a space, followed by one of the operators + or –, followed by another space, followed by an usigned integer numeric literal all delimited by the balanced pair of separators left parenthesis and right parenthesis following the table element data-name. The occurrence number resulting from relative indexing is determined by incrementing or decrementing by the value of the literal, the occurrence number represented by the value of the index. When more than one index-name is required, they are written in the order of successively less inclusive dimensions of the data organization.

When a statement, which refers to an indexed table element,is executed, the value in the associated index must neither be less than zero, nor greater than the highest occurrence number of an element in the table. This restriction also applies to the values resultant from relative indexing.

The general format for indexing is:



Restrictions on qualification, subscripting and indexing.

- A data-name must not itself be subscripted nor indexed when that data-name is being used as an index, subscript or qualifier.
- Indexing is not permitted where subscripting is not permitted.
- An index may be modified only by the SET, SEARCH, and PERFORM statements. Data items described by the USAGE IS INDEX clause permit storage of the values associated with index-names. Such data items are called index data items.

ARITHMETIC EXPRESSIONS

Definition

An arithmetic expression must be an identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses. Any arithmetic expression may be preceded by a unary operator. The permissible combinations of variables, numeric literals, arithemetic operators and parentheses are given in Table 4-1.

Table 4-1. Symbol combinations in Arithmetic Expressions

FIRST SYMBOL	SECOND SYMBOL				
	Variable	* / – +	Unary + OR –	(	)
Variable	X	P	X	X	P
* / + –	P	X	P	P	X
Unary + or –	P	X	X	P	X
(	P	X	P	P	X
)	X	P	X	X	P

In the table above, P = permissible, X = invalid, Variable indicates an identifier or literal.

Identifiers and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic may be performed.

Arithmetic operators

The specific characters below represent the binary and unary arithmetic operators. They must be preceded and followed by at least one space.

Binary arithmetic	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
Unary arithmetic	**Meaning**
+	The effect of multiplication by numeric literal +1.
–	The effect of multiplication by numeric literal –1.
Parenthesis	**Meaning**
( )	Used to enclose expressions to control the sequence in which conditions are evaluated.

Follow these general rules on arithmetic expressions:

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first; and within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive set. When

parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of execution is implied:

1st - Unary plus and minus
2nd - Multiplication and Division
3rd - Addition and Subtraction

When the sequence of execution is not specfied by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right. Example:

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

This expression is evaluated in the following ordered sequence:

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

Without parentheses, the expression

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

is evaluated as:

R1 = B/C
R2 = A+R1
R3 = D*E

final result = R2 – R3

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

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

The expressions 'A – B – C' is evaluated as '(A – B) – C'. Unary operators are permitted, Example:

```
COMPUTE A = + C + 4.6    COMPUTE X = - Y
```

Operators, variables, and parenthesis may be combined in arithmetic expressions as summarized in Table 4-1.

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

Arithmetic statements

The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements. These have several common features.

1. The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied throughout the calculation.
2. The maximum size of each operand is 18 decimal digits. The composite of operands, which is a hypothetical data item resulting from the superimposition of specified operands in a statement aligned on their decimal points, must not contain more than 18 decimal digits.

Overlapping operands

When a sending and a receiving item in an arithmetic statement or an INSPECT, MOVE, SET, STRING, UNSTRING, or other statements share a part of their storage areas, the result of the execution of such a statement is undefined and unpredictable.

CONDITIONAL EXPRESSIONS

Definition

Conditional expressions identify conditions which are nested to enable the object program to select between alternate paths of control depending upon the truth value of the condition. Conditional expressions are specified in the IF, PERFORM, and SEARCH statements.

Simple conditions

The simple conditions are the relation, class, condition-name, and sign conditions. A simple condition has a truth value of 'true' or 'false'. The inclusion in parentheses of simple conditions does not change the simple truth value.

Relation condition: A relation condition has this format:

```
operand relation operand
```

where operand is a data-name, literal or figurative-constant. A relation condition has a truth value of 'true' if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses. However, for all other comparisons, the operands must have the same usage.

Relation has three basic forms, expressed by the relational symbols: equals (=), less than (<), or greater than (>).

Relational Operator	Meaning
=	is equal to
<	is less than
>	is greater than
NOT =	is not equal to
NOT <	is greater than, or equal to
NOT >	is less than, or equal to

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

Note

Although required where indicated in formats, the relational characters ' < ', ' > ', and '=' are not underlined in this text.

The first operand of a conditional expression is called the subject of the condition; the second operand is called the object of the condition. The relation condition must contain at least one reference to a variable.

The relational operator specifies the type of comparison to be made in a relation condition. A space must precede and follow each reserved word comprisng the relational operator. When used, 'NOT' and the next key word or relation character form one relational operator defining the comparison to be executed for truth value; e.g., 'NOT EQUAL' is a truth test for an 'unequal' comparison; 'NOT GREATER' is a truth test for an 'equal' or 'less' comparison.

Numeric comparisons: For numeric operands, a comparison is made with respect to their algebraic value. The length of the literal or arithmetic expression operands, in terms of number of digits represented, is not significant. Zero is considered a unique value regardless of the sign.

Comparison of these operands is permitted irrespective of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparison.

The data operands are compared after assignment of their decimal positions.

An index-name or index item may appear in a numeric comparison

(See Section 7 for details.)

Non-numeric comparisons: For non-numeric operands, a comparison is made with respect to Prime collating sequence of characters. The octal value associated with each ASCII character in the Prime computer is the basis for the sequence. (Refer to Appendix C for all ASCII character representations and the Prime collating sequence.)

If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

The data class (see Data Representation of this Section) of the two operands, where one is a literal, must be the same.

Class condition: The class condition determines whether the contents of a data-name are numeric or alphabetic. A numeric data item consists entirely of the digits 0 through 9, with or without the operational sign. An alphabetic data item consists entirely of the alphabetic characters A through Z and the space. The general format for the class conditions is:

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

The data-name must be described, implicitly or explicity, as USAGE IS DISPLAY.

The NUMERIC test cannot be used with a data-name described as alphabetic or as a group item composed of signed elementary items.

If the PICTURE clause of the data-name being tested does not contain an operational sign, the data-name is determined to be numeric only if the contents are numeric and an operational sign is not present.

If the PICTURE clause of the data-name being tested does contain an operational sign, the data-name is determined to be numeric only if the contents are numeric and a valid operational sign is present.

The ALPHABETIC test cannot be used with a data-name described as numeric. The data-name being tested is determined to be alphabetic only if the contents consists of any combination of the alphabetic characters and the space.

Condition-name condition: In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name. The general format for the condition-name condition is as follows, where condition-name is defined by a level 88 Data Division entry:

```
IF condition-name statement(s)
```

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values. (See Section 7 for details.)

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

The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable. Condition-names are allowed in the File Section and Linkage Section where VALUE clauses are not.

Sign condition: The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero. The general format for a sign condition is as follows:

```
data-name IS [NOT] { POSITIVE }
                   { NEGATIVE }
                   { ZERO     }
```

Complex conditions

A complex condition is a concatenation of simple conditions, combined conditions and/or complex conditions with logical connectors (logical operators 'AND' and 'OR') or negating these conditions with logical negation (the logical operator 'NOT').The truth of a complex condition is that truth value which results from the interaction of all the stated logical operators on the individual truth values of simple conditions, or the intermediate truth values of conditions logically connected or logically negated. Five levels of parentheses are permitted in complex conditions.

The logical operators are:

Logical operator	Meaning
AND	Logical conjunction; the truth value is 'true' if both of the conjoined conditions are true; 'false' if one or both of the conjoined conditions is false.
OR	Logical inclusive OR; the truth value is 'true' if one or both of the included conditions is true; 'false' if both included conditions are false.
NOT	Logical negation is the reversal of the truth value; i.e., the truth value is 'true' if the condition is false, and 'false' if condition is true.

Logical operators must be preceded and followed by a space.

Negated simple conditions: The general format of a negated simple condition is:

```
NOT simple-condition
```

Thus, the simple condition is negated through the use of the logical operator NOT.

The truth value of a negated simple condition is the opposite of the truth value for a simple condition; i.e., true if the simple condition is false,and false if the simple condition is true.

Inclusion in parenthesis of a negated simple condition does not affect the truth value.

Combined and negated combined conditions: Combined conditions are simple conditions connected by one of the logical operators AND or OR. A combined condition has the format:

```
condition-1 { AND } [NOT] condition-2
            { OR  }
```

where condition is:

- A simple condition
- A negated simple condition
- A combined condition
- A negated combined condtion, i.e., the logical operator NOT followed by a combined condition enclosed in parentheses
- Combinations of the above.

Table 4-2 below sets forth the permissible combinations of conditions, logical operators and parentheses.

Table 4-2. Permissible combinations of conditions, logical operations and parentheses

ELEMENT	Conditional Expression Location		Using a left to right sequence of elements	
			When not first, the element can be immediately preceded only by:	*When not last, the element can be immediately followed only by:*
	First	*Last*		
Simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR and AND	No	No	Simple-condition,)	Simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	Simple-condition, (
(	Yes	No	OR, NOT, AND, (	Simple-condition, NOT, (
)	No	Yes	Simple-condition,)	OR, AND,)

Multiple conditions: Multiple conditions refer to complex conditions grouped in parentheses; as previously stated, parentheses are permitted to five levels.

When more than five levels of parentheses are required, explicit grouping, condition-names, nested IF statements, or some combination of the above should be substituted.

For example, in the statement

IF a = b AND (c = d OR e = f)

explicit grouping may be achieved by coding

IF a = b AND c = d OR a = b AND e = f

Abbreviated combined relation conditions

Abbreviated combined relation conditions refer to conditions with implied subjects. That is, the omission of the subject of the relation, or the omission of both the subject and the relational operator of the relation condition.

The format for a abbreviated combined relation condition is:

relation-condition { **AND** / **OR** } **[NOT] [relational-operator] object ...**

Within a sequence as described above, either form of abbreviation may be used: the omission of subject, or the omission of subject and relational operator.

The effect of such abbreviations is that of inserting the previously stated subject in place of the omitted subject, or the previous stated relational operator.

All insertions terminate once a complete simple condition is encountered within a complex condition.

In all instances, the results must comply with the rules outlined in Table 4-2 above.

Negated relation conditions arise from the use of the word NOT in an abbreviated combined relation condition. They are evaluated as follows:

- NOT participates as part of the relational operator if the word immediately following NOT is GREATER, >, LESS, <, EQUAL, or eq;
- Not is interpreted as a logical operator if the above condition does not apply, with the result that the implied insertion of subject or relational operator results in a negated relation condition.

Below are examples of abbreviated combined relation conditions:

Abbreviated Combined and Negated Combined Relation Conditions	**Expanded Equivalent**
`a = b OR c OR d`	`a = b OR a = c OR a = d`
`a > b AND NOT < c OR d`	`((a > b) AND (a NOT < c)) or (a NOT < d)`
`NOT a = b OR c`	`(NOT (a = b)) OR (a = c)`
`a NOT EQUAL b OR c`	`(a NOT EQUAL b) OR (a NOT EQUAL c)`
`NOT (a GREATER b OR < c)`	`NOT ((a GREATER b) OR (a < c))`
`NOT (a NOT > b AND c AND NOT d)`	`NOT ((( a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d)))`

Note

The reader is cautioned about the ambiguities which arise from such coding.

Condition evaluation rules

Parentheses can be used to specify the order in which individual conditions of complex conditions can be evaluated when it is necessary to depart from the implied evaluation precedence. Conditions within parentheses are evaluated first, and, within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When parentheses are not used, or when parenthesized conditions are at the same level of inclusiveness, the following hierarchical order of logical evaluation is implied until the final truth value is determined.

1. Truth values for simple conditions are evaluated in the following order:
   - Relation (following the expansion of any abbreviated relation condition)
   - Class
   - Condition-name
   - Sign
2. Truth values for negated simple conditions are established.
3. Truth values for combined conditions are established:
   - AND logical operators, followed by
   - OR logical operators
4. Truth values for negated combined conditions are established.

5. When the sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations of the same hierarchical level is from left to right.

The following examples apply to the condition evaluation rules:

1. The condition below contains both AND and OR connectors.

```
IF x = y AND FLAG = 'z' OR SWITCH = Ø, GO TO PROCESSING
```

Execution will be as follows, depending on various data values:

	Data Value			EXECUTES
X	Y	FLAG	SWITCH	PROCESSING
10	10	'Z'	1	YES
10	11	'Z'	1	NO
10	11	'Z'	0	YES
10	10	'p'	1	NO
6	3	'p'	0	YES
6	6	'p'	1	NO

2. A < B OR C = D OR E NOT > F: The elvaluation is equivalent to (A < B) OR (C = D) OR NOT (E < F) and is true if any of the three individual parenthesized simple conditions is true.
3. WEEKLY AND HOURS NOT = 0: The evaluation is equivalent, after expanding level 88 condition-name WEEKLY, to (PAY-CODE = 'W') AND NOT (HOURS = 0) and is true only if both the simple conditions are true
4. A = 1 AND B = 2 AND G > −3 OR P NOT EQUAL TO "SPAIN": is evaluated as

   (A = 1) AND (B = 2) AND (G < −3) OR NOT (P = "SPAIN")

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

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

5

Identification division

IDENTIFICATION DIVISION

Function

The Identification Division *must* be included in every COBOL source program as the first entry. This division identifies the source program and the resultant output listings. Additional user information, such as the date the program was written or the program author, may be included under the appropriate paragraph(s) in the general format shown below.

Format

```
ID DIVISION. (or IDENTIFICATION DIVISION.)

PROGRAM-ID. program-name. (no special characters in name)

[AUTHOR. comments.]

[INSTALLATION. comments.]

[DATE-WRITTEN. comments]

[DATE-COMPILED. comments.]

[SECURITY. comments.]

REMARKS. comments.]
```

Syntax rules

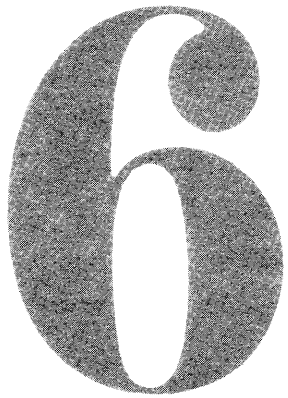
1. The Identification Division must begin with **ID DIVISION** or **IDENTIFICATION DIVISION** followed by a period and a space.
2. The **PROGRAM-ID** paragraph is required and *must* follow immediately after the division header.
3. **Program-name** follows the general rules for Word Formation. It may be any alphanumeric string. However, the first character must be alphabetic. Special characters, including the hyphen, are prohibited. (Only the first six characters of program-name are retained by the compiler.)
4. All remaining paragraphs are optional. When included, these must be presented in order shown above.
5. The comments entry can be any combination of characters. Use of the hyphen in the continuation indicator area is not permitted; however, the comments entry can appear on one or more lines.

▶ General rule

Fixed paragraph names identify the type of information contained in the paragraph.

▶ Example

```
ID DIVISION.
PROGRAM-ID. REF2.
AUTHOR. PRIME COMPUTER.
INSTALLATION. CORPORATE TECHNICAL PUBLICATIONS DIVISION.
DATE-WRITTEN. SEPTEMBER 1, 1979.
DATE-COMPILED. SEPTEMBER 1, 1979.
SECURITY. NONE.
REMARKS. THIS AREA IS USED TO DESCRIBE THE PROGRAM.
```



Environment division

ENVIRONMENT DIVISION

Function

The Environment Division defines those aspects of a data processing problem which are dependent upon hardware configurations and considerations.

Format

```
ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.

[SOURCE-COMPUTER. computer-name.]

[OBJECT-COMPUTER. computer-name.]

[SPECIAL-NAMES. [CONSOLE IS mnemonic-name]

    [, CURRENCY SIGN IS literal]

    [, DECIMAL-POINT IS COMMA]

    [, ASCII IS NATIVE]]. ]

[INPUT-OUTPUT SECTION.

 FILE-CONTROL.

    {SELECT file-name ASSIGN TO device

    [; RESERVE integer-1 [AREA  ]  ]
                         [AREAS ]

    [; ORGANIZATION IS {SEQUENTIAL} ]
                       {INDEXED    }
                       {RELATIVE   }

    [; ACCESS MODE IS {SEQUENTIAL} ]
                      {RANDOM     }
                      {DYNAMIC    }

    [; FILE STATUS IS data-name-1]} ...
```

[I-O-CONTROL.

SAME AREA FOR file-name-1{, file-name-2}, ...]]

▶ Syntax rules

1. The Environment Division *must* begin with the header, ENVIRONMENT DIVISION, followed by a period and a space.
2. Mandatory sequence of required and optional paragraphs is shown in the above format.

Note

In the rare instance when hardware-dependent configurations and consideratins do not apply, the entire ENVIRONMENT DIVISION may be omitted. However, the header, ENVIRONMENT DIVISION, must be presented all the time.

▶ General rule

Each section within the Environment Division begins with its section-name, followed by the word SECTION, and each paragraph within each section begins with its paragraph-name.

[CONFIGURATION SECTION.

This section is optional. It is required only if one or more of the following three paragraphs is used.

1. **[SOURCE-COMPUTER. computer-name.]**

   Computer-name serves only as a comment entry. It is used to identify the computer for which the COBOL program is written.

2. **[OBJECT-COMPUTER. computer-name.]**

   Computer-name serves only as a comment entry. It is used to identify the computer on which the COBOL program will be executed.

3. **[SPECIAL-NAMES.**

   This paragraph is optional. It is required only if one or more of the following four statements is used.

   - **[CONSOLE IS mnemonic-name]**

     Mnemonic-name is a programmer-defined word which will be associated with CONSOLE throughout the program.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.  CONSOLE IS TTY.
```

```
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
DISPLAY YEAR OF HIRE-DATE UPON TTY.
```

The coding above would cause the field, YEAR OF HIRE-DATE, to be output on the CONSOLE.

Note

CONSOLE IS is an optional statement. If omitted, the computer will automatically associate CONSOLE (terminal) with ACCEPT and DISPLAY.

- **[CURRENCY SIGN IS literal]**

  Literal represents the currency sign to be used in the PICTURE clause. It is a single character, non-numeric literal which will be used to replace the dollar sign as the currency sign. The designated character may not be a quote mark, or any of the characters defined for PICTURE representations.

- **[DECIMAL-POINT IS COMMA]**

  The "European" convention of separating integer and fraction positions of numbers by the comma character, rather than the decimal point or period, is specified by use of the DECIMAL-POINT IS COMMA clause.

Note

The Reserved Word, IS, is required in entries for currency sign definition and decimal-point convention specification.

- **[ASCII IS NATIVE]].]**

  The entry, ASCII IS NATIVE, specifies that the data representation adheres to the American Standard Code for Information Interchange as shown in Appendix C. This convention is assumed even if the entry is not present.

[INPUT-OUTPUT SECTION.

The INPUT-OUTPUT SECTION is used when there are external data files. It allows specification of peripheral devices and information needed to transmit and handle data between the devices and the program. The section has two paragraphs: FILE-CONTROL and I-O-CONTROL.

FILE-CONTROL.

This entry names each file and specifies its device medium, allowing specified hardware assignments. It can also specify other file-related information, such as number of input-output areas allocated, file organization, and method of file access. The format chosen is

dependent upon file organization. Each file requires one SELECT statement and the appropriate sequence of optional clauses.

Format one

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

Format two

```
SELECT file-name
    ASSIGN TO device
        [; RESERVE integer-1 [ AREA | AREAS ] ]
        ; ORGANIZATION IS RELATIVE
        [; ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name-1] |
                            { RANDOM | DYNAMIC } , RELATIVE KEY IS data-name-1 } ]
        [; FILE STATUS IS data-name-2].
```

Format three

```
SELECT file-name
    ASSIGN TO device
        [; RESERVE integer-1 [ AREA | AREAS ] ]
        ; ORGANIZATION IS INDEXED
```

```
[; ACCESS MODE IS { SEQUENTIAL } ]
                  { RANDOM     }
                  { DYNAMIC    }

; RECORD KEY IS data-name-1

[; ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] ...

[; FILE STATUS IS data-name-3].
```

Format four

```
SELECT file-name

    ASSIGN TO device
```

1. **SELECT file-name ASSIGN TO device**

File-name is a programmer-defined name described in the Data Division. Each Data Division FD entry must be specified once in a SELECT statement and only as a file-name. The **ASSIGN TO device** clause associates the file with a storage medium or input/output hardware.

Device	Hardware Device
TERMINAL	CRT TERMINAL
	TTY TERMINAL
READER	CARD READER *(for future designation)*
PRINTER	SYSTEM PRINTER
PUNCH	CARD PUNCH *(for future designation)*
MT9	9 TRACK MAG. TAPE DRIVE
PFMS *	DISK STORAGE
OFFLINE-PRINT	FORMS PRINTER OUTPUT

* PFMS = PRIME FILE MANAGEMENT SYSTEM

Examples:

```
SELECT file-name ASSIGN TO TERMINAL.
SELECT file-name ASSIGN TO PFMS.
SELECT file-name ASSIGN TO MT9.
```

2. **[RESERVE integer-1** **[AREA]** **]**
[AREAS]

The RESERVE clause allows the user to specify the number of input-output buffer areas to be allocated. For tape applications only, the **integer** value can be from 1 to 7, permitting up to 7 buffers in memory at one time.
If tape is not involved, the integer must be specified as one. Should the RESERVE clause be omitted, the default of one buffer area will be assigned by the compiler.

3. **[ORGANIZATION IS** { **SEQUENTIAL** / **RELATIVE** / **INDEX** } **]**

The ORGANIZATION clause specifies the type of file organization. When omitted, the default is **SEQUENTIAL**.

4. **[ACCESS MODE IS** { **SEQUENTIAL** / **RANDOM** / **DYNAMIC** } **]**

The sequence in which records are accessed is described through the use of the ACCESS MODE clause. When omitted, the default is **SEQUENTIAL.**

5. **[FILE STATUS IS data-name-1]**

The FILE STATUS clause permits the user to specify a two character, unsigned field (**data-name-1**) described in the Working-Storage Section.
When the FILE STATUS clause is specified in the FILE-CONTROL paragraph, the operating system moves a value into data-name-1 after the execution of every statement which references that file either explicitly or implicitly. Specifically, the FILE STATUS data item is updated during the execution of the OPEN, CLOSE, READ, WRITE, REWRITE, DELETE or START statement. This value in data-name-1 indicates to the COBOL program the status of execution of the statement. The left most character of the FILE STATUS data item is known as status key 1; the rightmost character is status key 2. Status key 1 is set to indicate a specific condition upon completion of the input-output operation; status 2 further describes the results of the operation.
Valid combinations of key values for each type of file organization are shown in the File Status Key Definitions, Table C-4, Appendix C.

[I-O-CONTROL.

The I-O-CONTROL paragraph is optional unless SAME AREA is used.

The SAME AREA clause allows the programmer to share the same I/O buffer areas for files which are not open concurrently. No file may be listed in more than one SAME AREA clause.

Example

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. PRIME-750.
OBJECT-COMPUTER. PRIME-750.
SPECIAL-NAMES. CONSOLE IS TTY ,
               ASCII IS NATIVE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PRINT-FILE ASSIGN TO PRINTER.
    SELECT CARD-FILE ASSIGN TO PFMS.
    SELECT DIRECTORY-FILE ASSIGN TO PFMS,
           ORGANIZATION IS INDEXED,
           ACCESS MODE IS DYNAMIC,
           RECORD KEY IS PHONE-NUMBER,
           ALTERNATE RECORD KEY LAST-NAME,
           ALTERNATE RECORD KEY STATE,
           ALTERNATE RECORD KEY BIRTH-DATE,
           ALTERNATE RECORD KEY FIRST-NAME,
           FILE STATUS IS FILE-STATUS.
```

7

Data division

DATA DIVISION

Function

The Data Division of the COBOL source program defines the nature and characteristics of the data to be processed by the program. Data to be processed falls into three categories:

1. Data is contained in files and enters or leaves the internal memory of the computer from a specified area or areas.
2. Data is developed internally and placed into intermediate or working storage.
3. Constants which are defined by the user.

The Data Division consists of three optional sections. If used, they must appear in the following order:

1. **FILE SECTION.** Files and records in files are described.
2. **WORKING-STORAGE SECTION.** Memory space is allocated for the storage of intermediate processing results.
3. **LINKAGE SECTION.** Data available to a called program is described.

Format

```
DATA DIVISION.

[FILE SECTION.

[file-description-entry.
[record-description-entry] ... ] ...
[sort-file-description-entry.
{record-description-entry}... ] ... ]

[WORKING-STORAGE SECTION.

[level-77-data-description-entry] ...
[data-item-description-entry] ... ]

[LINKAGE SECTION.

[level-77-data-description-entry] ...
[data-item-description-entry] ... ]
```

Syntax rules

1. The Data Division must begin with the header DATA DIVISION, followed by a period and a space.

2. When included, optional sections of the Data Division must be in the same order as shown above.

▶ General rules

1. Each section within the Data Division begins with its section-name, followed by a period and a space.
2. The record description entry format used in the File Section is also applied to the Working-Storage and Linkage Sections.

FILE SECTION

▶ Function

The file section of the Data Division defines the structure of data files.Each file is defined by a file description entry (FD) or a sort file description entry (SD), and by one or more associated record description entries.

Format

```
[ FILE SECTION.
    [ file-description-entry. [record-description-entry] ...
      sort-file-description-entry. { record-description-entry }... ] ... ]
```

▶ Syntax rules

1. The File Section is optional. If used, it must begin with the header, FILE SECTION, followed by a period and a space.
2. The File Section contains FD and SD entries, each one must be immediately followed by one or more associated record description entries. The total number of FD and/or SD entries in the File Section cannot exceed 126.

▶ General rule

Each file associated with an I/O device must be represented by an FD or an SD entry.

Note

The format and the clauses required in an FD entry for a typical file are described in this section. For a complete discussion of an SD entry for a sort-file, see section 11, SORT MODULE.

FILE DESCRIPTION

▶ Function

The FD file description provides information concerning the physical structure, identification, and record names pertaining to a typical file.

Format

```
[FD file-name [UNCOMPRESSED]
;LABEL { RECORD IS    } { STANDARD }
       { RECORDS ARE } { OMITTED  }
[;BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS    } ]
                                         { CHARACTERS }
[;RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]
[;VALUE OF FILE-ID IS literal-1]
[;OWNER IS literal-2]
[;DATA { RECORD IS    } data-name-1 [, data-name-2] ...]
       { RECORDS ARE }
[;CODE-SET IS ASCII]].
```

Syntax rules

1. The level indicator **FD** identifies the beginning of a file description and must precede the **file-name**.
2. File-name follows the general rules for Word Formation.
3. The **UNCOMPRESSED** option is used only with READ files. It allows a PRWFIL READ, rather than an RDASC READ.
4. The FD entry is a sequence of clauses which must be terminated by a period; up to 126 FD entries are permitted.
5. The **LABEL RECORD** clause is required; other clauses which follow file-name are optional.
6. If the **DATA RECORD** clause is used, one or more record description entries must follow the file description entry.
7. These rules apply to the overall File Section of a typical file.
8. If there are no files in the program or if the main program contains an EXIT PROGRAM statement then C$IN will not ask for file assignments and will take the default **VALUE OF FILE-ID** value as defined in the FD. If there is not a VALUE OF FILE-ID in the program the compiler will generate files with the name F1, F2, F3...etc.

UNCOMPRESSED

Function

The UNCOMPRESSED clause enables a disk READ based on record length, rather than on compression control characters.

Format

```
FD file-name [UNCOMPRESSED]
```

General rules

1. The **UNCOMPRESSED** clause is optional. When used, it enables a READ based on record length (PRWFIL), rather than compression control characters (RDASC).
2. The UNCOMPRESSED option must be used when reading sequential I/O files containing packed or binary data.

Note

The UNCOMPRESSED reserved word is non-ANSI and is peculiar to Prime.

Never use the standard utilities such as EDITOR on a file which is to be accessed UNCOMPRESSED for a COBOL program.

LABEL RECORDS

Function

The LABEL RECORDS clause specifies whether labels are present for the file.

Format

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

Syntax rule

This clause is required in every file description entry.

General rules

1. **OMITTED** specifies that no explicit labels exist for the file or device to which the file is assigned.
2. **STANDARD** specifies that a label exists for the file and that the label conforms to system specifications. The STANDARD option must be specified for all files assigned to DISK (PFMS) or tape. See Table 7-1 below.

Note

Standard labels are automatically provided for disk files. See Appendix G, LABEL COMMAND, for information on standard labels for magtape.

Table 7-1. Label Options

Device	Standard	Omitted
Terminal		✔
Reader		✔
Printer		✔
Punch		✔
MT9 (Tape)	✔	
PFMS (Disk)	✔	

BLOCK CONTAINS

Function

The BLOCK CONTAINS clause specifies the size of a physical record.

Format

```
[BLOCK CONTAINS [integer-1 TO ] integer-2 { RECORDS    } ]
                                          { CHARACTERS }
```

▶ Syntax rules

1. The **BLOCK CONTAINS** clause is optional.
2. The clause can only be used in connection with tape files.

▶ General rules

1. The clause may be omitted if the physical record contains one, and only one, complete logical record.
2. Omission of this clause assumes records are unblocked.
3. When the **RECORDS** option is used, the compiler assumes that the block size provides for **integer-2** records of maximum size and then provides additional space for any required control words.
4. When the word **CHARACTERS** is specified, the physical record size is specified in terms of the number of character positions required to store the physical record, regardless of the types of characters used to represent the items within the physical record.
5. When neither the CHARACTERS nor the RECORDS option is specified, the CHARACTERS option is assumed.
6. When both **integer-1** and integer-2 are used, integer-1 is for documentation purpose only.

RECORD CONTAINS

▶ Function

The RECORD CONTAINS clause specifies the size of data records.

Format

```
RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
```

▶ General rules

1. Since the size of each data record is defined fully by the set of data description entries constituting the record (level 01) declaration, this clause is always optional.
2. **Integer-4** may not be used by itself unless all the data records in the file have the same size. In this case, integer-4 represents the exact number of characters in the data record. If **integer-3** and integer-4 are both shown, they refer to the minimum number of characters in the smallest size data record, and the maximum number of characters in the largest size data record, respectively.
3. The maximum size of a single data record is 32,767 characters.

VALUE OF FILE - ID

Function

The VALUE OF FILE-ID clause particularizes the description of an item in the label records associated with a file, thus allowing for the linkage of internal and external program names.

Format

```
[VALUE OF FILE-ID is literal-1]
```

Syntax rule

This clause is mandatory if labels are standard.

General rules

1. **Literal-1** associates the internal FD file-name with an external file-name. It is a non-numeric value which may not exceed eight characters.
2. If there are no file assignments at run-time (explained in Section 3), literal-1 will become the default value for the internal file-name.

OWNER IS

Function

The OWNER IS clause specifies the User File Directory (UFD) in a Prime system, in which VALUE OF FILE-ID value is contained.

Format

```
[OWNER is literal-2]
```

Syntax rule

The **OWNER IS** clause may be used only with disk files.

General rules

1. **Literal-2** is a non-numeric value which may not exceed six characters.
2. The clause is overridden by explicit definition at run-time.
3. If the clause is used, it must follow the above rules. If omitted, a default of the current UFD may apply.

DATA RECORDS

Function

The DATA RECORDS clause serves only as documentation for the names of data records and their associated file.

Format

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

▶ Syntax rule

Data-name-1 and **data-name-2** are the names of the data records. They must be specified by subsequent 01 level-numbers and follow the general rules for Word Formation.

▶ General rules

1. The presence of more than one data-name indicates that the file contains more than one type of data record. These records may have different sizes, different formats, etc. The order in which they are listed is not significant.
2. Conceptually, all data records within a file share the same area, regardless of the number of types of data records within the file.

CODE - SET

▶ Function

The CODE-SET clause specifies the character code set used to represent data on the external media.

Format

```
[CODE-SET IS ASCII]].
```

▶ General rule

The **CODE-SET** clause serves only as documentation in this compiler.

RECORD DESCRIPTION

▶ Function

A record description entry describes all elementary and group items in a record, and their relationship. It is comprised of a set of data description entries, each of which defines the particular characteristics of a unit of data, utilizing a series of clauses to detail such characteristics.

Format one

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

[; OCCURS integer-1 TIMES
      [ { ASCENDING  } KEY IS data-name-3 [, data-name-4] ...]
        { DESCENDING }
   [ INDEXED BY index-name-1 [, index-name-2] ...]]

[; { PICTURE } IS picture-string] (or character-string)
   { PIC     }

[; [USAGE IS] { DISPLAY             } ]
              { COMPUTATIONAL       }
              { COMP                }
              { INDEX               }
              { COMPUTATIONAL-3     }
              { COMP-3              }

[; [SIGN IS ] { LEADING  } [SEPARATE CHARACTER] ]
              { TRAILING }

[; { SYNCHRONIZED } [ LEFT  ] ]
   { SYNC         } [ RIGHT ]

[; { JUSTIFIED } RIGHT ]
   { JUST      }

[; BLANK WHEN ZERO]

[; VALUE IS literal].
```

Format two

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

Format three

```
88 condition-name; { VALUE IS    } literal-1 [ { THROUGH } literal-2]
                   { VALUES ARE }             { THRU    }

      [, literal-3 [ { THROUGH } literal-4]] ... .
                     { THRU    }
```

Format four

```
88 condition-name; { VALUE IS    } literal-1 [, literal-2] ... .
                   { VALUES ARE }
```

Syntax rules

1. The **level-number** in Format one may contain a value of 01 through 49, or 77.
2. In Format one, clauses can be written in any order with two exceptions: The **data-name-1** or **FILLER** clause must immediately follow the level-number; and the **REDEFINES** clause, when used, must immediately follow the data-name-1 clause.
3. In Format one, **PICTURE** clause must be specified for every elementary item except when **USAGE** is described as binary (**COMPUTATIONAL**). A group item cannot contain a PICTURE clause.
4. The OCCURS clause cannot be specified in a data description entry which has a 01, 66, 77, or an 88 level-number.
5. Format two permits alternative possible overlapping groups of elementary items.
6. Formats three and four are used only for condition-names which must have a level-number 88. Formats three and four may not be combined for a single level 88 entry.
7. The words THRU and THROUGH are equivalent and interchangeable Reserved Words.

General rule

A record description entry can appear in the File, Working-Storage, or Linkage Section of the Data Division. All records in each file referenced by a file description entry (FD) must be described by record description entries.

Function

The level-number shows the position of a data-item within the hierarchy of data in a logical record. It also identifies entries for condition-names, and data items in the Working-Storage and Linkage Sections.

Format

```
level-number
```

Syntax rules

1. A **level-number** is required as the first element in each data description entry (see RECORD DESCRIPTION).
2. Data description entries subordinate to an FD entry must have level-numbers 01 through 49, 66, or 88.
3. Data description entries in the Working-Storage and Linkage Sections must have level-numbers 01 through 49, 66, 77 or 88.

General rules

1. Level-numbers are used to subdivide a record so that each item in the record may be referred to. A record can be divided, and each subdivision further divided, until a basic level is reached which cannot be further divided. An item at this basic level is called an elementary item. A record can itself be an elementary item.
2. A group consists of one or more consecutive elementary items; groups can, in turn, be combined into other groups of two or more group items. A group consists of a specified group item and all following group and elementary items with level-numbers greater than that of the specified group item, and continuing until the next item with a level-number less than or equal to that of the specified group item is reached.
3. Level-numbers range from 01, the most inclusive level, to 49, the least inclusive level. Any level-number except 49 can denote a group.
4. The level number 01 identifies the first entry in each Data Description. A reference to level-number 01 data-name in the Procedure Division is a reference to the entire record.
5. Multiple level 01 entries subordinate to one FD level indicator represent implicit redefinitions of the same area.
6. Special level-numbers have been assigned to certain entries where there is no real concept of hierarchy.
7. Level-number 77 is assigned to identify noncontiguous working storage or linkage data items. They may be used only as described in Format one of the data description entry.

   Level-number 77 data items are elementary items which cannot be subdivided.
8. Level-number 88 is assigned to entries which define condition-names associated with a conditional variable. They can be used only with Formats three and four of the data description entry.

Level 88 entries can contain individual values, series of individual values, or a range of values. Such entries cannot combine ranges and individual values.

Example:

```
Ø1  Test-Area PIC X.
    88  Test-Value-1 Value '1'.
    88  Test-Value-2 Value '1', '2'.
    88  Test-Value-3 Value '1' thru '8'.

    88  Test-Value-4 Value '1' thru '4', '6', '7'.
```

In the example above, the last 88 level definition is invalid.

A level 88 entry must be preceded by one of the following: Another level 88 entry, where there are several consecutive condition-names pertaining to an elementary item, or, an elementary item.

Every condition-name pertains to an elementary item in such a way that the condition-name be qualified by the name of the elementary item and the elementary item's qualifiers. A condition-name is used in the Procedure Division in place of a relational condition.

A condition-name may pertain to an elementary item (a conditional variable) requiring subscripts. In this case, the condition-name, when written in the Procedure Division, must be subscripted according to the same requirements as the associated elementary item.

The type of literal in a condition-name entry must be consistent with the data type of the conditional variable. In the following example, PAYROLL-PERIOD is the conditional variable. The picture associated with it limits the value of the 88 condition-name to one digit.

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

Using the above description, one may write the procedural condition-name test:

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

An equivalent statement is:

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

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

9. Level number 66 is assigned to identify RENAMES entries. They can be used only with Format two of the data description entry.

Any number of RENAMES entries may be written for a logical record. They must all immediately follow the last entry of that record.

Data-name-1 cannot be used as a qualifier. Neither data-name-2 nor data-name-3 may have an OCCURS clause nor be subordinate to an item with an OCCURS clause. Data-name-2 and data-name-3 must be names of elementary items or groups of elementary items in the same logical record and cannot be the same data-name.

The beginning of the area described by data-name-3 must be to the right of the area described by data-name-2.

A level 66 entry cannot RENAME another level 66 entry or a 77, 88 or 01 level entry.

DATA-NAME/FILLER

▶ Function

A data-name specifies the name of the data being described, FILLER specifies an elementary item of the logical record which cannot be referred to explicitly.

Format

```
{ data-name }
{ FILLER    }
```

▶ Syntax rule

In the File, Working-Storage, and Linkage Sections of the Data Division, a **data-name** or the keyword **FILLER** must be the first word following the level-number in each data description entry.

▶ General rules

1. FILLER can only be used to name an elementary item in a record. Under no circumstances can a FILLER item be referred to explicitly. However, FILLER can be used as a conditional variable because such use does not require explicit reference to the FILLER item, but rather to its value.
2. A VALUE clause can be used with a FILLER item.

REDEFINES

▶ Function

The REDEFINES clause allows the same computer storage area to be described by different data description entries.

Format

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

Note

Level-number, data-name-1 and the semicolon are not part of the REDEFINES clause, but are included to show the context.

Syntax rules

1. The **REDEFINES** clause is optional; when specified, it must immediately follow **data-name-1**.
2. **Level-numbers** of data-name-1 and **data-name-2** must be identical, but must not be 66 or 88.
3. This clause must not be used in level-number 01 entries in the File Section.
4. The data description entry for data-name-1 may contain a REDEFINES clause.
5. The data description entry for data-name-2 may not contain an OCCURS clause, nor may data-name-1 be subordinate to an entry which contains an OCCURS clause.
6. Data-name-2 can be qualified, but not subscripted.

General rules

1. Redefinition starts at data-name-2 and ends when a level-number less than or equal to that of data-name-2 is encountered. In the followning example, redefinition of the data-name-2 area by data-name-1 ends when data-name-3 is encountered:

```
05  data-name-2 PICTURE A(3).
05  data-name-1 REDEFINES data-name-2.
    10  ITEM-A PICTURE A.
    10  ITEM-B PICTURE AA.
05  data-name-3 PICTURE X.
```

2. The entries giving the new description of the area must not contain VALUE clauses except in condition-name entries.
3. Redefinition to a depth greater than one level is permitted (see Syntax Rule 4, above). Thus, the nested REDFINES outlined below is valid:

```
01  FIELD-A  PIC X(10).
01  FIELD-B  REDEFINES FIELD-A.
    05  FIELD-C  PIC X(5)
    05  FIELD-D  REDEFINES FIELD-C.
        10  FIELD-E1  PIC X(3).
        10  FIELD-E2  PIC X(2).
    05  FIELD-F  PIC X(5).
```

Note

The REDEFINES clause specifies the redefinition of a storage area, not of the data items contained therein.

RENAMES

Function

The RENAMES clause permits alternative possibly overlapping groups of elementary items.

Format

66 data-name-1 RENAMES data-name-2 { **THROUGH** / **THRU** } **data-name-3**

Note

Level-number 66 and data-name-1 are not part of the RENAMES clause, but are included to show the context.

▶ Syntax rules

1. Any number of **RENAMES** entries may be written for a logical record. They must all immediately follow the last entry of that record.
2. **Data-name-1** cannot be used as a qualifier but can be qualified to the 01 or FD entries. Neither data-name-2 nor data-name-3 may have an OCCURS clause nor be subordinate to an entry that has an OCCURS clause in its data description entry.
3. **Data-name-2** and **data-name-3** must be the names of elementary items or groups of elementary items in the same record and cannot have the same data-name.
4. A level 66 entry cannot rename another level 66 entry or a 77, 88 or 01 level entry.
5. The beginning of the area described by data-name-3 must be to the right of the area described by data-name-2.

▶ Example

```
01  MASTER.
    05  EMP-REC.
        10  EMP-NO          PIC 9(4).
        10  EMP-NAME.
            15  LASTT       PIC X(14).
            15  FIRSTT      PIC X(11).
        10  DEPT-CODE       PIC 99.
        10  TEL-EXT         PIC 9(4).
    05  WORKED-PER-WEEK     PIC 9(4).
66  NAME-TAG RENAMES EMP-NAME THRU DEPT-CODE.
```

OCCURS

▶ Functions

The OCCURS clause permits the definition of related sets of repeated data, such as tables, arrays, lists, supplying required information for the application of subscripts or indexes.

Format

```
OCCURS integer-1 TIMES
[ { ASCENDING  } ] KEY IS data-name-1 [, data-name-2] ... ]
  { DESCENDING }
[INDEXED BY index-name-1 [, index-name-2] ... ]
```

Syntax rules

1. **Integer-1** must be greater than one and less than 32,767.
2. The **OCCURS** clause must not be used in a data description entry having a 01, 66, 77, or an 88 level-number.
3. If the data-name applies to a group item, then all data-names belonging to the group must be subscripted or indexed whenever they are used.

General rules

1. When the OCCURS clause is used, the data-name which is the defining name of the entry must be subscripted whenever it appears in the Procedure Division. If the INDEXED BY phrase is specified, then the **data-name** must also be indexed. However, if the data-name is referred to in a SEARCH Statement, it must not be subscripted or indexed.
2. The OCCURS clause specification causes a fixed length table to be generated. Its length is equal to the value of integer-1 times the size of each element. The size of the table illustrated below is 10 × 10 (100):

```
01  FIRST-TABLE.
    05  ELEMENT PIC X(10) OCCURS 10 TIMES.
```

See Section 10, TABLE HANDLING, for further detailed discussion of the OCCURS clause.

PICTURE

Function

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

Format

```
[ { PICTURE } IS picture-string] (or character-string)
  { PIC     }
```

Syntax rules

1. A **PICTURE** clause can be specified only at the elementary item level.
2. A **picture-string** consists of certain allowable combinations of characters in the COBOL character set used as symbols. The allowable combinations determine the category of the elementary item.
3. The maximum number of character positions allowed in a picture-string is 30. For example, PIC X(35) and PIC X(3) consist of 5 and 4 PICTURE characters, respectively.
4. The PICTURE clause must be specified for every elementary item except binary items.
5. PIC is a valid abbreviation for PICTURE.
6. The asterisk when used as the zero suppression symbol and the clause BLANK WHEN ZERO may not appear in the same entry.

General rules

1. **Data:** Five categories of data can described with a PICTURE clause: Alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited.
   - **Alphabetic:** Picture-string can only contain the characters A and B; and, item contents must be any combination of the letters of the English alphabet and the COBOL space character.
   - **Numeric:** Picture-string can only contain the symbols 9, P, S, and V. The number of digit positions which may be represented by picture-string is from 1 to 18; and item contents must be a combination of the digits 0 through 9. These may be signed, or not. If signed, the item may be positive or negative.
   - **Alphanumeric:** Picture-string is a combination of data description characters X, A, or 9, and the item is treated as if the string contained all X's. Alphanumeric picture-strings may not employ all 9's or all A's; and, item contents may be any character from the computer's ASCII character set.
   - **Alphanumeric edited:** Picture-string is restricted to certain combinations and the following symbols: A, X, 9, B, 0, /: and, item contents are any character from the computer's ASCII character set.
   - **Numeric Edited:** The picture-string is a certain combination of the editing symbols: Z . CR DB , $ + * B 0 = – / 9 V P; and, the picutre-string must contain at least one of the editing symbols in conjunction with numeric symbols; and, item contents must be one of the digits.
2. **Size:** The size of an elementary item (the number of character positions occupied by the item in standard data format) is determined by the number of allowable symbols which represent character positions.

   An integer, enclosed in parentheses, following the symbols A, X 9 P X * B / 0 + – or the currency symbol, indicates the number of consecutive occurrences of that symbol. The following symbols can appear only once in a given PICTURE: S V . CR DB.
3. **Decimal-Point Clause:** When DECIMAL-POINT IS COMMA is specified, the explanations for period and comma are understood to apply to comma and periods, respectively.

4. **Symbols:** Symbols used in a picture-string to define an elementary item have the following functions:

A Each A represents a character position which contains only a letter of the alphabet, or a space.

B Each B represents a character position into which the space character will be inserted.

P Each P indicates an assumed decimal scaling position. It specifies the location of an assumed decimal point when the point is not within the number that appears in the data item. The P is not counted in the size of the data item, but is counted in determining the maximum number of digit positions (18) in numeric edited items or numeric items.

The scaling position character P may appear only to the left or right of the other characters in the string as a continuous string of P's within a PICTURE description. The sign character S and the assumed decimal point V are the only characters which may appear to the left of a leftmost string of P's. Since the scaling position character P implies an assumed decimal point (to the left of the P's if the P's are leftmost PICTURE characters, and to the right of the P's if the P's are the rightmost PICTURE characters), the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.

Example: If a field in memory contains the digits 37, and the picture-string for the field is PPP99, the field has the implied value of .00037. The same field, with a picture-string 99ppp has an implied value of 37000. In both instances, only digits 37 are actually stored in memory.

S The picture-string symbol S indicates the presence of a sign in a data item, but implies nothing about the actual format or location of the sign in storage.

The symbol S is not counted in determining the size of the elementary item, unless the entry is subject to a SIGN clause. (See SIGN.)

When used, the S symbol must be written as the leftmost character in picture-string.

V The 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 information concerning the alignment of items involved in computations. Storage is never reserved for the character V. Only one V, if any, is permitted in any single picture.

X Each X represents a character position which contains any allowable character from the computer's character set.

Z Each character Z is a replacement character which represents a digit position. Leading data item zeros are suppressed and replaced by blanks if corresponding picture string positions are defined by Z. Zero suppression terminates upon encountering the decimal point (.), or a non zero digit. Each Z is counted in the size of the item.

9 Each 9 in a picture-string represents a character position which contains a numeral and is counted in the size of the item.

/ Each stroke, or virgule (/), in the picture-string represents a character position into which the stroke character will be inserted. (/) is counted in the size of the item.

, The comma character (,) 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.

. A period character (.) in a picture-string is an editing symbol representing the decimal point for alignment purposes. The character also serves to indicate the position for decimal point insertion.

Numeric character positions to the right of an actual decimal point in a PICTURE must consist of characters of one type. The period character (.) is counted in the size of the item.

For a given program, the functions of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. In this exchange, the rules for the period apply to the comma and the rules for the comma apply to the period whenever they appear in a PICTURE clause.

The decimal insertion character (.) must not be the last character in the picture-string.

+
–
CR
DB These symbols are used as editing sign control symbols and represent the character position into which the editing sign control symbol is placed. The symbols are mutually exclusive in any one picture-string, and each character used in the symbol is counted in determining the size of the data item, i.e., CR and DB = 2 character positions each; + and – = 1 character position each.

* Each * (asterisk) in a picture-string is a replacement character. Leading data item zeros are suppressed and replaced by *. Each * is counted in the size of the item.

5. **Editing:**

- The PICTURE clause provides two basic methods for editing: character insertion and character suppression/replacement. The type of editing which may be performed upon an item is dependent upon the category to which the item belongs. The table below specifies which type of editing may be performed upon a given category:

Table 7-2. Categories of Data and Editing.

Category Of Data	Type Of Editing
Alphabetic	Simple insertion 'B' only
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple insertion 0, B and /
Numeric Edited	All, subject to rules for Fixed insertion editing

- Insertion Editing includes the following types: Simple insertion, special insertion, fixed insertion, and floating insertion.
- **Simple insertion editing:** utilizes B 0 , / as insertion characters. The insertion characters are counted in the size of the item and represent the position in the item into which the character will be inserted.
- **Special insertion editing:** refers to decimal point insertion (.) and resulting receiving item alignment. The insertion character used for the actual decimal point is counted in the size of the item. The use of the assumed decimal point - represented by the symbol V, and the use of an actual decimal point - represented by the insertion character, is disallowed in the same picture-string; the two are mutually exclusive. The result of special insertion editing is that the insertion character is placed in an item in the same position in which it appears in the picture-string.
- **Fixed insertion editing:** employs the currency sign and editing sign control symbols as insertion characters. The editing sign control symbols are: + – CR DB.
- Only one currency symbol, and only one of the editing sign control symbols, can be used in a given picture-string. When the symbols CR or DB are used, they represent two character positions in determining the size of the item. They must represent the rightmost character positions to be counted in the size of the item. The symbol + or –, when used, must be either the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of an item, except that it can be preceded by either a + or – symbol. Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the picture-string. Editing sign control symbols produce the following results depending upon the value of the data item:

Table 7-3. Results of Sign Control Symbols in Editing

EDITING SYMBOL IN PICTURE-STRING	RESULT	
	DATA ITEM POSITIVE OR ZERO	DATA ITEM NEGATIVE
+	+	–
–	space	–
CR	2 spaces	CR
DB	2 spaces	DB

- **Floating insertion editing:** utilizes the currency symbol and editing sign control symbols + or – as floating insertion characters. These are mutually exclusive in a given picture-string.
- A floating picture-string is defined as a leading, continuous series of either $ + or –, or a string composed of one such character interrupted by one or more insertion commas and/or decimal point.

- For example:

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

- Floating insertion editing is indicated in a picture-string by using a string of at least two of the floating insertion characters. The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.
- The second floating character from the left represents the leftmost limit of the numeric data which can be stored in the data item. Non-zero numeric data may replace all the characters at or to the right of this limit.
- In a picture-string, there are only two ways of representing floating insertion editing. One way is to represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character. The other way is to represent all of the numeric character positions in the picture-string by the insertion character.
- If the insertion characters are only to the left of the decimal point in the picture-string, the result is that a single floating insertion character will be placed into the character position immediately preceding the first non-zero digit in the data item. If all data item digits to the left of the decimal are zero, the floating insertion character will be placed into the character position immediately preceding the decimal point. The character positions preceding the insertion character are replaced with spaces.
- If all numeric character positions in the picture-string are represented by the insertion character, the result depends upon the value of the data. If the value is zero, the entire data item will contain spaces.
- If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point.
- To avoid truncation, the minimum size of the picture-string for the receiving data item must be the number of characters in the sending data item, plus the number of non-floating insertion characters being edited into the receiving data item, plus one for the floating insertion character. That is, a floating string containing n + 1 occurrences of $ or + or – defines n digit positions.
- In the following examples, b represents a blank in the developed items.

- Examples:

Picture-string	Numeric Value	Developed Item
$$$999	14	bb$014
--,---,999	-456	bbbbbb-4'
$$$$$$	14	bbb$14

- A floating string need not constitute the entire PICTURE of a numeric edited item, as shown in the preceding examples. However, the characters to the right of a decimal point and up to the end of a PICTURE, excluding the fixed insertion characters +, –, CR, DB (if present), are subject to the following restrictions:

  Only one type of digit position character may appear. That is, Z * 9 and floating-string digit position characters $ + – are mutually exclusive.

  If any of the numeric character positions to the right of a decimal point is represented by + or – or $ or Z, then all the numeric character positionsin the PICTURE must be represented by the same character.

  The PICTURE character 9 can never appear to the left of a floating string, or replacement character. In fact, nothing can precede a floating string.

  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.
- Suppression/replacement editing includes two types: zero suppression and replacement with spaces, and zero suppression and replacement with asterisks.
- Floating insertion editing and editing by zero suppression/replacement are mutually exclusive in a PICTURE clause.
- The suppression of leading zeros in numeric character positions is indicated by the use of the alphabetic character Z, or the character * (asterisk) as suppresion symbols in a picture-string. Each suppression symbol is counted in determining the size of the item. If Z is used, the replacement character will be the space. If the asterisk is used, the replacement character will be *
- Zero suppression and replacement are indicated in a picture-string by one or more of the allowable symbols (Z or *), representing leading numeric character positions. These, in turn, are to be replaced when the associated character position in the data contains a zero. Any simple insertion character embedded in the string of symbols, or to the immediate right of this string, is part of the string.
- The two ways of representing zero suppression in a character-string are:

  Represent any or all leading numeric character positions to the left of the decimal point by suppression symbols; and, represent all numeric character positions in the picture-string by suppression symbols.

- If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates either at the first non-zero digit in the data represented by the suppression symbol string, or at the decimal point, whichever is first.
- If all numeric character positions in the picture-string are represented by suppression symbols, and the value of the data is not zero, the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero, the entire data item will be spaces if the symbol is Z, or all asterisks (except for the actual decimal point) if the symbol is *.
- A picture-string must consist of at least one of the characters Z A * X 9, or at least two consecutive appearances of the characters + – $.
- The examples below illustrate the use of the PICTURE clause. In each example, a movement of data is implied, as indicated by the column headings.

Source Area		Receiving Area	
PICTURE	Data Value	PICTURE	Edited data
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00123	$$$,$$9.99	$123.00
9(5)	00000	$$$,$$9.99	$0.00
9(3)V99	00000	$$$.$$	
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
S9(5)	-12345	+ZZ,ZZZ.99	-12,345.00
S9(5)	12345	-ZZ,ZZZ.99	12,345.00
S9(5)	-12345	ZZ,ZZ9.99-	12345.00-
S9(5)	12345	ZZ,ZZ9.99+	12,345.00+
S9(5)	00000	$ZZ,ZZZ.ZZ	
9(5)	00123	++++++.99	+123.00
9(5)	00123	------.99	123.00
9(5)	00000	$**,***.**	**********
9(5)	00000	$**,***.99	$******.00
S9(5)	12345	*******.99CR	**12345.00
S999V99	02345	ZZZVZZ	2345
S999V99	00004	ZZZVZZ	04
S9(5)	-12345	*******.99CR	**12345.00CR
S9(5)	12345	$$$$$$.99CR	$12345.00

Figure 13-1. Examples of PICTURE Clauses

USAGE

Function

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

Format

```
[USAGE IS { DISPLAY                 }  ]
          { COMPUTATIONAL           }
          { COMP                    }
          { INDEX                   }
          { COMPUTATIONAL-3         }
          { COMP-3                  }
```

Syntax rules

1. **COMP** is a valid abbreviation for **COMPUTATIONAL**.
2. **COMP-3** is a valid abbreviation for **COMPUTATIONAL-3**.
3. The PICTURE clause cannot be used if **USAGE** is specified as **COMPUTATIONAL** or **INDEX**.

General rules

1. The USAGE clause can be written at any level. If the USAGE clause is written at a group level, it applies to each elementary item in the group. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group item to which it belongs.
2. A COMPUTATIONAL item can represent a value to be used in computations and must be numeric. When a group item is described as COMPUTATIONAL, only the elementary items in that group are COMPUTATIONAL; the group item itself cannot be used in computations.
3. **DISPLAY** is the system default if the USAGE clause is not specified.
4. If USAGE is specified as COMPUTATIONAL for an item, and a PICTURE clause is included for the same item, the computer will ignore the USAGE clause.

Note

See Data Representation in section 4 for additional information.

SIGN

Function

The SIGN clause specifies the position and the mode of representation of the operational sign when it is necessary to describe these properties explicitly.

Format

```
[SIGN IS { LEADING  }  [SEPARATE CHARACTER]]
         { TRAILING }
```

Syntax rules

1. The **SIGN** clause may be specified only for a numeric data description entry whose PICTURE contains the character S, or for a group item containing at least one such numeric data description entry. If an S is not present in the data item picture-string., the item is considered unsigned (capable of storing only absolute values), and the SIGN clause is prohibited.

2. Numeric data description entries to which the SIGN clause applies must be described by USAGE IS DISPLAY.
3. Only one SIGN clause can apply to any given numeric data description entry.

General rules

1. When S appears in a picture-string, but no SIGN clause is included in an item's description, the system default is **SIGN IS TRAILING**.
2. If the optional **SEPARATE CHARACTER** phrase is not present, then:
   - The operational sign is presumed associated with the leading (or, respectively, trailing) digit position of the elementary numeric data item.
   - The character S in picture-string is not counted in determining item size.
3. If the SEPARATE CHARACTER phrase is present, then:
   - The operational sign will be presumed the leading (or respectively, trailing) character position of the elementary numeric data item; this character position is not a digit position.
   - The letter S in a picture-string is counted in determining the size of the item (in terms of standard data format characters).
   - The operational signs for positive and negative are the standard data format characters + and –, respectively.
4. Every numeric data description entry whose PICTURE contains the character S is a signed numeric data description entry. If a SIGN clause applies to such an entry and conversion is necessary for purposes of computation or comparisons, conversion takes place automatically.
5. Table 7-4 depicts sign representations for the various SIGN clause options.

Table 7-4. Sign Representation

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

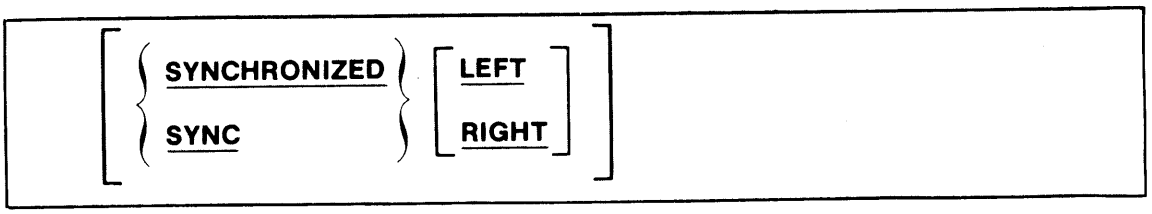
6. At a group level, an attribute of SEPARATE will cause a group type error at compile-time. Such attributes must be specified at the elementary level.

SYNCHRONIZED

Function

The SYNCHRONIZED clause specifies the alignment of an elementary item on its natural addressing boundaries in the computer memory.

Format



Syntax rules

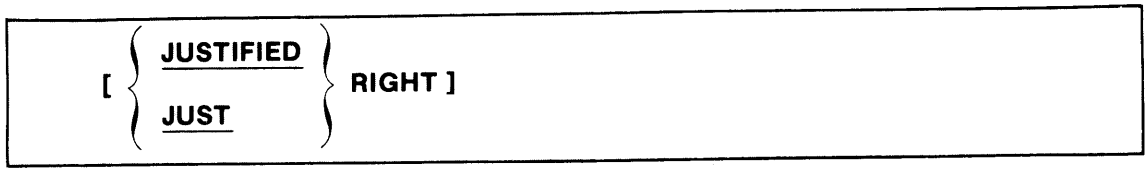
1. **SYNC** is a valid abbreviation for **SYNCHRONIZED**.
2. In this compiler, the SYNCHRONIZED specification is treated as commentary.

JUSTIFIED

Function

The JUSTIFIED clause specifies nonstandard positioning of data within a receiving data item.

Format



Syntax rules

1. This clause can be specified only at the elementary level.
2. **JUST** is a valid abbreviation of **JUSTIFIED**.
3. The JUSTIFIED clause cannot be used for data items described as numeric, or for those for which editing is specified.

General rules

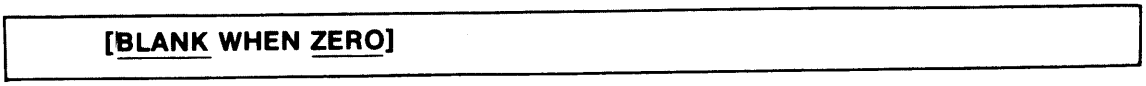
1. When the JUSTIFIED clause option is taken, values are stored in right-to-left fashion. The clause is effective in connection with a MOVE statement. In a MOVE operation, if the sending field is shorter then the receiving field, space filling occurs in the left-most-positions. If the sending field is longer than the receiving field, the left-most characters are truncated.
2. When the JUSTIFIED clause is omitted, Standard Alignment Rules apply.

BLANK WHEN ZERO

Function

The BLANK WHEN ZERO clause permits the blanking of an item when its value is zero.

Format



Syntax rule

The **BLANK WHEN ZERO** clause can be used only for an elementary numeric or numeric edited item.

General rules

1. When used, the BLANK WHEN ZERO clause specifies that the data item will be set to blanks when the value is all zeros. Leading zeros are not supressed by this clause.
2. If the clause is specified for a numeric item, the category of the item is interpreted as numeric edited.

VALUE	DESCRIPTION OF OUT-COST		RESULT
0012.34	9999.99	BLANK WHEN ZERO	0012.34
0123.45	$9999.99	BLANK WHEN ZERO	$0123.45
01.2345	$9999.99	BLANK WHEN ZERO	$0001.23
0000.04	$$$$$.99	BLANK WHEN ZERO	$.04
0000.00	$$$$$.99	BLANK WHEN ZERO	
0000.00	$$$$$.99		$.00
0012.34	****.99	BLANK WHEN ZERO	**12.34
0012.34	****.99		**12.34
0000.00	****.99	BLANK WHEN ZERO	****.**
0000.00	****.99		****.00
0000.00	ZZZZVZZ	BLANK WHEN ZERO	
0000.04	ZZZZVZZ	BLANK WHEN ZERO	4
0000.00	ZZZZ.ZZ	BLANK WHEN ZERO	
0000.04	ZZZZ.ZZ	BLANK WHEN ZERO	.04
0000.00	ZZZZ.99	BLANK WHEN ZERO	
0000.00	ZZZZ.99		.00

Figure 13-2. Examples: BLANK WHEN ZERO

VALUE

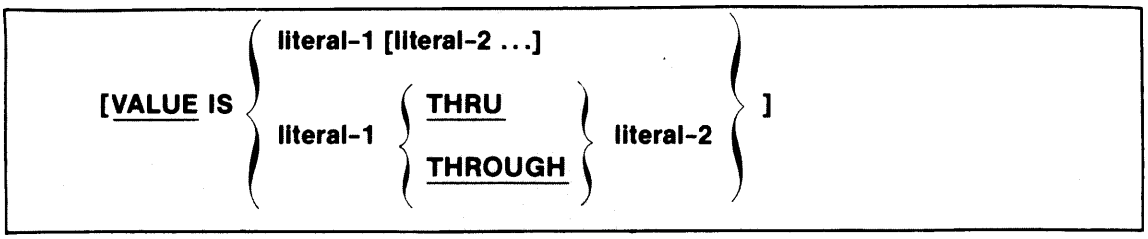
Function

The VALUE clause defines the value of constants, the initial values of WORKING-STORAGE items, and the values associated with a condition-name.

Format one

```
[VALUE IS literal]
```

Format two



Syntax rules

1. The words **THROUGH** and **THRU** are equivalent.
2. The **VALUE** clause is not permitted in a data description entry specifying an OCCURS or REDEFINES clause.
3. Numeric **literals** in a VALUE clause must have a value which is within range of values indicated by the PICTURE clause, and must not have a value which would require truncation of nonzero digits. Non-numeric literals in a VALUE clause must not exceed the size indicated by the PICTURE clause.
4. The type of literal written in a VALUE clause depends on the type of data item, as specified in the data item formats earlier in this text. For edited items, values must be specified as non-numeric literals. A type conflict, producing a compile time error, will arise if a figurative constant or literal is not compatible with the PICTURE. For example, PICTURE X VALUE ZERO will produce a type conflict error, since ZERO is a numeric figurative constant, but PICTURE X specifies an alphanumeric item.
5. In a data item with a VALUE clause, the size of the data item cannot exceed 32,767 characters.
6. A VALUE clause may not occur in the FILE SECTION of the Data Division except in level 88 condition-name entries.

General rules

1. The positioning of the literal within a data area is the same as would result from specifying a MOVE of the literal to a data area.
2. The VALUE clause may be specified at the group level in the form of a correctly sized, non-numeric literal, or figurative constant.
3. When an initial value is not specified, no assumption should be made regarding the initial contents of an item in Working-Storage.
4. A figurative constant may be specified in both Format one and Format two instead of a literal.
5. Format one is required to define an initial value for a data item or a constant.

6. Format two is reqired for condition-name entries. The VALUE clause and the level-number 88 condition-name itself are the only two items permitted in the entry. The characteristics of a condition-name are implicitly those of its conditional variable. Wherever the TRHU phrase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.
7. Rules governing the VALUE clause differ in the respective sections of the Data Division:
   - In the File and Linkage Sections, the clause can be used only in condition-name entries.
   - In the Working-Storage Section, the clause must be used in condition-name entries; it can also be used to specify the initial value of any other data item, with the result that the item assumes the specified value at the start of the object program.
8. Level 88 condition-name entries specify a value, list of values, or a range of values which an elementary item may assume.
   - A level 88 entry must be preceded either by another level 88 entry (in the case of several consecutive condition-names pertaining to an elementary item) or by an elementary item.
   - Every condition-name pertains to an elementary item in such a way that the condition-name may be qualified by the name of the elementary item and the elementary item's qualifiers.
   - A condition-name is used in the Procedure Division in place of a simple relational condition.
   - A condition-name may pertain to an elementary item (a conditional variable) requiring subscripts. In such a case, the conditional-name, when written in the Procedure Division, must be subscripted according to the same requirements as the associated elementary item.
   - 88 Level specifications can contain individual values, series of individual values, a range of values, or a series of ranges of values, but not a combination of ranges and individual values. (See also LEVEL-NUMBER.)

WORKING-STORAGE SECTION

Function

The WORKING-STORAGE SECTION of the Data Division describes noncontiguous data (level 77), and records which are not part of external files, but are developed and processed internally. This section also contains data assigned fixed or constant values.

Format

```
[ WORKING-STORAGE SECTION. ]
[   [ 77-level-description-entry ]  ... ]
[   [ record-description-entry   ]      ]
```

Syntax rules

1. The Working-Storage Section is optional. If included, it must begin with the words 1WORKING-STORAGE SECTION, followed by a period and a space.
2. Noncontiguous item names and record names in the Working-Storage Section must be unique; they cannot be qualified. Subordinate data-names need not be unique if they can be made unique by qualifications.
3. The level-number 77 is applied to noncontiguous elementary data items, each defined in a separate data description entry which must contain the level-number 77, a data-name, and a PICTURE clause or USAGE IS INDEX clause, with other optional data description clauses as necessary.
4. Data items in the Working-Storage Section with a definite hierarchic relationship to one another must be grouped into records according to the rules for formation of record descriptions. Any clause used in a record description in the File Section can be used in a record description in the Working-Storage Section (see RECORD DESCRIPTION).

General rules

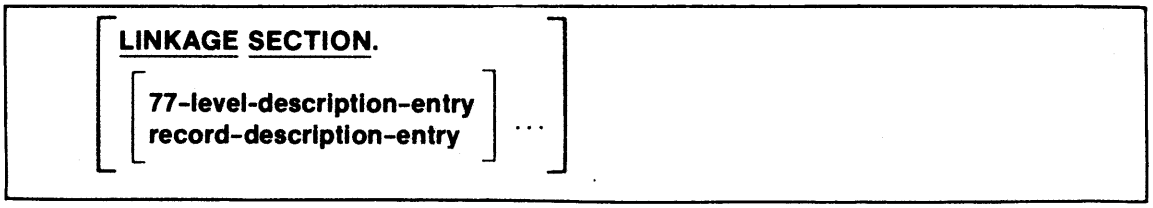
1. Working-Storage items described in this section include the following:
   - In the File and Linkage Sections, the clause can be used only in condition-name entries.
   - In the Working-Storage Section, the clause must be used in condition-name entries; it can also be used to specify the initial value of any other data item, with the result that the item assumes the specified value at the start of the object program.
2. Level 88 condition-name entries specify a value, list of values, or a range of values which an elementary item may assume.
   - A level 88 entry must be preceded either by another level 88 entry (in the case of several consecutive condition-names pertaining to an elementary item) or by an elementary item.
   - Every condition-name pertains to an elementary item in such a way that the condition-name may be qualified by the name of the elementary item and the elementary item's qualifiers.
   - A condition-name is used in the Procedure Division in place of a simple relational condition.
   - A condition-name may pertain to an elementary item (a conditional variable) requiring subscripts. In such a case, the conditional-name, when written in the Procedure Division, must be subscripted according to the same requirements as the associated elementary item.
   - 88 Level specifications can contain individual values, series of individual values, a range of values, or a series of ranges of values, but not a combination of ranges and individual values. (See also LEVEL-NUMBER.)

LINKAGE SECTION

Function

The Linkage Section describes data previously defined in a calling program which is available to a called program.

Format



Syntax rules

1. The Linkage Section is optional. If included, it must begin with the words **LINKAGE SECTION** followed by a period and a space.
2. Each Linkage Section record-name and noncontiguous item name must be unique within the called program; it cannot be qualified.
3. Level-number 77 refers to noncontiguous elementary data items, with no hierarchic relationship to one another, and therefore not grouped into records. Each level-number 77 data item is defined in a separate data description entry which must include the level-number 77, a data-name, and a PICTURE clauses may be included as necessary.
4. Data items in the Linkage Section, which have a definite hierarchic relationship to one another, must be grouped into records according to the rules for formation of Record Description.
5. The VALUE clause must not be specified in the Linkage Section except in level 88 condition-name entries.

General rules

1. The Linkage Section of the Data Division is meaningful if and only if the called program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.
2. The Linkage Section is used to describe data which is available through the calling program, but is to be referred to in both the calling program and the called program. No space is allocated in the program for data items referenced by data-names in the Linkage Section of that program. Procedure Division references to these data items are resolved at load time by equating the reference in the called program to the location used in the calling program.
3. Data items defined in the Linkage Section of the called program may be referenced within the Procedure Division of the called program only if they are specified as operands of the USING phrase of the Procedure Division header, or are subordinate to such operands, and the called program is under the control of a CALL statement which specifies a USING phrase.

Note

A Linkage Section example is presented in Section 9, INTER-PROGRAM COMMUNICATION.

Example

```
DATA DIVISION.
FILE SECTION.
FD  PRINT-FILE, LABEL RECORDS ARE OMITTED
    DATA RECORDS ARE PRINT-LINE, PRINT-LINE1.
Ø1  PRINT-LINE  PICTURE IS X(1ØØ).
Ø1  PRINT-LINE1.
    Ø5  USER-CARRIAGE-CONTROL PICTURE IS X.
    Ø5  PRINT-LINE-DETAIL PICTURE IS X(99).
FD  CARD-FILE , LABEL RECORDS ARE STANDARD
    DATA RECORDS ARE CARD-IMAGE , CARD-RECORD ,
    RECORD CONTAINS 8Ø CHARACTERS ,
    VALUE OF FILE-ID IS 'DATAIN'.
Ø1  CARD-IMAGE PICTURE IS X(8Ø).
Ø1  CARD-RECORD.
    Ø5  PHONE-IN PICTURE IS X(8).
    Ø5  DATA-IN  PICTURE IS X(64).
    Ø5  STATE-IN PICTURE IS XX.
    Ø5  D-O-B-IN PICTURE IS X(6).
FD  DIRECTORY-FILE, LABEL RECORDS ARE STANDARD,
    DATA RECORDS ARE DIRECTORY-RECORD-OUTPUT,
         DISPLAY-RECORD, DIRECTORY-RECORD-INPUT,
    RECORD CONTAINS 1ØØ CHARACTERS,
    VALUE OF FILE-ID IS 'INDXFILE'.
Ø1  DIRECTORY-RECORD-OUTPUT.
    Ø5  PHONE-NUMBER    PICTURE IS X(8).
    Ø5  NAME.
        1Ø  LAST-NAME   PICTURE IS X(14).
        1Ø  FILLER      PICTURE IS X.
        1Ø  FIRST-NAME  PICTURE IS X(13).
        1Ø  FILLER      PICTURE IS XXX.
    Ø5  ADDRESS  PICTURE IS X(25).
    Ø5  FILLER   PICTURE IS X.
    Ø5  CITY     PICTURE IS X(4).
    Ø5  FILLER   PICTURE IS X(3).
    Ø5  STATE    PICTURE IS XX.
    Ø5  BIRTH-DATE   PICTURE IS 9(6).
    Ø5  FILLER   PICTURE IS X(2Ø).
Ø1  DISPLAY-RECORD.
    Ø5  DISPLAY-DIR   PICTURE IS X(72).
    Ø5  FILLER        PICTURE IS X(28).
Ø1  DIRECTORY-RECORD-INPUT.
    Ø5  PHONE-IN PICTURE IS X(8).
    Ø5  DATA-IN  PICTURE IS X(64).
    Ø5  STATE-IN PICTURE IS XX.
    Ø5  D-O-B-IN PICTURE IS X(6).
    Ø5  FILLER   PICTURE IS X(2Ø).
```

```
WORKING-STORAGE SECTION.
77  AT-END-SWITCH  PICTURE IS 9     VALUE IS ZERO.
77  GO-TO-READ     PICTURE IS 9     VALUE IS ZERO.
77  GO-TO-NAME     PICTURE IS 9     VALUE IS ZERO.
77  CREATE-UPDATE  PICTURE IS X     VALUE IS SPACE.
77  FILE-STATUS    PICTURE IS XX    VALUE IS SPACE.
77  ACCEPT-TRANSACTION-TYPE PIC X   VALUE IS SPACE.
Ø1  PERFORM-COUNT1.
    Ø5  PERFORM-COUNT    PICTURE IS 999.
    Ø5  PER-CO  REDEFINES  PERFORM-COUNT
                  PICTURE IS X,  OCCURS 3 TIMES.
Ø1  WS-RECORD.
    Ø5  WS-LAST-NAME      PICTURE IS X(14).
    Ø5  FILLER            PICTURE IS X.
    Ø5  WS-FIRST-NAME     PICTURE IS X(13).
    Ø5  FILLER            PICTURE IS XXX.
    Ø5  WS-ADDRESS        PICTURE IS X(25).
    Ø5  FILLER            PICTURE IS X.
    Ø5  WS-CITY           PICTURE IS X(4).
    Ø5  FILLER            PICTURE IS XXX.
    Ø5  WS-PHONE-NUMBER   PICTURE IS X(8).
    Ø5  WS-STATE          PICTURE IS XX.
    Ø5  WS-BIRTH-DATE     PICTURE IS X(6).
Ø1  HEADER.
    Ø5  CARRIAGE-CTRL  PICTURE IS XX VALUE IS SPACE.
    Ø5  HEADER-Ø   PICTURE IS X(3)   VALUE IS 'PHONE'.
    Ø5  FILLER     PICTURE IS X      VALUE IS SPACE.
    Ø5  HEADER-1   PICTURE IS X(4)   VALUE IS 'NAME'.
    Ø5  FILLER     PICTURE IS X(27)  VALUE IS SPACE.
    Ø5  HEADER-2   PICTURE IS X(6)   VALUE IS 'STREET'.
    Ø5  FILLER     PICTURE IS X(2Ø)  VALUE IS SPACE.
    Ø5  HEADER-3   PICTURE IS X(4)   VALUE IS 'CITY'.
    Ø5  FILLER     PICTURE IS X      VALUE IS SPACE.
```

8

Procedure division

PROCEDURE DIVISION

▶ Function

The procedure division contains instructions specifying the data processing steps to be performed by the program. COBOL instructions are written as sentences which are combined to form paragraphs under paragraph names. These, in turn, are combined to form sections under section names.

Within COBOL sentences, verbs (commands) are employed to denote actions. Statements and sentences denote procedures.

Format

```
PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ...].

[DECLARATIVES.

{section-name SECTION. USE sentence.

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

END DECLARATIVES.]

{section-name SECTION.

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

▶ Syntax rules

1. The first entry in the Procedure Division must be the words **PROCEDURE DIVISION**.
2. The **USING** clause is specified only if:
   - The program being written is a CALLable subprogram which is to function under the control of a CALL statement.
   - The CALL statement in the calling program contains a USING clause.
3. Each of the **data-name** operands in the USING clause must be defined as a data item in the Linkage Section of the subprogram.
4. Within the subprogram, Linkage Section data items are processed according to their data descriptions as given in the subprogram.
5. Data-name level-numbers in the USING clause must be 01 or 77. See Section 9, INTER-PROGRAM COMMUNICATION for complete discussion.

6. Declarative sections are optional. When included, they must be grouped at the beginning of the Procedure Division, preceded by the key word **DECLARATIVES** and followed by the key words **END DECLARATIVES.** These entries must appear on separate lines.
7. A **SECTION** entry is optional. When included, it must consist of **section-name**, followed by the word SECTION and a period. Each section header must appear on a line by itself; each section-name must be unique.
8. A **paragraph** is a logical entry consisting of one or more sentences. A **paragraph-name** must precede the first sentence.
9. A **sentence** is a single statement or a series of statements terminated by a period and followed by a space.
10. A **statement** consists of a COBOL verb followed by appropriate operands (data-names or literals) and other words necessary for the completion of the statement. There are two types of statements, the Imperative and Conditional:
    - **Imperative Statements**: An imperative statement specifies an unconditional action to be taken by the object program. An imperative statement consists of a verb and its operands, excluding the IF conditional statement, the READ statement and any I/O statement which has an INVLAID KEY clause.
    - **Conditional Statements**: A conditional statement stipulates a condition which is tested to determine whether an alternate path of program flow is to be taken. The IF statement provides this capability. READ statements, and any I/O statement having an INVALID KEY clause are also considered to be conditional. When an arithmetic statement possesses a SIZE ERROR suffix, the statement is considered to be conditional rather than imperative.
11. Arithmetic statements may be imperative or conditional. The five arithmetic verbs are: ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE.

General rules

1. The sections under the DECLARATIVES header provide a method for including procedures which are invoked when a condition occurs which cannot normally be tested by the programmer. Each Declaratives Section comprises a section header, a USE compiler-directing sentence, and, optionally, one or more paragraphs.

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

   Since such procedures are executed only at the time an error in reading and writing occurs, they cannot appear in the regular sequence of procedural statements. Instead, they must appear in the DECLARATIVES section. Related procedures are preceded by a USE sentence.

   Within a USE procedure, there must be no reference to non-declarative procedures. Conversely, in the non-declarative portion, there must be no reference to procedure-names which appear in the declarative portion, except that PERFORM statements may refer to the procedures associated with a USE statement. For additional information, see USE statement.

2. After END DECLARATIVES is specified, no text can appear before the next section header.
3. The Procedure Division is usually, though not necessarily, written in sections, each with a section header followed optionally by one or more successive paragraphs.
4. Section-name and paragraph-name follow the general rules for WORD FORMATION.
5. Arithmetic statements in the Procedure Division are governed by the following rules:
    - All data-names used in arithmetic statements must be elementary numeric data items which are defined in the Data Division of the program, except when they are the operands of GIVING. The data item may be numeric edited. Index-names and index items are not permissible in these arithmetic statements.
    - Decimal point alignment is supplied automatically throughout the computations.
    - Intermediate result fields generated for the evaluation of arithmetic expressions assure the accuracy of the result field, except where high-order truncation is necessary.
    - The maximum size of each operand is 18 decimal digits. The composite of operands, which is a hypothetical data item resulting from the superimposition of specified operands in a statement aligned on their decimal points, must not contain more than 18 decimal digits.
    - When arithmetic is attempted with one or more nonnumeric operands, the program will execute, but results are invalid.
6. The three statement components which may appear in all arithmetic statements are: The GIVING option, the ROUNDED option, the SIZE ERROR option.
    - If the GIVING option is written, the value of the data-name which follows the word GIVING is made equal to the calculated result of the arithmetic operation. The data-name which follows GIVING is not used in the computation and may be a numeric edited item.
    - When the ROUNDED option is specified, if the most significant digit of the excess is greater than or equal to 5, the least significant digit of the resultant data-name has its value increased by 1. If the ROUNDED option is not taken, truncation will occur after decimal-point alignment if the result is greater than the size of the receiving data item.
    - Rounding of a computed negative result is performed by rounding the absolute value of the computed result and then making the final result negative.
    - The following chart illustrates the relationship between a calculated result and the value stored in an item which is to receive the calculated result, with and without rounding.

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

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

[ON SIZE ERROR imperative-statement ...]

- If, after decimal-point alignment and any low-order truncation, the value of a calculated result exceeds the largest value which the receiving field is capable of holding, a size error condition exists.
- If the SIZE ERROR option is present, and a size error condition arises, the value of the resultant data-name is unaltered and the series of imperative statements specified for the condition is executed.
- If the SIZE ERROR option has not been specified and a size error condition arises, no assumption should be made about the final result.
- An arithmetic statement, if written with a SIZE ERROR option, is not an imperative statement. Rather, it is a conditional statement since it is data-dependent and is prohibited in contexts where only imperative statements are allowed.
- An example of a conditional arithmetic statement is:

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

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

PROCEDURE STATEMENTS

COBOL statements (verbs) are described on the following pages in alphabetic sequence. For a brief reference, see the Prime COBOL Verb Index, Table C-1, in Appendix C.

ACCEPT

▶ Function

The ACCEPT statement causes low-volume data to be made available to the specified data item.

Format one

```
ACCEPT data-name [FROM mnemonic-name]
```

Format two

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

Syntax rule

The mnemonic-name in Format one must be specified also in the SPECIAL-NAMES paragraph of the Environment Division, and must be associated with the console (terminal).

General rules

1. The ACCEPT statement causes transfer of data from the hardware device. The transferred data replaces the contents of the field specified by **data-name**.
2. One line is read, and as many characters as necessary (depending on the size of the named data field) are moved, without change, to the indicated field. The maximum number of characters which can be read is 72.
3. Omission of **FROM** mnemonic-name implies that input is from the terminal.
4. When FROM **mnemonic-name** is specified, input is keyed-in at the terminal by the operator; mnemonic-name must be assigned to CONSOLE in the SPECIAL-NAMES paragraph.

   When input is to be accepted from the terminal, execution consists of the following steps:

   - Execution is suspended.
   - When the operator enters a response, the program stores the acquired data in the field designated by data-name, and normal execution proceeds.
   - The data size is controlled by the size specified for data-name.
   - For unequal sizes of data-name and terminal input the result is treated as an alphanumeric to alphanumeric move with space fill on the right or right truncation.

5. The Format two ACCEPT statement causes the requested information to be transferred to the data item specified by data-name according to the rules of the MOVE statement. DATE, DAY, and TIME are conceptual data items and are therefore not described in the COBOL program.
6. **DATE** has the following data elements: Year, month, and day of the month, in that sequence, from high to low order (left to right). July 1, 1974 is expressed as 740701. DATE, when accessed by a COBOL program, is treated as though described in the COBOL program as an unsigned elementary numeric integer data item six digits long.
7. **DAY** has the following data elements: Year, and day of year, in that sequence, from high to low order (left to right). July 1, 1974 would be expressed as 74183. DAY, when accessed by a COBOL program, is treated as though described in a COBOL program as an unsigned elementary numeric integer data item five digits long.

8. **TIME** has the following data elements: Hours, minutes, seconds and hundreds of a second. TIME is based on time elapsed after midnight on a 24-hour basis; thus 2:41 p.m., or 1441 hours, is expressed as 14410000. TIME, when accessed by a COBOL program, is treated as though described in a COBOL program as an unsigned elementary numeric integer data item eight digits long. The minimum value of TIME is 00000000; maximum value is 23595999.

ADD

▶ Function

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

Format one

```
ADD { data-name-1 } [ , data-name-2 ] ... TO data-name-3 [ROUNDED]
    { literal-1   } [ , literal-2   ]
    [; ON SIZE ERROR imperative-statement]
```

Format two

```
ADD { data-name-1 } { , data-name-2 } [ , data-name-3 ] ...
    { literal-1   } { , literal-2   } [ , literal-3   ]
GIVING data-name-4 [ROUNDED] [; ON SIZE ERROR imperative-statement]
```

Format three

```
ADD { CORRESPONDING } identifier-1 TO identifier-2
    { CORR          }
    [ROUNDED] [; ON SIZE ERROR imperative-statement]
```

▶ Syntax rules

1. In Formats one and two, each **data-name** must refer to an elementary numeric item, except that in Format two each item following GIVING can be either an elementary numeric item or an elementary numeric edited.
2. Each **literal** must be a numeric literal.
3. The maximum size of each operand is 18 digits. If all operands, excluding those following the word GIVING, were to be superimposed upon each other, aligned by their implied decimal points, their composite could not exceed 18 decimal digits in length.
4. In Format three, elementary items under group name **identifier-1** are added to and stored into the corresponding elementary items under group name **identifier-2**.

General rules

1. In Format one, the values of the operands preceding the word **TO** are added, the sum is added to the current value of **data-name-3** and the result is stored immediately in data-name-3.
2. In Format two, the values of the operands preceding the word **GIVING** are added, and the sum is stored as the new value of **data-name-4** following GIVING.
3. In Format three, data items in **identifier-1** are added to and stored in corresponding data items in **identifier-2**.
4. See the rules for arithmetic statements under PROCEDURE DIVISION, General Rules. The **ROUNDED** and **ON SIZE ERROR** options may be used when truncation of the results could occur.
5. The rules for signs are those presented in FUNDAMENTAL CONCEPTS OF COBOL, Algebraic Signs.

Examples

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

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

ALTER

Function

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

Format

ALTER paragraph-name-1 TO [PROCEED TO] paragraph-name-2

Syntax rules

1. **Paragraph-name-1** contains a single GO TO sentence without the DEPENDING phrase.
2. **Paragraph-name-2** is the name of another paragraph or section in the Procedure Division.

General rule

Execution of the ALTER statement modifies the GO TO statement in paragraph-name-1 so that subsequent executions of the modified GO TO statements cause transfer of control to paragraph-name-2.

Example

```
GATE.
    GO TO M-F-OPEN.
M-F-OPEN.
    OPEN INPUT MASTER-FILE.
    ALTER GATE TO PROCEED TO NORMAL.
```

```
NORMAL.
    READ MASTER-FILE, AT END GO TO EOF-MASTER.
```

Examination of the above code reveals the technique for providing for a one-time initializing program step.

Note

ALTER is fully supported in PRIME COBOL. Its use, however, is inconsistent with structured programming techniques. The reader should be aware that the ALTER statement presents difficulties in the degugging process.

CALL

▶ Function

The CALL statement allows one program to communicate with one or more other programs. It causes control to be transferred from one loaded program to another within a run unit, with both programs having access to data items referred to in the CALL statement.

Format

CALL literal-1 [USING data-name-1 [, data-name-2] ...]

▶ Syntax rule

The **literal-1** must be a non-numeric literal which is a subprogram name defined as the PROGRAM-ID of a separately compiled program; it must be enclosed in quote marks.

Note

The relationship of literal-1 and PROGRAM-ID is illustrated in the example at the end of Section 9.
When calling subroutines, literal-1 is the subroutine name (for example, 'SUBSRT') and data-names in the USING list are the arguments passed and returned. For available subroutines and calling sequences, refer to the PRIMOS Subroutines Reference Guide.

▶ General rule

Data-name(s) in the USING list are made available to the called subprogram by passing addresses to the subprogram; these addresses are assigned to the Linkage Section items declared in the USING list of that subprogram. Therefore, the number of data-names specified in matching CALL and Procedure Division USING lists must be identical. Up to 14 data-names are permitted.

Note

Correspondence between caller and callee lists are positional, not by identical spelling of names. For additional information, see CALL statement in Section 9, INTER-PROGRAM COMMUNICATION.

CLOSE

▶ Function

The CLOSE statement terminates the processing of files with optional lock where applicable.

Format one

```
CLOSE file-name-1 [, file-name-2] ...
```

Format two

```
CLOSE file-name-1 [WITH LOCK] ...
```

▶ Syntax rule

The files referenced in the CLOSE statement need not all have the same access or organization.

▶ General rules

1. Format one is the only option possible for both indexed and relative files.
2. A CLOSE statement must be executed upon completion of file processing, or before a STOP RUN is executed.
3. For this compiler, CLOSE statement options are treated as comments.

COMPUTE

▶ Function

The COMPUTE statement evaluates an arithmetic expression, a numeric-literal, or a data-name, and then stores the result in a designated numeric or numeric edited item.

Format

```
COMPUTE data-name-1 [ROUNDED] = { data-name-2
                                { numeric-literal
                                { arithmetic-expression }

    [; ON SIZE ERROR imperative-statement]
```

▶ Syntax rule

In general, **data-name** appearing to the left of = must refer to either an elementary numeric item or an elementary numeric edited item.

▶ General rule

The COMPUTE statement is governed by the regulations imposed by the statement components GIVING, ROUNDED, SIZE ERROR, as outlined in the General Rules, PROCEDURE DIVISION. It is also governed by the general regulations for Arithmetic Statements as described in FUNDEMENTAL CONCEPTS OF COBOL.

COPY

▶ Function

The COPY statement provides a means of including pre-written COBOL source coding in the programs at compile time; this is a compiler-directing function.

Format

```
COPY text-name [ { OF } library-name]
                 { IN }
```

Syntax rules

1. **OF** and **IN** are interchangeable and mutally exclusive.
2. A COPY statement may occur anywhere in the source program, in any Division where a character-string or a separator might usually occur, except that it may not occur within another COPY statement.

General rules

1. **Text-name** must be a unique name on the UFD (User's File Directory) which contains the COBOL program if the library-name is not specified.
2. If the text name is not on the same UFD as the program, **library-name** must be specified and must be the UFD name which contains the text-name.

   Examples:

```
FILE-CONTROL.  COPY text-name.
FD  MASTER-FILE COPY text-name OF SUB.
Ø1  MASTER-RECORD.  COPY text-name IN SUB.
SECTION-NAME SECTION.  COPY text-name.
PARAGRAPH-NAME.  COPY text-name IN SUB.
```

   Of the examples above, the first and fourth ones have copy members contained on the same UFD as the source program. The rest of them have copy members not contained in the source program UFD; these have copy members contained in a UFD named SUB.
3. The data preceding the COPY statement must not be contained within the copy member.

Example

The following is from Data Division coding in a source program.

```
Ø1  MASTER-DESCRIPTION.  COPY MASDES.
      .
      .
      .
      .
```

The text-name MASDES exists in the same UFD as the source program. It must not contain the 01 MASTER-DESCRIPTION entry; it might have the format:

```
05  BADGE-NO  PIC 9(5).
05  NAME.
    10  LAST-NAME  PIC X(15).
    10  FIRST-NAME  PIC X(15).
```

After compilation, examination of the listing file would reveal:

```
01  MASTER-DESCRIPTION.   (COPY MASDES.) (where the copy member is
    05  BADGE-NO  PIC 9(5).                comment only).
    05  NAME.
        10  LAST-NAME  PIC X(15).
        10  FIRST-NAME  PIC X(15).
```

Line numbering of the COPY file in the listing file is independent of the line numbers of the source.

Using the example above, the corresponding listing file might look like:

```
(0059)          .
(0060)          .
(0061)          .
(0062)  01  MASTER-DESCRIPTION.  COPY MASDES.
[0001]      05  BADGE-NO  PIC 9(5).
[0002]      05  NAME.
[0003]          10  LAST-NAME  PIC X(15).
[0004]          10  FIRST-NAME  PIC X(15).
(0062)  01  MASTER-DESCRIPTION.  COPY MASDES.
(0063)  01  EMPLOYMENT-HISTORY.
(0064)          .
(0065)          .
(0066)          .
```

DELETE

▶ Function

The DELETE statement logically removes a record from an indexed or relative file.

Format

DELETE file-name RECORD [; INVALID KEY imperative-statement]

▶ Syntax rule

The **INVALID KEY** option must not be specified for a DELETE statement referencing a file in SEQUENTIAL access mode. This was not allowed in the ANSI standard X3.23-1974.

▶ General rules

1. A DELETE statement logically removes a data record from a file. When operating on an indexed file, the DELETE statement removes all corresponding indices as well.
2. Execution of a DELETE statement does not affect the contents of a record area associated with **file-name**.
3. In SEQUENTIAL access, the record to be deleted must have been successfully read before a DELETE can be executed.

4. In indexed files with RANDOM or DYNAMIC access modes, the value of the record to be deleted must be placed in the RECORD KEY field.
5. In relative files with RANDOM or DYNAMIC access modes, the value of the record to be deleted must be placed in the RELATIVE KEY field.
6. For additional discussion, see Sections 12 and 13.

DISPLAY

Function

The DISPLAY statement causes low-volume data to be output to the appropriate hardware device.

Format

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

Syntax rules

1. The **mnemonic-name** must be specified in the SPECIAL-NAMES paragraph in the Environment Division.
2. The maximum total number of characters which may be output is 72.

General rules

1. When the **UPON** suffix is omitted, the system default is the standard display device, the on-line terminal.
2. If a **figurative-constant** is given as an operand, it will be displayed as a single character.
3. If a data item operand is packed, it is displayed as a series of digits followed by a separate trailing sign.

Examples

Type	Statement	Output
data-name	DISPLAY BADGE-NO	52207
data-name	DISPLAY 'BADGE-NO = ' BADGE-NO	BADGE-NO = 52207
literal	DISPLAY 'END-JOB'	END-JOB
figurative-constant	DISPLAY 'SELECT' ZERO	SELECT0

DIVIDE

Function

The DIVIDE statement divides one numeric data item into another and stores the quotient.

Format one

```
DIVIDE { data-name-1 } INTO data-name-2 [ROUNDED]
       { literal-1   }
    [; ON SIZE ERROR imperative-statement]
```

Format two

```
DIVIDE { data-name-1 } { INTO } { data-name-2 } GIVING data-name-3 [ROUNDED]
       { literal-1   } { BY   } { literal-2   }
    [; ON SIZE ERROR imperative-statement]
```

Syntax rules

1. Each **data-name** must refer to an elementary numeric item, except that a data-name associated with the GIVING phrase can refer either to an elementary numeric item or to an elementary numeric edited item.
2. Each literal must be a numeric literal.
3. The maximum size of each operand is 18 decimal digits. If all receiving data items were to be superimposed upon each other, aligned by their decimal points, their composite should not exceed 18 decimal digits in length.
4. Division by zero always causes a size-error condition.

General rules

1. In Format one, **data-name-1** or **literal-1** is divided into **data-name-2**; the quotient then replaces the dividend, data-name-2.
2. In Format 2, division occurs as in the cases below, and the quotient is stored in the data items following the word GIVING.
   - If the keyword **INTO** is used, the value of data-name-1 or literal-1 is divided into data-name-2 or literal-2 and the result is stored in data-name-3.
   - If the keyword **BY** is used, data-name-1 or literal-1 is divided by data-name-2 or literal-2 and the result is stored in data-name-3.
3. The REMAINDER clause of the DIVIDE statement is not supported. The user may substitute by a simple modification:

   For the statement:

```
DIVIDE data-name-1 by data-name-2 GIVING data-name-3
REMAINDER data-name-4.
```

   Substitute:

```
DIVIDE data-name-1 by data-name-2 GIVING data-name-3.
COMPUTE data-name-4 = data-name-1 - ( data-name-2 *
data-name-3 ).
```

ENTER

Function

The ENTER statement is classified as a compiler-directing statement; it acts as a modifer to a subsequent CALL statement and permits the use of more than one language in the same program.

Format

```
ENTER { COBOL     }
      { ASSEMBLER }
```

Syntax rules

1. A CALLed subprogram may be written in COBOL, FORTRAN, or Assembly, language, etc.. The parameter **ASSEMBLER** in the ENTER statement signifies a subprogram is other than COBOL.
2. The form ENTER COBOL may be used following a CALL statement; this traditional usage is optional. After any CALL statement, ENTER COBOL is assumed.
3. Each CALL upon an assembly Language subroutine must be preceded by its own ENTER ASSEMBLER statement.
4. The ENTER statement is optional in PRIME compiler.

General rule

The other language statements are executed in the called program as if they had been compiled in the called program following the ENTER statement. See INTER-PROGRAM COMMUNICATION for additional information.

EXHIBIT

Function

The EXHIBIT statement provides a means for displaying critical data at specified points in a procedure.

Format

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

General rules

1. The EXHIBIT statement is injected at critical points in the Procedure Division to provide debugging information. Specified data is EXHIBITED on the terminal.
2. The EXHIBIT statement differs from DISPLAY in that both the **data-name** and its value, connected by an '=' character, are printed. The '=' character is preceded and followed by a space.

   Example:

Statement	Output
EXHIBIT NAMED EMPLOYEE-NO	EMPLOYEE-NO = 950

EXIT

Function

The EXIT statement provides an end-point for a procedure.

Format

```
EXIT.
```

Syntax rules

1. The **EXIT** statement must appear in a sentence by itself.
2. For documentation purpose, the EXIT sentence may be the only sentence in the paragraph.

General rule

An EXIT statement serves only to enable the use to assign a procedure-name to a given point in a program. Such an EXIT statement has no other effect on the compilation or execution of the program.

EXIT PROGRAM

Function

The EXIT PROGRAM statement marks the logical end of a called program.

Format

```
EXIT PROGRAM.
```

Syntax rules

1. The **EXIT PROGRAM** statement must appear in a sentence by itself.
2. For documentation purpose, the EXIT PROGRAM sentence may be the only sentence in the paragraph. However, Prime COBOL does not require it.

General rules

1. The execution of an EXIT PROGRAM statement in a called program causes control to be passed to the calling program. Execution of an EXIT PROGRAM statement in a program which is not called behaves as if the statement were an EXIT statement.
2. If a main program contains an EXIT PROGRAM statement, C$IN will not ask for file assignments and will take the default VALUE OF FILE-ID value as defined in the FD.

GO TO

Function

The GO TO statement transfers control from one part of the PROCEDURE DIVISION to another, overriding the normal sequential execution of sentences.

Format one

```
GO TO procedure-name.
```

Format two

```
GO TO procedure-name-1 [procedure-name-2] ...

     DEPENDING ON data-name.
```

Syntax rules

1. A paragraph referenced by an ALTER statement can consist only of a paragraph header followed by a Format one GO TO statement.
2. In Format two, **data-name** must be an elementary, numeric integer.

General rules

1. A GO TO statement must not branch out of a range of the PERFORM statements.
2. A **procedure-name** must follow the GO TO statement. Otherwise, the compiler will abort with internal code.
3. When a Format one GO TO statement is executed, control is transferred to procedure-name, or to another paragraph-name if the GO TO statement has been modified by an ALTER statement.
4. When a GO TO statement represented by Format two is executed, control is transferred to procedure-name-1, procedure-name-2, etc., depending on the value of the identifier being 1, 2, ..., n. If the value of the identifier is anything other than the positive or unsigned integers 1, 2, ..., n, then no transfer occurs and control passes to the next statement in the normal sequence for execution.

IF

Function

The IF statement causes the evaluation of a condition (see Section 4, Conditional Expressions), permitting the execution of specified procedural statements if the condition is true.

Format

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

Syntax rule

The conditions in the IF statement must conform to the rules and outlining of conditions specified in Conditional Expressions, Section 4.

General rules

1. If the condition is true, any **ELSE** phrase is bypassed and either **statement-1** or **NEXT SENTENCE** (whichever was specified in the statement) is executed as follows:
   - Statement-1, if specified, is executed. Control then passes to the next executable sentence following the IF statement, unless statement-1 contains a procedure-branch or conditional statement, in which case control is transferred according to the rules for that statement.

- If the NEXT SENTENCE phrase is specified, control passes to the next executable sentence.

2. If the condition is false, any **statement-1** or its replacement **NEXT SENTENCE** which may be specified is bypassed, and control passes as follows:
   - Statement-2, if specified, is executed. Control then passes to the next executable sentence, unless statement-2 contains a procedure-branch or conditional statement, in which case control is transferred according to the rules for that statement.
   - If no ELSE statement-2 phrase is specified, or if the ELSE NEXT SENTENCE phrase is specified, control passes to the next executable sentence.
3. The IF statement is said to be nested whenever statement-1 and/or statement-2 contains another IF statement. If statements within IF statements are considered as paired IF and ELSE combinations, proceeding from left to right. Thus, any ELSE encountered applies to the immediately preceding IF which has not been already paired with an ELSE. It is not required that the number of ELSE's in a sentence be the same as the number of IF s.
4. The relation condition has the format:

Relation	Meaning
=	is equal to
<	is less than
>	is greater than
NOT =	is not equal to
NOT <	is not less than
NOT >	is not greater than

5. The class condition determining whether an operand is numeric or alphabetic. Its format is:

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

The **NUMERIC** test is valid only for a group, decimal, or character item. The **ALPHABETIC** test is valid only for a group or character item.

6. The **condition-name** condition tests the value or status of a conditional variable. Its format is:

IF [NOT] condition-name

The condition-name is defined as a level 88 data item in the record description entry in the Data Division.

In a condition-name condition, the first series of statements is executed if, and only if, the designated condition is true. The second series of statements is executed if, and only if, the designated condition is false. The second series (ELSE part) is terminated by a sentence-ending period. If there is no ELSE part to an IF statement, then the first series of statements must be terminated by a sentence-ending period.

Whether the condition is true or false, the next sentence is executed after

execution of the appropriate series of statements. If a GO TO is contained in the imperatives which are executed, or the normal flow of program steps is superseded because of an active PERFORM statement, the next sentence is not executed.

Example:

```
IF BALANCE = Ø GO TO NOT-FOUND.

IF X = 1.74 MOVE 'M' TO FLAG.

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

7. The sign condition tests an arithmetic expression to determine whether its value is greater than, less than, or equal to zero. The format is:

IF data-name IS [NOT] { **NEGATIVE** / **ZERO** / **POSITIVE** }

8. Two or more conditions can be combined by the logical operators **AND** and **OR**.The format for a combined condition is:

IF condition { { **AND** / **OR** } **condition** }

9. Comparisons employing the IF statement can be made involving indexed data items.
10. A nested IF exists when, in a single sentence, more than one IF precedes the first ELSE.

    Example:

```
IF X = Y  IF A = B

    MOVE "*" TO SWITCH
    ELSE MOVE "A" TO SWITCH
    ELSE MOVE SPACE TO SWITCH
```

    The flow of the above sentence may be represented by the tree structure in Figure 8-1.

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

```
              IF(1)        X = Y
              IF(2)        A = B
true                       true-action (2):   MOVE "*" TO SWITCH
action (1):   ELSE(2)      false-action(2):   MOVE "A" TO SWITCH
              ELSE(1)      false-action(1):   MOVE SPACE TO SWITCH
```

    The above illustration shows clearly the fact that IF(2) is wholly nested within the true-action side of IF(1).
11. It is not required that the number of ELSEs in a sentence be the same as the number of IFs; there may be fewer ELSE branches.

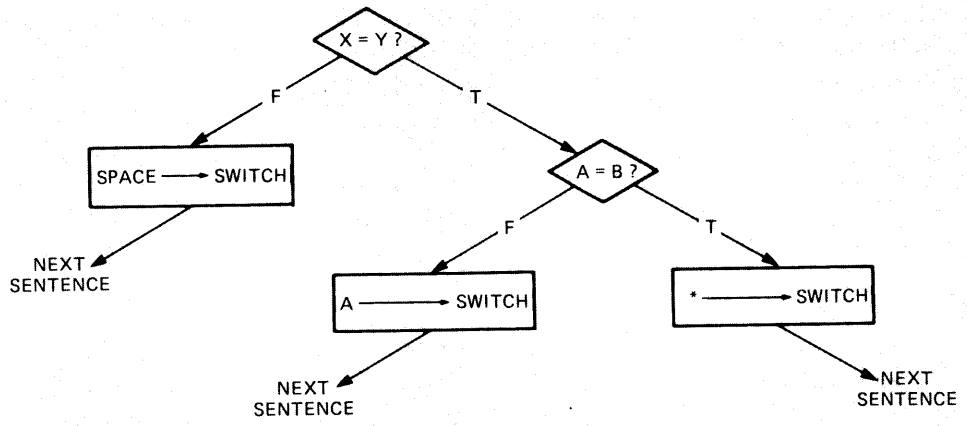


Figure 8-1. Nested IF Tree Structure

Examples:

```
IF M = 1  IF K = Ø
   GO TO M1KØ ELSE GO TO MNØT1.

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

In the latter case, IF(2) could equally well have been written as AND.

INSPECT

▶ Function

The INSPECT statement enables the programmer to examine a character-string item, to tally, replace, or tally and replace occurrences of single-characters in a data item.

Format

```
INSPECT data-name-1 TALLYING data-name-2 FOR { { ALL | LEADING } operand-2 | CHARACTERS }

[ { BEFORE | AFTER } INITIAL operand-3]

REPLACING { { ALL | LEADING | FIRST } operand-4 | CHARACTERS } BY operand-5

[ { BEFORE | AFTER } INITIAL operand-7]
```

Syntax rules

1. Data-name-1 must be a group item or an elementary item described (implicitly or explicitly) as USAGE IS DISPLAY.
2. Data-name-2 must be an elementary numeric data item.
3. Operands may either be data items or literals. If they are data items, operands must reference elementary alphabetic, alphanumeric or numeric items described (implicitly or explicitly) as USAGE IS DISPLAY. If they are literals, each operand must be a nonnumeric literal and may be any figurative constant, except ALL.

General rules

1. When both **TALLYING** and **REPLACING** clauses are present, the two clauses behave as if two INSPECT statements were written. The first contains only a TALLYING clause, the second contains only a REPLACING clause.
2. The **INSPECT** statement enables examination of a character-string item, permitting various combinations of the following actions:
   - Counting appearances of a specified character
   - Replacing a specified character by an alternative
   - Qualifying and limiting the above actions by keying those actions to the appearance of other specific characters
3. The TALLYING clause causes character-by-character comparison, from left to right, of **data-name-1**.
   - the user may initialize data-name-2 prior to the operation, but this is not required.
   - When AFTER INITIAL operand-3 sub-clause is present, the counting process begins only after detection of a character in data-name-1 matching operand-3. If BEFORE INITIAL operand-3 is specified, the counting process terminates upon encountering a character in data-name-1 which matches operand-3. The count is accumulated in data-name-2.
   - If the ALL phrase is specified, the content of data-name-2 is incremented by one for each occurrence of operand-2 matched within the content of data-name-1.
   - If the LEADING phrase is specified, the content of data-name-2 is incremented by one for each contiguous occurrence of operand-2 matched within the content of data-name-1, provided that the leftmost such occurrence is at the point where the comparison began and wherin operand-2 was eligible to participate.
   - If the CHARACTERS phrase is specified, the content of data-name-2 is incremented by one for each character in data-name-1.
4. The REPLACING clause causes replacement of characters under specified conditions.
   - If BEFORE INITIAL operand-7 is present, replacement does not continue after detection of a character in data-name-1 matching operand-7. If AFTER INITIAL operand-7 is present, replacement does not commence until detection of a character in data-name-1 matching operand-7.

- If the ALL phrase is specified, each occurrence of operand-4 matched within the content of data-name-1 is replaced by operand-5.
- If the LEADING phrase is specified, each contiguous occurrence of operand-4 matched within the content of data-name-1 is replaced by operand-5, provided that the leftmost occurrence is at the point where the comparison began and wherein operand-4 was eligible to participate.
- If the FIRST phrase is specified, the leftmost occurrence of operand-4 matched within the content of data-name-1 is replaced by operand-5.
- When the CHARACTERS phrase is specified, each character in data-name-1 is replaced by operand-5.

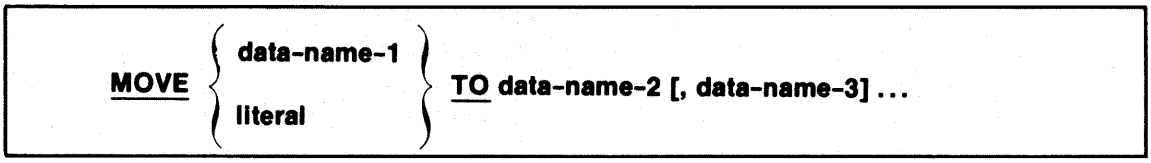
```
INSPECT name TALLYING countr FOR ALL 'L'.

  name Before     countr After     name After

   LILLY               3             LILLY
   SMALL               2             SMALL

INSPECT name TALLYING countr FOR LEADING 'B'
AFTER INITIAL 'A'
REPLACING CHARACTERS BY 'X'.

  name Before     countr Affter    name After

   ABACK               1             XXXXX
   CABBAGE             2             XXXXXXX

INSPECT name REPLACING CHARACTERS BY '$'
BEFORE INITIAL '.'.

  name Before     countr After     name After

   AB D.99                          $$$$.99

INSPECT name TALLYING countr FOR CHARACTERS
AFTER INITIAL 'E'
REPLACING ALL 'B' BY 'A'.

  name Before     countr After     name After

   DEBATE              4             DEAATE
   IBEX                1             IAEX

INSPECT name REPLACING FIRST 'A' BY 'P'
AFTER INITIAL 'M'.

  name Before     countr After     name After
   LLAMAA                           LLAMPA
   LLOYD                            LLOYD
```

MOVE

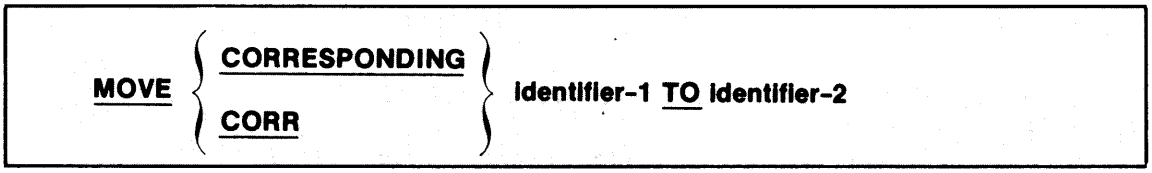
Function

The MOVE statement transfers data from one area of main storage to another, performing conversion and editing as indicated.

Format one



Format two



Syntax rule

Data-name-1, identifier-1, and **literal** represent the sending area; **data-name-2**, identifier-2, and **data-name-3** represent the receiving area.

General rules

1. When a group item is a receiving field, characters are moved without conversion and without editing.
2. During elementary moves, data is converted as necessary, editing occurs, and alignment is performed according to Standard Alignment Rules, LANGUAGE SPECIFICATIONS.
3. For numeric (external or internal decimal, binary, numeric literal) to numeric or numeric edited:
   - The items are aligned by decimal points, with generation of zeros or truncation on either end, as required.
   - When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place.
   - The items may have special editing performed on them with suppression of zeros, insertion of dollar signs, etc., and decimal point alignment, as specified by the receiving area. (This rule is only for numeric edited.)
4. For non-numeric source and targets:
   - The characters are placed in the receiving area from left to right (unless JUSTIFIED RIGHT applies).
   - If the receiving field is not completely filled by data being moved, the remaining positions are filled with spaces.
   - If the source field is longer than the receiving field, the move is terminated as soon as the receiving field is filled.
5. When overlapping fields are invoked, results are not predictable.
6. When **MOVE ALL** literal is used, the literal must be a single character. The receiving field is filled with the specified character.
7. When the CORRESPONDING option is used, identifier-1 and identifier-2 must be group items. Elementary items under identifier-1 are moved to the corresponding items under identifier-2.

If no correspondence is found, the compiler will return a warning message.

Note

Table C-6 in Appendix C summarizes the various types of moves permitted with the MOVE statement.

MULTIPLY

Function

The MULTIPLY statement computes the product of two numeric data items.

Format

```
MULTIPLY { data-name-1         }
         { numeric-literal-1   }

BY { data-name-2 [GIVING data-name-3]  }
   { numeric-literal-2 GIVING data-name-3 }

[ROUNDED] [ON SIZE ERROR imperative-statement]
```

Syntax rules

1. Each **data-name** must refer to an elementary numeric item, except that data-name-3 may be an elementary numeric edited item.
2. Each **literal** must be a numeric literal.
3. The maximum size of each operand is 18 decimal digits. The composite of operands, excluding those following GIVING, must not contain more than 18 decimal digits.

General rules

1. If the **GIVING** option is omitted, the second operand must be a data-name; the product will replace the second operand data-name.
2. Example:
   If the field BALANCE is to be multiplied by 1.03, it must be written as:

   ```
   MULTIPLY 1.Ø3 BY BALANCE.
   ```

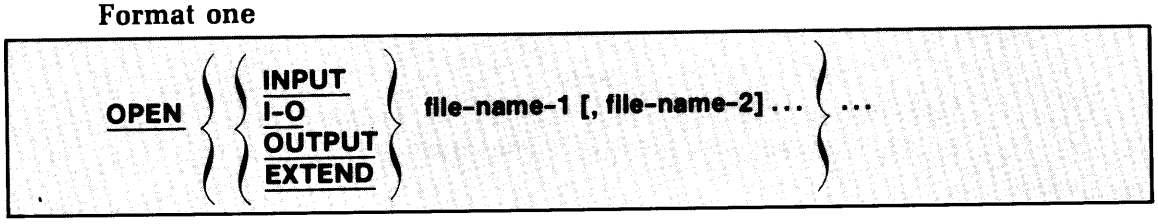
   Where the result will be stored in the data item named BALANCE.
3. When the GIVING option is taken, the product is stored in data-name-3.
4. The rules for signs are those presented in FUNDAMENTAL CONCEPTS OF COBOL, Algebriac Signs.

OPEN

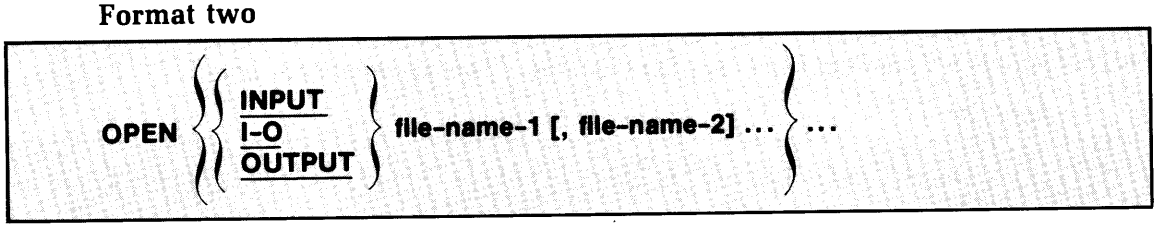
Function

The **OPEN** statement initiates the processing of files, and enables other input/output operations, such as label checking and writing.

Format one



Format two



▶ Syntax rules

1. There must be an OPEN statement for each file prior to a READ, WRITE, or REWRITE statement.
2. The files referred to in the OPEN statement need not all have the same organization or access.
3. The **EXTEND** phrase can be used only for sequential files.

▶ General rules

1. Format one is used for Sequential I/O.
2. Format 2 is used for Indexed I/O and Relative I/O.
3. A file opened as **INPUT** can only be accessed in a READ statement.
4. A file opened as **OUTPUT** can only be accessed in a WRITE statement.
5. A file opened as **I-O** can be accessed by a READ, REWRITE (disk only) or WRITE statement.
6. When the EXTEND phrase is specified, the OPEN statement opens the file, then positions to the bottom of that file (immediately following the last logical record). Subsequent WRITE statements to the file will add records as though the file had been opened with the OUTPUT phrase.
7. No statement which references a given file can be executed, either explicitly or implicitly, until an OPEN statement is successfully executed for that file.
8. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. (For permissible statements, see Table C-5 in Appendix C.)
9. If the OPEN statement does not produce access to the file (i.e., it cannot locate the desired file), the program will terminate abnormally at execution time.

Note

See Sections 12 and 13 for additional information on Indexed I/O and Relative I/O, respectively.

PERFORM

▶ Function

The PERFORM statement is used to transfer control explicitly to one or more procedures, and to return control implicitly to the normal sequence after transfer execution.

Format one

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2]
                           { THRU    }

[ { integer     } TIMES]
  { data-name-1 }
```

Format two

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2]
                           { THRU    }

[ VARYING { data-name-1 } FROM { data-name-2 } BY
          { index-name-1 }    { index-name-2 }
                              { literal-1    }

{ data-name-3 } ] UNTIL condition-1
{ literal-2   }
```

Format three

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2]
                           { THRU    }

VARYING { data-name-1 } FROM { data-name-2 }
        { index-name-1 }    { index-name-2 }
                            { literal-1    }

BY { data-name-3 } UNTIL condition-1
   { literal-2   }

[AFTER { data-name-4 } FROM { data-name-5 }
       { index-name-3 }    { index-name-4 }
                           { literal-3    }

BY { data-name-6 } UNTIL condition-2
   { literal-4   }

[AFTER { data-name-7 } FROM { data-name-8 }
       { index-name-7 }    { index-name-8 }
                           { literal-5    }

BY { data-name-9 } UNTIL condition-3 ]]
   { literal-6   }
```

Syntax rules

1. The words **THROUGH** and **THRU** are equivalent.
2. Each **data-name** represents an elementary numeric item described in the Data Division.
3. Each **literal** represents a numeric literal.
4. If an **index-name** is specified in the **VARYING** or **AFTER** phrase, then:
   - The data-name in the associated FROM and BY phrases must be an integer data item.
   - The literal in the associated FROM phrase must be a positive integer.
   - The literal in the associated BY phrase must be a non-zero.
5. If an **index-name** is specified in the **FROM** phrase, then:
   - The data-name in the associated VARYING or AFTER phrase must be an integer data item.
   - The data-name in the associated BY phrase must be an integer data item.
   - The literal in the associated BY phrase must be an integer.

   Note

   Integer is a numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point.
6. Literal in the BY phrase must not be zero.
7. **Condition-1, condition-2, condition-3** may be any conditional expression as described in FUNDAMENTAL CONCEPTS OF COBOL, Simple Conditional Expressions.

General rules

1. If the PERFORM statement is written with no options, control is transferred to the first statement of **procedure-name-1**. At the completion of procedure-name-1, control is implicitly returned to the next executable statement following the PERFORM statement.
2. If procedure-name-2 is specified and it is a paragraph-name, then the control is returned to the next sequential instruction after the last statement of that paragraph.
3. If procedure-name-2 is specified and it is a section-name, then the control is returned to the next sequential instruction after the last statement of the last paragraph of that section.
4. In Format one:
   - If the **THROUGH** option is taken, multiple paragraphs or sections can be executed before control is returned to the next sequential statement.
   - If the **TIMES** option is taken, procedures are performed the number of times specified by data-name-1 or integer. At the completion of procedure-name-2, control is returned to the statement following PERFORM statement.
   - **Data-name-1** or **integer** must be a positive numeric integer which cannot be greater than 32,767.
   - If data-name-1 or integer is initially zero or negative, the PERFORM statement is not executed; control passes to the statement following PERFORM statement.

5. In Format two:
   - If the **UNTIL** option is taken, successive execution of procedures occurs until a condition is satisfied.
   - The statement is coded as:

     PERFORM procedure-name-1 [THRU procedure-name-2]

     UNTIL condition-1.

   - **Condition-1** must be a simple condition, excluding an ELSE phrase. The condition is tested prior to execution of the PERFORM statement. If the condition is not met, PERFORM is executed until the condition is satisfied. If the condition is satisfied prior to execution of the PERFORM statements, PERFORM is not executed and control passes to the next sequential instruction.
   - If all options are used to vary the values referred to by data-name-1 or **index-name-1**:
   - The condition is tested prior to execution of the PERFORM statement. If the condition is true, PERFORM is not executed; control passes to the next sequential instruction.
   - If the condition is false, data-name-1 is set to the current value of data-name-2 or literal-1 at the point of initial execution of the PERFORM statement. If the condition is still false, procedure-name-1 THRU procedure-name-2 are executed once.
   - The value of data-name-1 is incremented or decremented by the value in **data-name-3** or **literal-2**. The condition is reevaluated. The cycle continues until the condition is satisfied, at which point control is transferred to the next executable statement following PERFORM statement. See Figure 8-3.
   - At the termination of PERFORM statement, data-name-1 or index-name-1 has a value which exceeds the last used setting by the value of data-name-3 or literal-2. If the condition was true before initial execution of PERFORM statement, data-name-1 or index-name-1 contains the current value of data-name-2 or **index-name-2**.

6. In Format three:
   - The rules related to varying one identifier are shown in Rule 5 above.
   - When two identifiers are varied, data-name-1 and **data-name-4** are set to the current value of data-name-2 and **data-name-5**, respectively. After the identifiers have been set, condition-1 is evaluated; if true, control is transferred to the next executable statements; if false, condition-2 is evaluated. If condition-2 is false, procedure-name-1 through procedure-name-2 is executed once, then data-name-4 is augmented by **data-name-6** or **literal-4** and condition-2 is evaluated again. This cycle of evaluation and augmentation continues until this condition is true. When condition-2 is true, data-name-4 is set to the value of literal-3 or the current value of data-name-5, data-name-1 is augmented by data-name-3 and condition-1 is reevaluated.

The PERFORM statement is completed if condition-1 is true; if not, the cycles continue until condition-1 is true.

- During the execution of the procedures associated with the PERFORM statement, any change to the VARYING variable (data-name-1 and index-name-1), the BY variable (data-name-3), the AFTER variable (data-name-4 and index-name-3), or the FROM variable (data-name-2 and index-name-2) will be taken into consideration and will affect the operation of the PERFORM statement.
- At the termination of the PERFORM statement, data-name-4 contains the current value of data-name-5. Data-name-1 has a value that exceeds the last used setting by an increment or decrement value, unless condition-1 was true when the PERFORM statement was entered, in which case data-name-1 contains the current value of data-name-2.
- When two identifiers are varied, data-name-4 goes through a complete cycle (FROM, BY, UNTIL) each time data-name-1 is varied. See Figure 8-4.
- When three identifiers are varied, the mechanism is the same as for two identifiers except that data-name-7 goes through a complete cycle each time that data-name-4 is augmented by data-name-6 or literal-4, which in turn goes through a complete cycle each time data-name-1 is varied. See Figure 8-5.
- After the completion of PERFORM statement, each data item varied by an AFTER phrase contains the current value of the data-name in the associated FROM phrase. Data-name-1 has a value that exceeds its last used setting by one increment or decrement value, unless condition-1 is true when the PERFORM statement is entered, in which case data-name-1 contains the current value of data-name-2.
- An example for a Format three PERFORM statement is shown below:

```
                    .
                    .
                    .
                    .
START-PARA.
    PERFORM INT-PARA
        VARYING INDX1 FROM 1 BY 1
          UNTIL INDX1 > 2
        AFTER INDX2 FROM 1 BY 1
          UNTIL INDX2 > 12
        AFTER INDX3 FROM 1 BY 1
          UNTIL INDX3 > 7.
    GO TO SORT-PARA.

INT-PARA.
    MOVE ZEROS TO DEPT-TOTAL(INDX1, INDX2, INDX3).
                    .
                    .
                    .
                    .
```

7. The only necessary relationship between procedure-name-1 and procedure-name-2 is that the sequence of operations is executed, beginning at the procedure-name-1 and ending with procedure-name-2, so that control can be implicitly transferred to the next executable statement following the PERFORM statement.

   GO TO, PERFORM and CALL statements may occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the common return point, then, for documentation purposes, procedure-name-2 may be the name of a paragraph consisting of an EXIT statement, to which all of these paths may lead. For example:

```
    PERFORM-1
      PERFORM INT-PARA THRU EXIT-PARA.
      ADD TOTAL-1, TOTAL-2, TOTAL-3 GIVING DEPT-TOTAL.
        .
        .
        .
INT-PARA.
    IF INDX1 = 2      GO TO PATH-1.
    IF INDX2 = 12     GO TO PATH-2.
    IF INDX3 = 7      GO TO PATH-3.
PATH-1.
        .
        .
        .
    GO TO EXIT-PARA.
PATH-2.
        .
        .
        .
    GO TO EXIT-PARA.
PATH-3.
        .
        .
        .
EXIT-PARA.
    EXIT.
```

8. If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements may not have a common exit. See Figure 8-2.

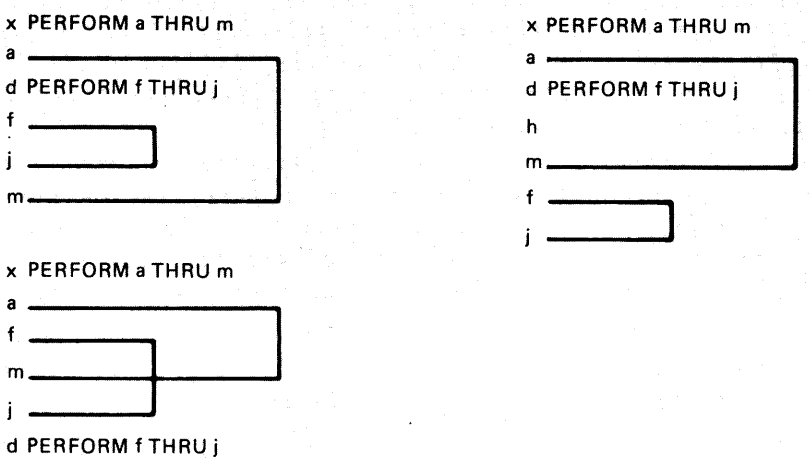


Figure 8-2 Permissible PERFORM Sequences

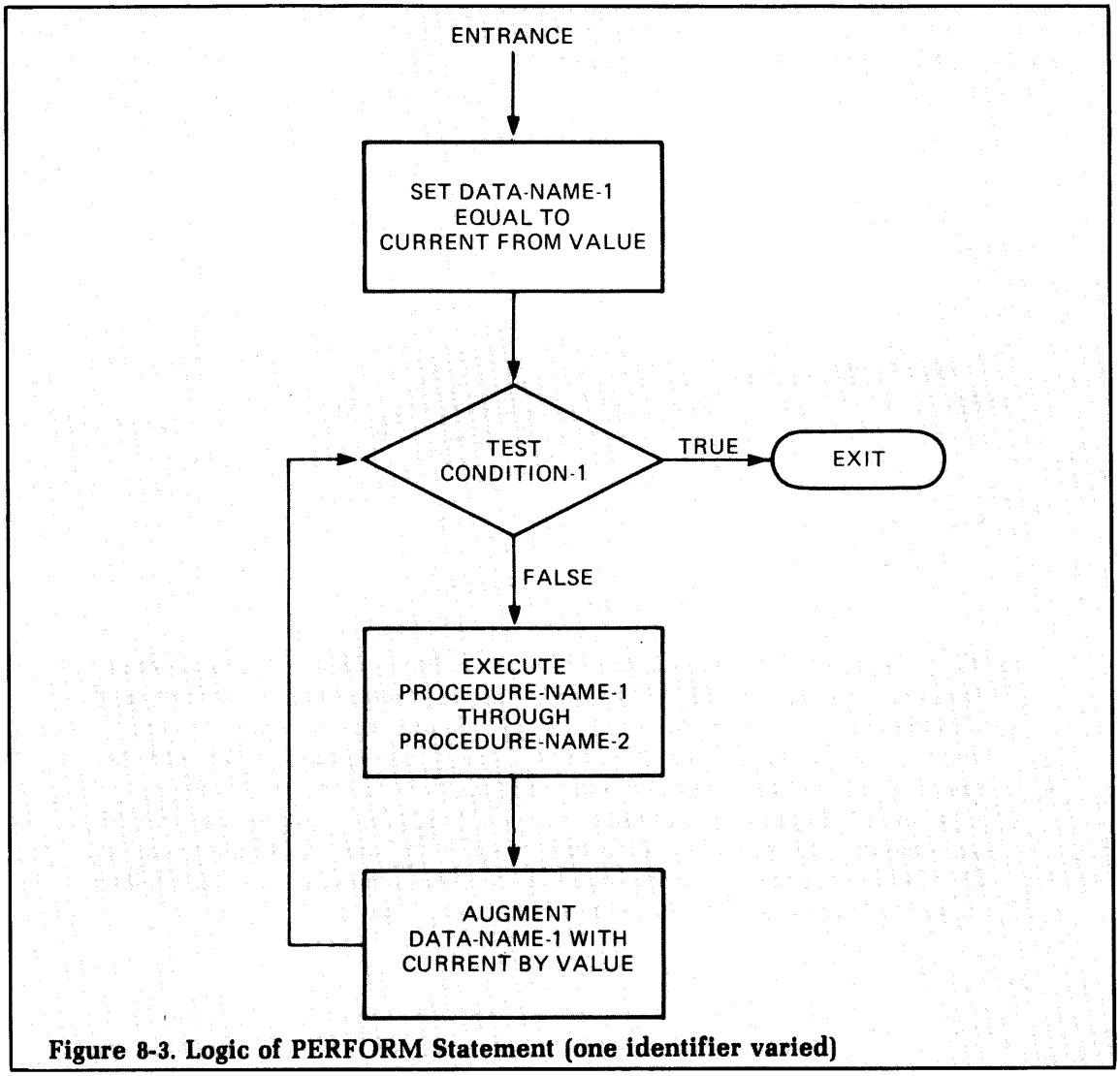


Figure 8-3. Logic of PERFORM Statement (one identifier varied)

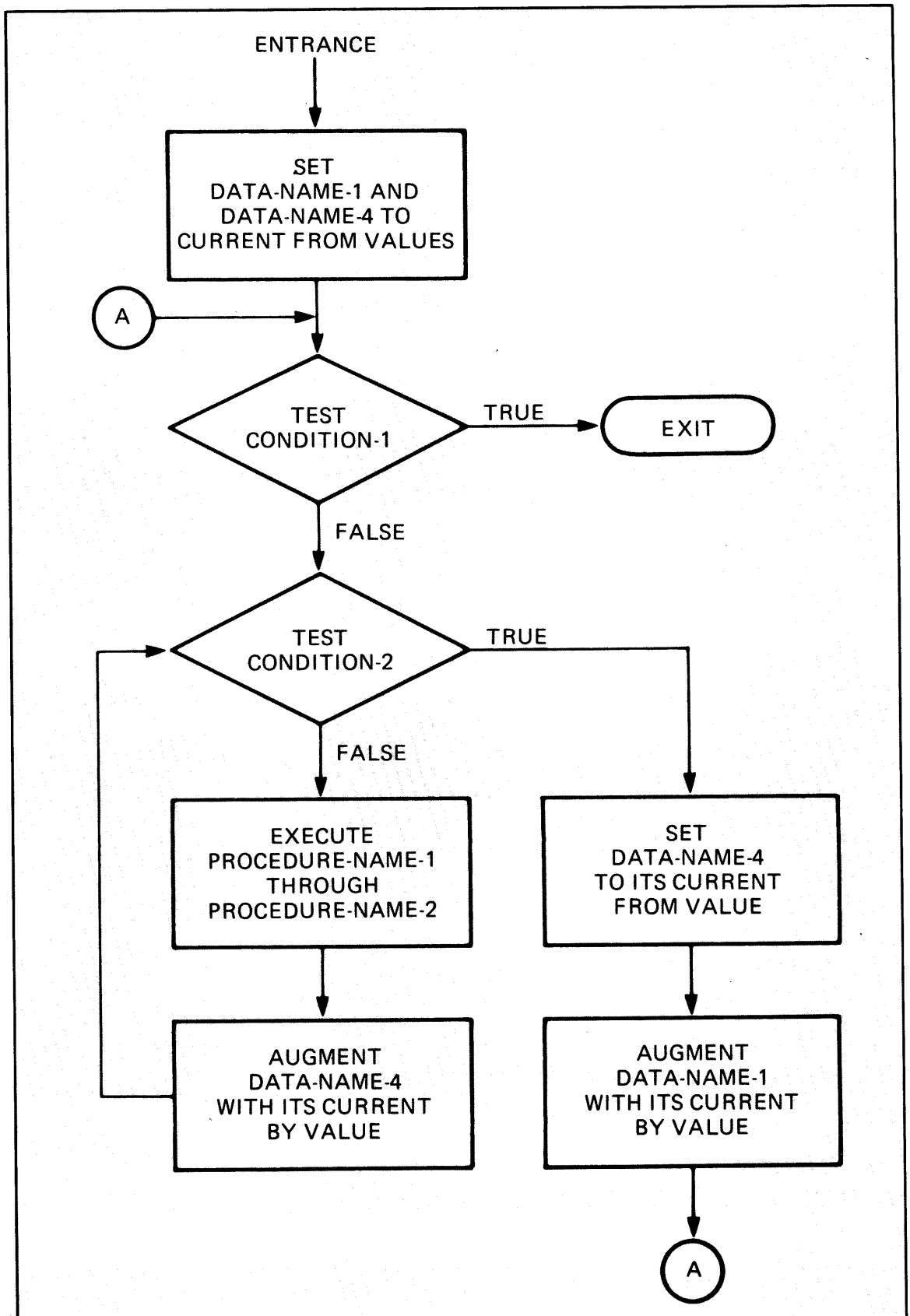


Figure 8-4. Logic of PERFORM Statement (two identifiers varied)

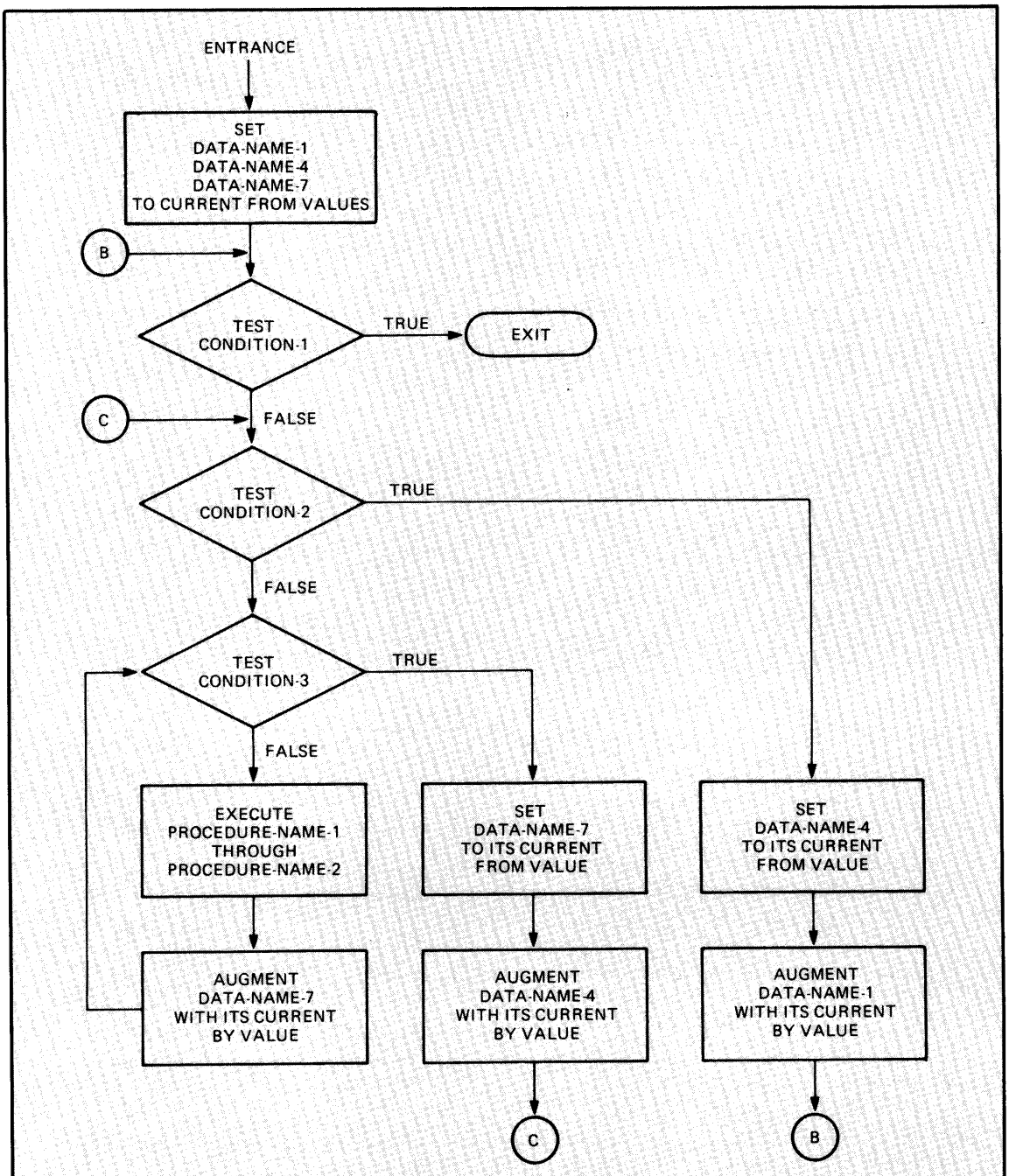


Figure 8-5. Logic of PERFORM Statement (three identifiers varied)

READ

▶ Function

The READ statement makes available a record from a file.

▶ Format one

```
READ file-name [NEXT] RECORD [INTO data-name-1]

     [AT END imperative statement]
```

Format two

```
READ file-name RECORD [INTO data-name-1] [KEY IS data-name-2]

     [INVALID KEY imperative-statement]
```

Syntax rules

1. Format one is used for all sequentially read files.
2. The **NEXT** phrase option in Format one is used only for Indexed and Relative I/O files, in SEQUENTIAL or DYNAMIC access modes, when records are to be retrieved sequentially.
3. Format two is used only for Indexed I/O and Relative I/O files.
4. The **KEY IS** option of Format two is used only for Indexed I/O files.

General rules

1. A file must be OPEN in the INPUT or I/O mode when a READ statement for that file is executed.
2. The READ statement makes a record available to the program before execution of any subsequent statement, provided **AT END** or **INVALID KEY** are not invoked.
3. Format one, without the NEXT option, is used for Sequential I/O files. The **INTO** option permits the user to specify that a copy of the data record is to be placed into a data area immediately after the READ statement. The **data-name** must not be defined in the file itself.

   If end-of-file occurs, but there is no AT END clause in the READ statement, an applicable Declaratives procedure is performed. If neither AT END nor Declaratives exists, an execution I/O error occurs.
4. Format one, without the NEXT option, is used for sequential reads of Indexed I/O files in SEQUENTIAL access mode. The read is based on the primary index (RECORD KEY).
5. Format one, without the NEXT option, is used for sequential reads of Relative I/O files in SEQUENTIAL access mode. The read is based on the RELATIVE KEY.
6. Indexed and Relative I/O files in DYNAMIC mode, may be read sequentially, rather than randomly, by use of the NEXT option.
7. For General Rules 4, 5, and 6 above, if the INTO clause is used, the data record is automatically moved into data-name-1. When AT END is specified, control is passed to the imperative-statement after the complete file has been read.
8. For Indexed I/O files in DYNAMIC and RANDOM mode, if NEXT is not specified, and the file is to be read sequentially, the value of the record to be retrieved must be placed in the RECORD KEY data-name.
9. For Relative I/O files, if NEXT is not specified, and the file is to be read sequentially, the value of the record to be retrieved must be placed in the RELATIVE KEY data-name.
10. For Indexed I/O files read sequentially, if one of the secondary index sequences is to be used, the index must first be established with a Format two statement. Thereafter, a Format one statement may be used.
11. For Sequential I/O disk files containing packed or binary data, the user should specify UNCOMPRESSED in the FD entry for that file.

12. Further detailed discussion of READ statement formats as they apply to Indexed I/O files and Relative I/O files will be found in Sections 12 and 13, respectively.

READY TRACE

Function

The READY TRACE statement turns on a Prime tracing function to assist in determining the point at which actual flow departs from expected flow.

Format

```
READY TRACE.
```

Syntax rule

The execution of the trace mode may be set or reset dynamically.

General rule

After a READY TRACE statement is executed, each time a paragraph or section in the Procedure Division is entered, that paragraph or section name is output to the terminal to provide debugging information. The format printed is:

ENTER: section-name/paragraph-name

RELEASE

Function

The RELEASE statement transfers records to the initial phase of a SORT operation.

Format

```
RELEASE record-name [ FROM identifier]
```

Syntax rule

A RELEASE statement may only be used within an input procedure associated with a SORT statement for a file whose SD entry contains the record-name.

Note

For complete discussion, see Section 11, SORT MODULE.

RESET TRACE

Function

This statement turns off the Prime tracing function.

Format

```
RESET TRACE.
```

General rule

The RESET TRACE statement can only occur after the execution of a READY TRACE statement.

RETURN

Function

The RETURN statement obtains sorted records from the final phase of a SORT operation.

Format

```
RETURN file-name RECORD [INTO identifier]

    AT END imperative-statement
```

Syntax rule

A RETURN statement may be used only within an output procedure associated with a SORT statement for file-name described by an SD entry.

Note

For complete information, see Section 11, SORT MODULE.

REWRITE

Function

The REWRITE statement logically replaces a record existing in a disk file.

Format

```
REWRITE record-name [FROM data-name]

    [INVALID KEY imperative-statement]
```

Syntax rules

1. **Record-name** and **data-name** must not refer to the same storage area.
2. Record-name is the name of a logical record in the File Section and may be qualified.

General rules

1. The file containing record-name must be a disk file and must be open for I/O (in all access methods) prior to execution of a REWRITE statement.
2. If the **FROM** option is used, the information in data-name is moved to the record area prior to the REWRITE. For Indexed I/O files, the primary RECORD KEY must equal the key from the previous READ, or the INVALID KEY conditions will occur.
3. A record must have been READ successfully prior to a REWRITE statement. This is required to lock the record to ensure that it cannot be updated by another program running concurrently.
4. The **INVALID KEY** option is not used for Sequential I/O files. The file status field, if specified, is updated by the REWRITE statement.
5. For Indexed I/O files, control is passed to the INVALID KEY statement if the primary key is changed. If this option is not written, control passes to the USE DECLARATIVES. One or the other of these options must be taken for indexed files. Refer to Appendix C for status codes.
6. The REWRITE statement must not change or rewrite the primary RECORD KEY in Indexed I/O files.

7. For Relative I/O files, control is passed to the INVALID KEY statement if the RELATIVE KEY is changed after a successful READ. If the INVALID KEY option is not taken, control passes to the USE DECLARATIVES. One or the other of these options must be taken.
8. A sequential file using REWRITE must be a COBOL-created file other than a printer file, or any uncompressed file.

Note

See Sections 12 and 13 for additional information on Indexed I/O and Relative I/O, respectively.

SEARCH

▶ Function

The SEARCH statement is used to search a table for a table element which satisfies the specified condition, and to adjust the associated index-name to indicate that table element.

Format one

```
SEARCH identifier-1 [ VARYING { identifier-2  } ]
                              { index-name-1  }

[; AT END imperative-statement-1]

 ; WHEN condition-1 { imperative-statement-2 }
                   { NEXT SENTENCE          }

[; WHEN condition-2 { imperative-statement-3 } ]
                    { NEXT SENTENCE          }
```

Format two

```
SEARCH ALL identifier-1 [; AT END imperative-statement-1]

    ; WHEN { data-name-1      { EQUALS       } { identifier-3 } }
          {                  { IS EQUAL TO  } {             } }
          {                  { IS =         } { literal-1   } }
          { condition-name-1                                  }

    [ AND { data-name-2      { EQUALS       } { identifier-4 } } ]
          {                  { IS EQUAL TO  } {             } }
          {                  { IS =         } { literal-2   } }
          { condition-name-2                                  }

      { imperative-statement-2 }
      { NEXT SENTENCE          }
```

Syntax rules

1. In both Formats one and two, **identifier-1** must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause.
2. **Identifier-2**, when specified, must be described as USAGE IS INDEX, or as a numeric elementary item without any positions to the right of the assumed decimal point.
3. In Format two, the description of identifier-1 must contain the KEY IS phrase in its OCCURS clause.

Note

A complete discussion of the SEARCH verb is presented in Section 10, TABLE HANDLING.

SET

Function

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements.

Format one

```
SET { index-name-1 [, index-name-2] ... } TO { index-name-3 }
    { data-name-1 [ data-name-2] ...  }      { data-name-3 }
                                             { integer-1   }
```

Format two

```
SET index-name-4 [, index-name-5] ... { UP BY   } { data-name-6 }
                                      { DOWN BY } { integer-2   }
```

Syntax rules

1. There must not be a name specifying an index data item after **UP BY** or **DOWN BY** option.
2. **Data-name-4** must be described as an elementary numeric integer.
3. **Integer-1** and **integer-2** may be signed. Integer-1 must be a positive value.

General rules

1. Format one is equivalent to moving the value in **index-name-3, data-name-3** or integer-1 to multiple receiving fields written immediately after the SET verb.
2. Format two is equivalent to reduction (DOWN), or increase (UP), applied to each of the quantities written immediately after the SET verb. The amount of the reduction or increase is specified by a name or value immediately following the word BY.
3. An index-name should only apply to the OCCURS which defines it.

Note

See Section 10, TABLE HANDLING, for complete information.

SORT

Function

The SORT statement creates a sort-file by executing input procedures or by transferring records from another file, sorts the records in the sort-file on a set of specified keys, and makes available the sorted records to output procedures or to an output file.

Format

```
SORT file-name-1 ON { ASCENDING  } KEY data-name-1 [, data-name-2] . . .
                    { DESCENDING }

{ INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2] }
{                                     { THRU    }                 }
{ USING file-name-2                                               }

{ OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
{                                      { THRU    }                  }
{ GIVING file-name-3                                                }
```

Syntax rules

1. SORT statements may appear anywhere except in the Declaratives portion of the Procedure Division or in an input or output procedure associated with a SORT statement.
2. In the Data Division, **file-name-1** must be described in an SD entry; **file-name-2** and **file-name-3** must be described in an FD entry.

General rules

1. At the time of execution of the SORT statement, neither file-name-2 nor file-name-3 may be open.
2. If the **USING** phrase is specified, all the records in file-name-2 are automatically transferred to file-name-1.
3. If the **GIVING** phrase is specified, all the sorted records are automatically written on file-name-3 as the implied output procedure for the SORT statement.

Note

A complete discussion of the SORT statement is presented in Section 11, SORT MODULE.

In addition to the SORT statement, two other suitable sort facilities are available to COBOL programs—the PRIMOS external sort utility (explained in the Prime User's Guide) and the internal sort subroutines (explained in the PRIMOS Subroutines Reference Guide).

START

Function

The START statement provides a basis for logical positioning, within an Indexed I/O or Relative I/O file, for subsequent sequential or dynamic retrieval of records.

Format

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

[INVALID KEY imperative-statement]
```

Syntax rule

File-name must be the name of a file with sequential or dynamic access.

General rules

1. Option 1:

   START file-name.

   - In an Indexed file, this option positions the file to the value contained in the RECORD KEY data-name.
   - In a Relative file, this option positions the file to the value contained in the RELATIVE KEY data-name.
   - In either file structure, if the indicated record is not present in the file, control is passed to DECLARATIVES section if present; otherwise, the program terminates.

2. Option 2:

   START file-name KEY IS data-name.

   - In an Indexed file, this option will position the file to the value contained in data-name (data-name is the name of either RECORD KEY or one of the ALTERNATE RECORD KEYs).
   - In a Relative file, this option will position the file to the value contained in data-name as defined in RELATIVE KEY.
   - In either file structure, if the indicated record is not present in the file, control is passed to the DECLARATIVES section if present; otherwise, the program terminates.

3. Option 3:

   ```
   START file-name

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

   [INVALID KEY imperative-statement]
   ```

For both Indexed I/O and Relative I-O files, if the option **GREATER** or **NOT LESS** is specified, the file is positioned for the next access to be greater than or less than the value specified in the data-name.

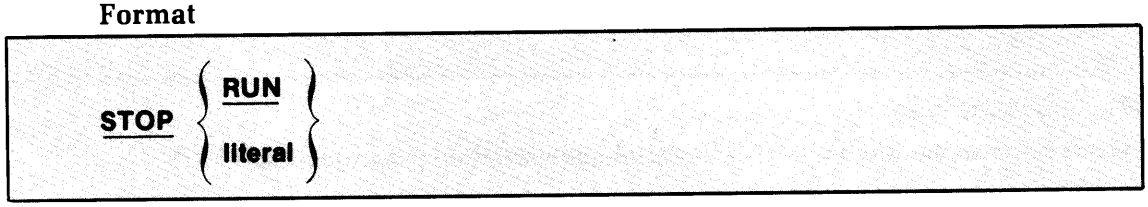
4. The INVALID clause or DECLARATIVES is taken if there is no data satisfying data-name and the STATUS code returned is a 23.

STOP

Function

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

Format



Syntax rule

If a STOP RUN statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

General rules

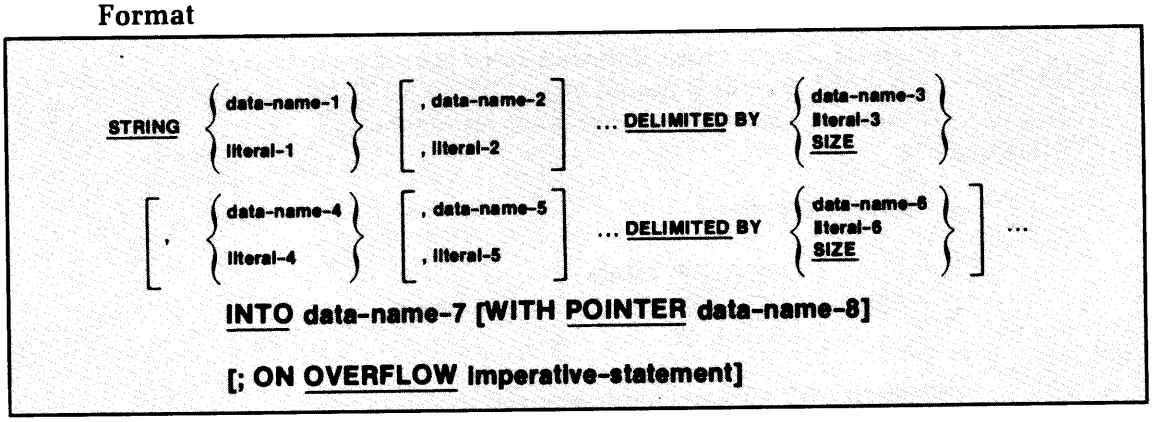
1. **STOP RUN** terminates execution of a progam, returning control to the operating system.
2. STOP RUN cannot be used in a called program.
3. If STOP **literal** is specified, the literal is communicated on the console, and execution is suspended. Execution is resumed at the next executable statement in sequence after operator intervention. Presumably, the operator performs a function suggested by the contents of the literal, prior to resuming program execution.

STRING

Function

The STRING statement provides juxtaposition of the partial or complete contents of two or more data items into a single data item.

Format



Syntax rules

1. Each **literal** may be any figurative constant (without the optional word ALL).
2. All literals must be described as nonnumeric literals. All **data-names**, except **data-name-8**, must be described implicitly or explicitly as usage is DISPLAY.
3. **Data-name-7** must represent an elementary alphanumeric data item without editing symbols or the JUSTIFIED clause.
4. **Data-name-8** must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size of data-name-7 + 1. The symbol P may not be used in the PICTURE character-string of data-name-8.
5. Where **data-name-1, data-name-2, ...,** or **data-name-3** is an elementary numeric data item, it must be described as an integer without the symbol P in its PICTURE character-string.

General rules

1. All references to data-name-1, data-name-2, data-name-3, literal-1, literal-2, literal-3 apply equally to data-name-4, data-name-5, data-name-6, literal-4, literal-5, and literal-6, respectively, and all recursions thereof.
2. Data-name-1, literal-1, data-name-2, literal-2, represent the sending items. Data-name-7 represents the receiving item.
3. Literal-3, data-name-3, indicate the character(s) delimiting the move. If the **SIZE** phrase is used, the complete data item defined by data-name-1, literal-1, data-name-2, literal-2, is moved. When a figurative constant is used as the delimiter, it stands for a single character nonnumeric literal.
4. When a figurative constant is specified as literal-1, literal-2, literal-3, it refers to an implicit one character data item whose usage is DISPLAY.
5. When the STRING statement is executed, the transfer of data is governed by the following rules:
   - Those characters from literal-1, literal-2, or from the contents of the data item referenced by data-name-1, data-name-2, are transferred to the contents of data-name-7 in accordance with the rules for alphanumeric to alphanumeric moves, except that no space-filling will be provided.
   - If the **DELIMITED** phrase is specified without the SIZE phrase, the contents of the data item referenced by data-name-1, data-name-2, or the value of literal-1, literal-2, are transferred to the receiving data item, this occurs in the sequence specified in the STRING statement, beginning with the leftmost character and continuing from left to right until the end of the data item is reached, or until the character(s) specified by literal-3, or by the contents of data-name-3 are encountered. The character(s) specified by literal-3 or by the data item referenced by data-name-3 are not transferred.
   - If the DELIMITED phrase is specified with the SIZE phrase, the entire contents of literal-1, literal-2, or the contents of the data item referenced by data-name-1, data-name-2, are transferred. The transfer proceeds in the

sequence specified in the STRING statement to the data item referenced by data-name-7, until all data has been transferred or the end of the data item referenced by data-name-7 has been reached.

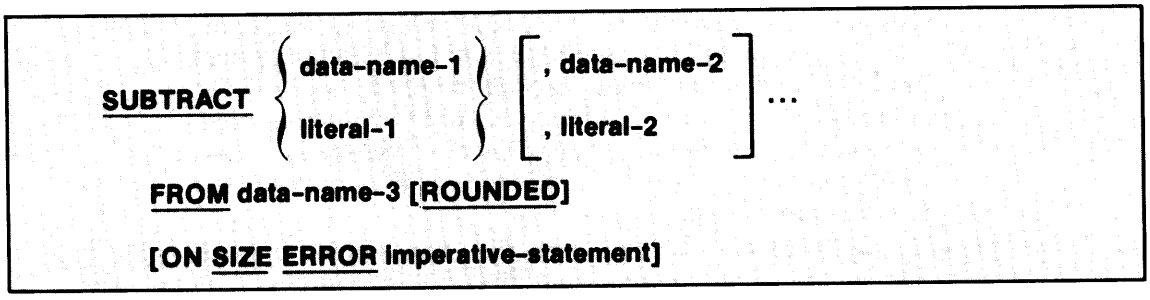
6. If the **POINTER** phrase is specified, data-name-8 is explicitly available to the programmer. The programmer is then responsible for setting its initial value. The initial value must not be less than one.
7. If the POINTER phrase is not specified, the following general rules apply as if the user had specified data-name-8 with an initial value of 1.
8. When characters are transferred to the data item referenced by data-name-7, the transfer behaves as though characters were moved, one at a time, from the source to the data item character position referenced by data-name-7 and designated by the value of data-name-8. Data-name-8 is increased by one prior to the move of the next character. The value associated with data-name-8 is changed during execution of the STRING statement only by the behavior specified above.
9. At the end of execution of the STRING statement, only the portion of the data item referenced by data-name-7 (that which was referenced during the execution of the STRING statement) is changed. All other portions of the data item referenced by data-name-7 will contain data which was present before this execution of the STRING statement.
10. Data transfer to data-name-7 terminates when the value in data-name-8 is either less than 1, or exceeds the number of character positions in data-name-7. Such termination may occur at any point at or after initialization of the STRING statement. If termination occurs as a result of such a condition, the imperative statement in an ON OVERFLOW phrase is executed, if specified.
11. If the **ON OVERFLOW** phrase is not specified when the conditions described in General Rule 10 above are encountered, control is transferred to the next executable statement.
12. If delimiters exceed the maximum number of five, the compiler will abort with error code 113.

SUBTRACT

▶ Function

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

Format one



Format two

```
SUBTRACT { data-name-1 } [ , data-name-2 ] ...
         { literal-1   }   [ , literal-2 ]

    FROM { data-name-3 } GIVING data-name-4 [ROUNDED]
         { literal-3   }

    [ON SIZE ERROR imperative-statement]
```

Format three

```
SUBTRACT { CORRESPONDING } identifier-1
         { CORR          }

    FROM identifier-2 [ROUNDED]

    [ON SIZE ERROR imperative-statement]
```

Syntax rules

1. Each **data-name** must refer to a numeric elementary item, except that **data-name-4** (following GIVING) may be an elementary numeric edited item.
2. Each **literal** must be a numeric literal.
3. The maximum size of each operand is 18 decimal digits. If all receiving data items were to be superimposed upon each other, aligned by their decimal points, their composite could not exceed 18 decimal digits in length.
4. In Format three, both **identifier-1** and **identifier-2** must be group items.

General rules

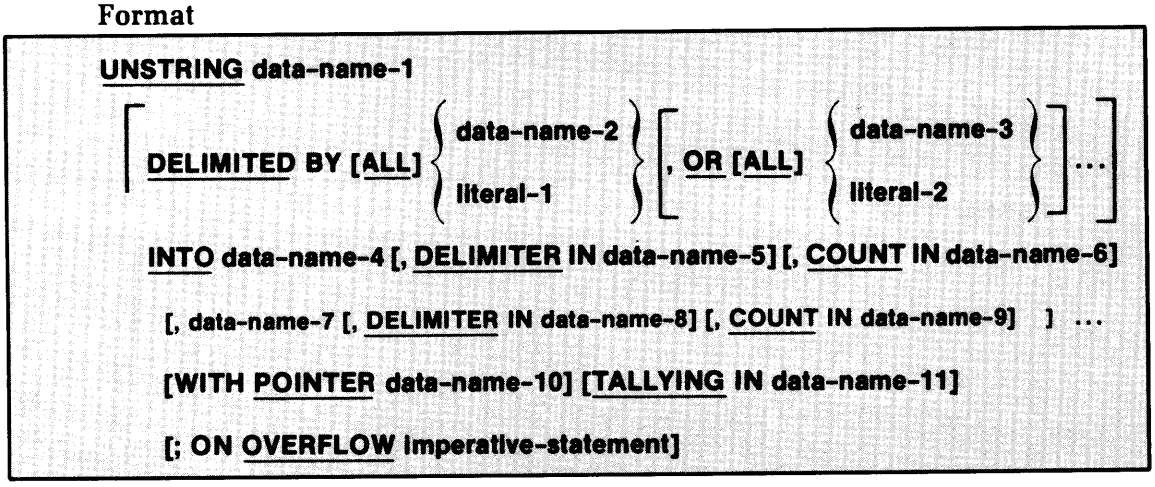
1. In Format one, the effect of the SUBTRACT statement is to sum the values of all the operands which precede FROM, and then to subtract that sum from the value of the item following FROM. The result is stored in **data-name-3**.
2. In Format two, all literals and data-names preceding FROM are added together, the sum is subtracted from data-name-3 or **literal-3**, and the result is stored in **data-name-4**.
3. See the rules for arithmetic statements under PROCEDURE DIVISION, General Rules. The **ROUNDED** and **ON SIZE ERROR** options may be used when truncation of results could occur.
4. The rules for signs are those presented in FUNDAMENTAL CONCEPTS OF COBOL, Algebraic Signs.
5. In Format three, each elementary item under identifier-1 is subtracted from and stored into the corresponding elementary item under identifier-2.

UNSTRING

Function

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

Format



Syntax rules

1. The **ALL** phrase option is not the figurative constant ALL.
2. Each **literal** must be a nonnumeric literal. In addition, each literal may be any figurative constant without the optional word ALL.
3. **Data-name-1, data-name-2, data-name-3, data-name-5**, and **data-name-8**, must be described, implicitly or explicitly, as an alphanumeric data item.
4. **Data-name-4** and **data-name-7** may be described as either alphabetic (except that the symbol B may not be used in its picture-string), alphanumeric, or numeric (except that the symbol P may not be used in its picture-strings), and must be described as usage is DISPLAY.
5. **Data-name-6, data-name-9, data-name-10, data-name-11** must be described as elementary numeric integer data items (except that the symbol P may not be used in their picture-strings).
6. No data-name may name a level 88 entry.
7. The **DELIMITER IN** phrase and the **COUNT IN** phrase may be specified only if the DELIMITED BY phrase is specified.

General rules

1. All references to data-name-2, literal-1, data-name-4, data-name-5, and data-name-6, apply equally to data-name-3, literal-2, data-name-7, data-name-8, and data-name-9, respectively, and all recursions thereof.
2. **Data-name-1** represents the sending area.
3. **Data-name-4** represents the data receiving area. **Data-name-5** represents the receiving area for delimiters.
4. **Literal-1** or the data item referenced by **data-name-2** specifies a delimiter.
5. **Data-name-6** represents the count of the number of characters within data-name-1, isolated by the delimiters for the move to data-name-4. This value does not include a count of the delimiter character(s).

6. The data item referenced by **data-name-10** contains a value which indicates a relative character position within the area defined by data-name-1.
7. The data item referenced by **data-name-11** is a counter which records the number of data items acted upon during the execution of an UNSTRING statement.
8. When a figurative constant is used as the delimiter, it stands for a single character, nonnumeric literal.
9. When the ALL phrase is specified, one occurrence (or two or more contiguous occurrences) of literal-1 (figurative constant or not), or the contents of the data item referenced by data-name-2, are treated as if it were only one occurrence. This occurrence is moved to the receiving data item according to the rules for DELIMITER IN phrase in General Rule 14 below.
10. When an examination encounters two contiguous delimiters, the current receiving area is either space or zero filled according to the description of the receiving area.
11. Literal-1, or the contents of the data item referenced by data-name-2, can contain any character in the computer's character set.
12. Each literal-1 or the data item referenced by data-name-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions of the sending item and in the order given to be recoginized as a delimiter.
13. When two or more delimiters are specified in the DELIMITED BY phrase, an OR condition exists between them. Each delimiter is compared to the sending field. If a match occurs, the character(s) in the sending field is considered to be a single delimiter. No character(s) in the sending field can be considered as part of more than one delimiter. Each delimiter is applied to the sending field in the sequence specified in the UNSTRING statement.
14. When the UNSTRING statement is initiated, the current receiving area is the data item referenced by data-name-4. Data is transferred from data-name-1 to data-name-4 according to the following rules:
    - If the **POINTER** phrase is specified, the string of characters referenced by data-name-1 is examined beginning with the relative character position indicated by the contents of data-name-10. If the POINTER phrase is not specified, the string of characters is examined beginning with the left-most character position.
    - If the **DELIMITED BY** phrase is specified, the examination proceeds, left to right, until either a delimiter specified by the value of literal-1 or the data item referenced by data-name-2 is encountered. (See General Rule 12.) If the DELIMITED BY phrase is not specified, the number of characters examined is equal to the size of the current receiving area. However, if the sign of the receiving item is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area.
    - If the end of the data item referenced by data-name-1 is encountered before the delimiting condition is met, the examination terminates with the last character examined.

- The characters thus examined (excluding the delimiting character(s), if any) are treated as an elementary alphanumeric data item, and are moved into the current receiving area according to the rules for the MOVE statement.
- If the DELIMITER IN phrase is specified, the delimiting character(s) are treated as an elementary alphanumeric data item and are moved into the data item referenced by data-name-5 according to the rules for the MOVE statement. If the delimiting condition is the end of the data item referenced by data-name-1, then the data-name-5 is space filled.
- If the COUNT IN phrase is specified, a value equal to the number of characters thus examined (excluding the delimiter character(s), if any) is moved into the area referenced by data-name-6 according to the rules for an elementary move.
- If the DELIMITED BY phrase is specified, the string of characters is further examined, beginning with the first character to the right of the delimiter. If the DELIMITED BY phrase is not specified, the string of characters is further examined, beginning with the character to the right of the last character transferred.
- After data is transferred to data-name-4, the current receiving area is data-name-7. The behavior described in the preceding four paragraphs is repeated until either all the characters are exhausted in the data item referenced by data-name-1, or until there are no more receiving areas.

15. The initialization of the contents of the data items associated with the POINTER phrase or the TALLYING phrase is the responsibility of the user.
16. The contents of the data item referenced by data-name-10 will be incremented by one for each character examined in the data item referenced by data-name-1. When the execution of an UNSTRING statement with a pointer phrase is completed, data-name-10 will contain a value equal to the initial value, plus the number of characters examined in the data item referenced by data-name-1.
17. When the execution of an UNSTRING statement with a TALLYING phrase is completed, the contents of the data-name-11 will be a value equal to its initial value, plus the number of data receiving items acted upon.
18. Either of the following situations causes an overflow condition:
    - An UNSTRING is initiated, and the value in the data item referenced by data-name-10 is less than one or greater than the size of the data item referenced by data-name-1.
    - If, during execution of an UNSTRING statement, all data receiving areas have been acted upon, and the data item referenced by data-name-1 contains characters which have not been examined.

19. When an overflow condition exists, the UNSTRING operaion is terminated. If an **ON OVERFLOW** phrase has been specified, the imperative-statement is executed. If the ON OVERFLOW phrase is not specified, control is transferred to the next executable statement.
20. The evaluation of subscripting and indexing for the data-names is as follows:
    - Any subscripting or indexing associated with data-name-1, data-name-10, data-name-11 is evaluated only once, immediately before any data is transferred as the result of the execution of the UNSTRING statement.
    - Any subscripting or indexing associated with data-name-2 through data-name-6 is evaluated immediately before the transfer of data into the respective data item.
21. Up to five delimiters may be specified. If more than five are specified, the compiler will abort with error code 113.

Note

Binary counter must not be used with the UNSTRING statement.

Example

```
ID DIVISION.
PROGRAM-ID. UNSTRING.
ENVIRONMENT DIVISION.
SOURCE-COMPUTER. PR1ME17-1.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 ID-SEND PIC X(17) VALUE '123  45678**90ABC'.
77 D-LIMITER PIC X VALUE '*'.
77 POINTR PIC 99 VALUE Ø1.
77 TALLY  PIC 99 VALUE ZEROES.
Ø1 FIELDS.
    Ø2 FIELD-1 PIC X(6).
    Ø2 FIELD-2 PIC X(6).
    Ø2 FIELD-3 PIC X(3).
    Ø2 FIELD-4 PIC X(5).
Ø1 DELIMS.
    Ø2 DELIM-1 PIC X VALUE IS SPACE.
    Ø2 DELIM-2 PIC X VALUE IS SPACE.
    Ø2 DELIM-3 PIC X VALUE IS SPACE.
Ø1 COUNTS.
    Ø2 COUNT-1 PIC 9 VALUE IS ZERO.
    Ø2 COUNT-3 PIC 9 VALUE IS ZERO.
    Ø2 COUNT-4 PIC 9 VALUE IS ZERO.
PROCEDURE DIVISION.
MAIN-PARA.
    UNSTRING ID-SEND DELIMITED BY D-LIMITER OR ALL ' '
        INTO FIELD-1 DELIMITER IN DELIM-1 COUNT IN COUNT-1
            FIELD-2 DELIMITER IN DELIM-2
            FIELD-3 DELIMITER IN DELIM-3 COUNT IN COUNT-3
            FIELD-4 COUNT IN COUNT-4
```

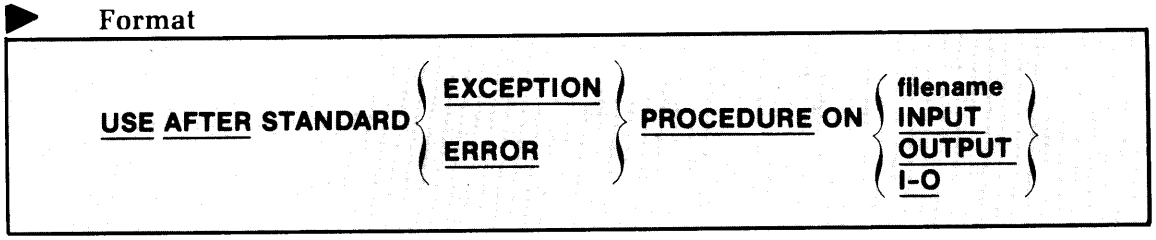
```
            WITH POINTER POINTR
            TALLYING IN TALLY
            OVERFLOW GO TO O-FLOW-PARA.
        GO TO DISPLAY-PARA.
O-FLOW-PARA.
        DISPLAY
            'OVERFLOW ENCOUNTERED,DISPLAY OF VARIABLES FOLLOWS:'.
DISPLAY-PARA.
        DISPLAY 'FIELD 1=' FIELD-1.
        DISPLAY 'FIELD 2=' FIELD-2.
        EXHIBIT NAMED FIELD-3.
        EXHIBIT NAMED FIELD-4.
        EXHIBIT NAMED DELIM-1.
        DISPLAY 'DELIM 2=' DELIM-2.
        DISPLAY 'DELIM 3=' DELIM-3.
        DISPLAY 'COUNT 1=' COUNT-1.
        DISPLAY 'COUNT 3=' COUNT-3.
        DISPLAY 'COUNT 4=' COUNT-4.
        EXHIBIT NAMED POINTR.
        EXHIBIT NAMED TALLY.
        STOP RUN.
```

USE

Function

The USE statement specifies procedures for input/output error handling which are in addition to the standard procedures provided by the input/output control system.

Format



Syntax rules

1. A USE statement, when present, must immediately follow a section header in the Declaratives section, followed by a period and a space. The remainder of the section must consist of zero, one, or more procedural paragraphs which define the procedures to be used.
   Example:

```
PROCEDURE DIVISION.

DECLARATIVES.

{section-name SECTION.  USE sentence.

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

2. The USE statement itself is never executed; rather, it defines the conditions for the execution of the USE procedures.

3. A given file-name may not be associated with more than one DECLARATIVES section.
4. The words **EXCEPTION** and **ERROR** are interchangeable.
5. The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

General rules

1. The DECLARATIVES section is executed (by the PERFORM mechanism) after the standard I/0 recovery procedures for the files designated, or after the invalid key condition arises on a statement lacking the INVALID KEY clause.
2. After execution of a USE procedure, control is returned to the invoking routine.
3. Within a USE procedure, there must be no reference to any non-declarative procedures. Conversely, in the nondeclarative portion, there must be no reference to procedure-names which appear in the declarative portion, except that PERFORM statements may refer to the procedures associated with such a USE statement.
4. Within a USE procedure, no statement may be executed which would result in the execution of a USE procedure previously invoked but not completed (that is, a USE procedure, which through previously invoked, had not yet returned control to the invoking routine).

WRITE

Function

The WRITE statement releases a logical record for an output or I/O file.It can also be used for vertical positioning of lines within a logical page.

Format one

```
WRITE record-name [FROM data-name-1]
    [ { AFTER  } ADVANCING { { { data-name-2 } [ LINE  ] } }
      { BEFORE }           { { { integer     } [ LINES ] } }
                           { { { PAGE        }           } } ]
```

Format two

```
WRITE record-name [FROM data-name-1]

    [INVALID KEY imperative-statement]
```

Syntax rules

1. Format one can only be used for sequential files.
2. Format two can only be used for Relative I/O and Indexed I/O files.
3. **Record-name** and **data-name** must not refer to the same storage area.
4. Record-name is the 01 level record-name of a logical record, described in a record description entry in the File Section of the Data Division.

General rules

1. For both WRITE statement formats, the associated file must be open as OUTPUT or I/O.
2. In Format one if the **FROM** option is taken, the information is moved to the record area prior to the WRITE. If the data being moved is longer than the receiving field, the data is truncated to the size of the receiving field. If the receiving field is longer than the data, the remaining area is filled with spaces.
3. In Format one if the **ADVANCING** option is taken, print control spacing is indicated. The first position in the record must be reserved as FILLER for the print control character being generated.
   - If the **BEFORE** option is taken, a line is written before advancing.
   - If the **AFTER** option is taken, spacing occurs, and then the line is written.
   - **Data-name-2 LINE(s)** is the number of spacing lines required between data lines. data-name-2 may be 0 to 62.
   - **PAGE** skips to a new page, then a line is written.

   If the ADVANCING option is not taken, the default is one line.
4. In Format one, the value of integer is as described below.

Integer	Carriage Control Actions
0	Overprinting
1	Single spacing
2	Double spacing
3	Triple spacing
4	4-line spacing
5	5-line spacing
6	6-line spacing
.	
.	
.	
62	62-line spacing
PAGE	Skips to top of new page

5. In Format two for Relative I/O files: prior to a WRITE statement, a valid unique value must be in the primary RECORD KEY data-name. If the FROM option is used, the unique value in RECORD KEY data-name must be in the relative location of data-name-1. If the primary key is not unique, the invalid statement or the DECLARATIVES section will be executed. Refer to C-4 in Appendix C for Error Conditions.
6. In Format two for Indexed I/O files: the INVALID KEY clause must be specified if the DECLARATIVE section is not applicable. The program will terminate if an error code condition arises.
   - **For sequential access**: If a file is opened as OUTPUT, records are placed in the file in sequential order. The first record would have a position of 1, and the record number returned into the RELATIVE KEY data-name would be 1, etc.

- **For dynamic and random access**: The value of the record number must be placed in the RELATIVE KEY data-name-1.

Example

```
PROCEDURE DIVISION.
REQUIRED-PARA.
     DISPLAY 'ENTER 1 TO CREATE NEW FILE'.
     DISPLAY 'ENTER 2 TO UPDATE OLD FILE'.
     ACCEPT CREATE-UPDATE.
     IF CREATE-UPDATE = '2'
           OPEN OUTPUT PRINT-FILE
           GO TO UPDATE-ONLY.
CREATE-FILE.
     MOVE SPACES TO WS-RECORD.
     OPEN INPUT CARD-FILE,
          OUTPUT PRINT-FILE, DIRECTORY-FILE.
     WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
READ-NEXT.
     READ CARD-FILE AT END GO TO LIST-DIRECTORY.
     WRITE PRINT-LINE FROM CARD-IMAGE.
     MOVE SPACES TO DISPLAY-RECORD.
     MOVE CORR CARD-RECORD TO DIRECTORY-RECORD-INPUT.
     WRITE DIRECTORY-RECORD-OUTPUT
           INVALID KEY DISPLAY 'FILE STATUS = ' FILE-STATUS.
     GO TO READ-NEXT.
LIST-DIRECTORY.
     CLOSE CARD-FILE, DIRECTORY-FILE.
     MOVE ' NEWLY CREATED FILE' TO PRINT-LINE.
     WRITE PRINT-LINE AFTER ADVANCING 3 LINES.
UPDATE-ONLY.
     MOVE SPACES TO PRINT-LINE.
     DISPLAY 'END TEST ONE'.
     OPEN I-O DIRECTORY-FILE.
     IF CREATE-UPDATE = '2'
          GO TO GET-NEXT-INQUIRY.
     CLOSE DIRECTORY-FILE, PRINT-FILE.
     GO TO REQUIRED-PARA.
LIST-DIR.
     MOVE LOW-VALUES TO PHONE-NUMBER, LAST-NAME,  BIRTH-DATE,
                        STATE, FIRST-NAME.
LIST.
     MOVE LOW-VALUES TO PHONE-NUMBER, AT-END-SWITCH.
     START DIRECTORY-FILE KEY IS NOT LESS THAN PHONE-NUMBER.
     WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
     GO TO READ-NEXT-DIRECTORY-RECORD.
LIST1.
     MOVE LOW-VALUES TO LAST-NAME, AT-END-SWITCH.
     START DIRECTORY-FILE KEY IS NOT LESS THAN LAST-NAME.
     WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
     GO TO READ-NEXT-DIRECTORY-RECORD.
```

```
 LIST2.
      MOVE LOW-VALUES TO STATE, AT-END-SWITCH.
      START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
      WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
      GO TO READ-NEXT-DIRECTORY-RECORD.
 LIST3.
      MOVE LOW-VALUES TO BIRTH-DATE, AT-END-SWITCH.
      START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTH-DATE.
      WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
      GO TO READ-NEXT-DIRECTORY-RECORD.
 LIST4.
      MOVE LOW-VALUES TO FIRST-NAME, AT-END-SWITCH.
      START DIRECTORY-FILE KEY IS NOT LESS THAN FIRST-NAME.
      WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
 READ-NEXT-DIRECTORY-RECORD.
      READ DIRECTORY-FILE NEXT RECORD AT END
         MOVE 1 TO AT-END-SWITCH.
      MOVE DIRECTORY-RECORD-OUTPUT TO PRINT-LINE.
      WRITE PRINT-LINE.
      GO TO READ-NEXT-DIRECTORY-RECORD.
 LIST-DONE.
      EXIT.
*
 GET-NEXT-INQUIRY.
      DISPLAY 'ENTER TRANSACTION TYPE'.
      DISPLAY ' # = READ FILE SEQ'.
      DISPLAY ' + = ADD'.
      DISPLAY ' - = DELETE'.
      DISPLAY ' / = CHANGE'.
      DISPLAY ' * = QUIT'.
      ACCEPT ACCEPT-TRANSACTION-TYPE FROM TTY.
      IF ACCEPT-TRANSACTION-TYPE = '+' GO TO ADDITION.
      IF ACCEPT-TRANSACTION-TYPE = '-' GO TO DELETION.
      IF ACCEPT-TRANSACTION-TYPE = '/', GO TO CHANGE.
      IF ACCEPT-TRANSACTION-TYPE = '*', PERFORM WRAPUP, STOP RUN.
      IF ACCEPT-TRANSACTION-TYPE = '#', GO TO READ-FILE.
      DISPLAY 'INVALID TRANSACTION TYPE = '
+               ACCEPT-TRANSACTION-TYPE.
      DISPLAY 'TRY AGAIN'.
      GO TO GET-NEXT-INQUIRY.
 NO-SUCH-NAME.
      DISPLAY ' NO SUCH RECORD = ' DISPLAY-DIR.
      GO TO GET-NEXT-INQUIRY.
*
 ADDITION.
      DISPLAY 'ENTER DATA RECORD FOR ADD'.
      PERFORM FORMAT-INPUT.
      PERFORM  MOVE-REC.
      WRITE DIRECTORY-RECORD-OUTPUT INVALID KEY
              DISPLAY FILE-STATUS
              DISPLAY DISPLAY-DIR.
      GO TO GET-NEXT-INQUIRY.
```

```
*
 DELETION.
      DISPLAY 'ENTER LAST NAME OF ENTRY TO BE DELETED'.
      ACCEPT LAST-NAME FROM TTY.
      READ DIRECTORY-FILE KEY IS LAST-NAME
              INVALID KEY GO TO NO-SUCH-NAME.
      DELETE DIRECTORY-FILE RECORD INVALID KEY
         GO TO NO-SUCH-NAME.
      GO TO GET-NEXT-INQUIRY.
*
 CHANGE.
      DISPLAY 'ENTER KEY TO BE CHANGED'.
      DISPLAY 'LAST-NAME  = 1'.
      DISPLAY 'STATE      = 2'.
      DISPLAY 'BIRTH-DATE = 3'.
      DISPLAY 'FIRST-NAME = 4'.
      ACCEPT GO-TO-NAME.
      GO TO READ-ALT1 READ-ALT2 READ-ALT3 READ-ALT4
              DEPENDING ON GO-TO-NAME.
      DISPLAY 'WRONG TYPE ENTERED TRY AGAIN', GO TO CHANGE.
*
 READ-ALT1.
      DISPLAY 'ENTER LAST-NAME'.
      ACCEPT WS-LAST-NAME.
      MOVE SPACES TO DIRECTORY-RECORD-OUTPUT.
      MOVE WS-LAST-NAME TO LAST-NAME.
      READ DIRECTORY-FILE KEY IS LAST-NAME


              INVALID KEY DISPLAY 'LAST-NAME = ' LAST-NAME
              DISPLAY 'FILE STATUS = ' FILE-STATUS
              DISPLAY DISPLAY-DIR
              GO TO GET-NEXT-INQUIRY.
      GO TO CHANGE-RECORD.
*
 READ-ALT2.
      DISPLAY 'ENTER STATE '.
      ACCEPT WS-STATE.
      MOVE SPACES TO DIRECTORY-RECORD-OUTPUT.
      MOVE WS-STATE TO STATE.
      READ DIRECTORY-FILE KEY IS STATE
             INVALID KEY DISPLAY 'STATE = ' STATE
             DISPLAY 'FILE STATUS = ' FILE-STATUS
             DISPLAY DISPLAY-DIR
             GO TO GET-NEXT-INQUIRY.
      GO TO CHANGE-RECORD.
*
 READ-ALT3.
      DISPLAY 'ENTER BIRTH-DATE'.
      ACCEPT WS-BIRTH-DATE.
      MOVE SPACES TO DIRECTORY-RECORD-OUTPUT.
      MOVE WS-BIRTH-DATE TO BIRTH-DATE.
      READ DIRECTORY-FILE KEY IS BIRTH-DATE
              INVALID KEY DISPLAY 'BIRTH-DATE = ' BIRTH-DATE
```

```
                DISPLAY 'FILE STATUS = ' FILE-STATUS
                DISPLAY DISPLAY-DIR
                GO TO GET-NEXT-INQUIRY.
           GO TO CHANGE-RECORD.
*
 READ-ALT4.
           DISPLAY 'ENTER FIRST-NAME'.
           ACCEPT WS-FIRST-NAME.
           MOVE SPACES TO DIRECTORY-RECORD-OUTPUT.
           MOVE WS-FIRST-NAME TO FIRST-NAME.
           READ DIRECTORY-FILE KEY IS FIRST-NAME
                   INVALID KEY DISPLAY 'FIRST-NAME = ' FIRST-NAME
                   DISPLAY 'FILE STATUS = ' FILE-STATUS
                   DISPLAY DISPLAY-DIR
                   GO TO GET-NEXT-INQUIRY.
*
 CHANGE-RECORD.
           DISPLAY DISPLAY-DIR.
           PERFORM FORMAT-INPUT.
*
 MOVE-REC.
           IF WS-RECORD = SPACES
                DISPLAY 'NO DATA ENTERED TRY AGAIN'
                GO TO GET-NEXT-INQUIRY.
           IF WS-LAST-NAME NOT = SPACES
                MOVE WS-LAST-NAME TO LAST-NAME.
           IF WS-FIRST-NAME NOT = SPACES
                MOVE WS-FIRST-NAME TO FIRST-NAME.
           IF WS-ADDRESS NOT = SPACES
                MOVE WS-ADDRESS TO ADDRESS.
           IF WS-CITY NOT = SPACES
                MOVE WS-CITY TO CITY.
           IF WS-PHONE-NUMBER NOT = SPACES
                MOVE WS-PHONE-NUMBER TO PHONE-NUMBER.
           IF WS-STATE NOT = SPACES
                MOVE WS-STATE TO STATE.
           IF WS-BIRTH-DATE NOT = SPACES
                MOVE WS-BIRTH-DATE TO BIRTH-DATE.
 MOVE-NEXT.
           EXIT.
*
 REWRITE-RECORD.
           REWRITE DIRECTORY-RECORD-OUTPUT INVALID KEY,
                   GO TO NO-SUCH-NAME.
           GO TO GET-NEXT-INQUIRY.
*
 READ-FILE.
           MOVE ZEROS TO PERFORM-COUNT.
           DISPLAY 'ENTER NUMBER OF RECORDS TO BE READ'.
           ACCEPT PERFORM-COUNT.
           IF PERFORM-COUNT = ZEROS
                DISPLAY 'NO RECORD COUNT ENTERED'
```

```
               GO TO GET-NEXT-INQUIRY.
          IF PERFORM-COUNT1 NOT NUMERIC
             PERFORM RIGHT-JUSTIFY.
      *
       READ-TYPE.
          DISPLAY 'ENTER SECONDARY KEY TO BE READ'.
          DISPLAY 'PHONE-NUMBER   = 1'.
          DISPLAY 'LAST-NAME  = 2'.
          DISPLAY 'STATE      = 3'.
          DISPLAY 'BIRTH-DATE = 4'.
          DISPLAY 'FIRST-NAME = 5'.
          ACCEPT GO-TO-READ.
          IF GO-TO-READ IS LESS THAN 1 OR GO-TO-READ IS GREATER THAN
5
               DISPLAY 'INVALID SECONDARY KEY TRY AGAIN'
               GO TO READ-TYPE.
          IF GO-TO-READ = 1 PERFORM READ-1
             ELSE IF GO-TO-READ = 2 PERFORM READ-2
                 ELSE IF GO-TO-READ = 3 PERFORM READ-3
                    ELSE IF GO-TO-READ = 4 PERFORM READ-4
                         ELSE IF GO-TO-READ = 5 PERFORM READ-5.
          PERFORM READ-FILE-GO THROUGH READ-FILE-EXIT.
      *
       READ-1.
          MOVE LOW-VALUES TO PHONE-NUMBER.
          START DIRECTORY-FILE KEY IS NOT LESS THAN PHONE-NUMBER.
       READ-2.
          MOVE LOW-VALUES TO LAST-NAME.
          START DIRECTORY-FILE KEY IS NOT LESS THAN LAST-NAME.
       READ-3.
          MOVE LOW-VALUES TO STATE.
          START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
       READ-4.
          MOVE ZEROS TO BIRTH-DATE.
          START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTH-DATE.
       READ-5.
          MOVE LOW-VALUES TO FIRST-NAME.
          START DIRECTORY-FILE KEY IS NOT LESS THAN FIRST-NAME.
       READ-FILE-GO.
          IF PERFORM-COUNT = Ø GO TO READ-FILE-EXIT.
          READ DIRECTORY-FILE NEXT RECORD
               AT END MOVE ZEROS TO PERFORM-COUNT
               GO TO READ-FILE-EXIT.
          DISPLAY DISPLAY-DIR.
          SUBTRACT 1 FROM PERFORM-COUNT.
          GO TO READ-FILE-GO.
       READ-FILE-EXIT.
          GO TO GET-NEXT-INQUIRY.
      *
       WRAPUP.
          PERFORM LIST-DIR.
```

```
           MOVE 'END OF INDEXED TEST TO CHANGE FILE' TO PRINT-LINE.
           DISPLAY 'END OF INDEXED TEST'.
           CLOSE PRINT-FILE, DIRECTORY-FILE.
*
 FORMAT-INPUT.
           MOVE SPACES TO WS-RECORD.
           DISPLAY 'ENTER LAST NAME'.
           ACCEPT WS-LAST-NAME.
           DISPLAY 'ENTER FIRST NAME'.
           ACCEPT WS-FIRST-NAME.
           DISPLAY 'ENTER ADDRESS'.
           ACCEPT WS-ADDRESS.
           DISPLAY 'ENTER CITY'.
           ACCEPT WS-CITY.
           DISPLAY 'ENTER PHONE NUMBER'.
           ACCEPT WS-PHONE-NUMBER.
           DISPLAY 'ENTER STATE XX'.
           ACCEPT WS-STATE.
           DISPLAY 'ENTER BIRTH-DATE MMDDYY'.
           ACCEPT WS-BIRTH-DATE.
*
  RIGHT-JUSTIFY.
           IF PER-CO(1) NUMERIC AND
                PER-CO(2) NOT NUMERIC AND
                PER-CO(3) NOT NUMERIC
                MOVE PER-CO(1) TO PER-CO(3)
                MOVE '0' TO PER-CO(1)  PER-CO(2)
                GO TO READ-TYPE.
           IF PER-CO(1) NUMERIC      AND
                PER-CO(2) NUMERIC    AND
                PER-CO(3) NOT NUMERIC
                MOVE PER-CO(2) TO PER-CO(3)
                MOVE PER-CO(1) TO PER-CO(2)
                MOVE '0' TO PER-CO(1).
```

9

Inter-program communication

DEFINITION

Inter-Program Communication provides a facility by which a program can communicate with one or more other programs. Control may be transferred from one program to another within a run unit, and both programs may have access to the same data items.

Inter-module communication of data is made possible through the use of the LINKAGE SECTION of the Data Division, and by the CALL statement and USING list appendage to the Procedure Division header of a subprogram module.

LINKAGE SECTION

The LINKAGE SECTION in a program is meaningful if, and only if, the called program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

The LINKAGE SECTION describes data made available in memory from another program module, but which is to be referred to in both the calling and the called program.

No space is allocated in the program for data items referenced by data-names in the Linkage Section of that program. Procedure Division references to such items are resolved at load time, equating the references in the called program to the location used in the calling program by passing address parameters. Thus, Record Description entries in the LINKAGE SECTION provide data-names by which data-areas reserved in memory by other programs may be referenced.

Data items defined in the LINKAGE SECTION of the called program may be referenced in the Procedure Division of that called program only if: they are specified as operands of the USING phrase of the Procedure Division header or are subordinate to such operands, and the called program is under the control of a CALL statement which specifies a USING phrase (see the example at the close of this section).

The structure of the LINKAGE SECTION is that described for the WORKING-STORAGE SECTION.

Any Record Description clause may be used to describe items in the LINKAGE SECTION except:

1. The VALUE clause may only be specified for level 88 items.
2. Data-names used in the LINKAGE SECTION must be unique (may not be qualified).
3. The programmer must ensure proper correspondence between an argument (pointer to data) in a CALL statement and the data-name in a USING list on a subprogram Procedure header. Arguments and data-names must be either level 01 or level 77 items. (See Rule 4 about level 77 entries.)
4. Items in the LINKAGE SECTION which bear no hierarchy relationship to one another need not be grouped into records. These are classified and defined as noncontiguous elementary items. They may be defined in separate level 77 entries.

Such Data Description entries must include a level-number 77, a data-name, and a PICTURE clause or the USAGE IS INDEX clause.

PROCEDURE DIVISION

In addition to LINKAGE SECTION entries, inter-program communication requires certain Procedure Division entries.

Using list appendage to procedure header

The Procedure Division header of a CALLable subprogram is written as:

PROCEDURE DIVISION [USING data-name ...]

where each of the **data-name** operands is an entry in the LINKAGE SECTION of the subprogram, having level 77 or 01. Addresses are passed from an external CALL in one-to-one correspondence to the operands in the **USING** list of the Procedure Division header so that data in the calling program may be manipulated in the subprogram.

CALL

The CALL statement allows one program to communicate with one or more other programs. It causes control to be transferred from one loaded program to another within a run unit.

Format

CALL literal-1 [USING data-name-1 [, data-name-2] ...]

Syntax rules

1. The CALL statement appears in the calling program. The called program, which must be known at compile time, is specified by name as **literal-1.** The program represented by literal-1 may have been written in a source language other than COBOL.
2. Literal-1 must be a non-numeric literal.
3. The **USING** phrase is included in the CALL statement only if there is a USING phrase in the Procedure Division header of the called program. Corresponding USING phrases in the calling and the called programs must have the same number of operands. Up to 14 data-names are allowed.
4. Each operand in the USING phrase must have been defined as a data item in the File Section, Working-Storage Section, or Linkage Section and must have a level-number of 0l or 77. **Data-name-1, data-name-2,** etc., may be qualified when they refer to data items defined in the File Section.

General rules

1. The execution of a CALL statement transfers control to the called program.
2. A program is in its initial state the first time it is called within a run unit. On all other entries into the called program, the state of the program remains the same as when control last past from its EXIT statement back to the calling program. This includes all data fields, the status and positioning of all files, and all alterable switch settings.

3. Called programs can contain CALL statements. However, a called program must not contain a CALL statement that directly or indirectly calls the calling program.
4. The data-names specified by the USING phrase of the CALL statement indicate those data items available to a calling program, that may be referred to in the called program. The order in which the data-names appear in the USING phrases of the two programs is critical; the data-names in the USING phrase of the CALL statement in the calling program are interpreted as corresponding on a one-to-one basis with those in the USING phrase in the Procedure Division header of the called program. Corresponding data-names refer to a single set of data which is available to the called and calling programs. Correspondence is positional, not by name. There is no such correspondence for index-names, however, since index-names in the calling and called programs always refer to separate indexes.

Note

See Section 8, PROCEDURE DIVISION, for additional information.

EXIT PROGRAM

The EXIT PROGRAM statement specifies the logical end of a called program.

Format

```
EXIT PROGRAM.
```

▶ Syntax rule

The EXIT PROGRAM statement may be in a paragraph by itself. However, Prime COBOL does not require it.

▶ General rule

The EXIT PROGRAM statement, appearing in a called subprogram, causes control to be returned to the next executable statement after a CALL in the calling program. See Section 8 for detailed discussion.

ENTER

An ENTER statement is classified as a compiler-directing statement; it acts as a modifier to a subsequent CALL statement.

Format

```
ENTER { COBOL     }
      { ASSEMBLER }
```

▶ Syntax rules

1. A called subprogram may have been written in COBOL, FORTRAN, ASSEMBLER, etc. language. The ENTER statement provides the means to identify the language in which a subprogram is written.

2. **ENTER ASSEMBLER** tells the compiler that the ensuing callee is not a COBOL subprogram.
3. **ENTER COBOL** tells the compiler that the ensuing callee is a COBOL subprogram.It may also be used following a CALL statement. This traditional usage is optional; after any CALL statement, ENTER COBOL is assumed.

Note

Additional information for ENTER statement is presented in Section 8.

Example

Filename = CALLER

```
       ID DIVISION.
       PROGRAM-ID.  CALLER.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       Ø1  WS-ITEMS.
           Ø5  WS-VALUE-1   PIC 99.
           Ø5  WS-VALUE-2   PIC 9(5).
           Ø5  WS-VALUE-3   PIC 9.
           Ø5  WS-VALUE-4   PIC X(6).
           Ø5  WS-VALUE-5   PIC AAA.
       PROCEDURE DIVISION.
       MAIN-PARA.
           MOVE 'ABC' TO WS-VALUE-5.
           MOVE 11111 TO WS-VALUE-2.
           EXHIBIT NAMED WS-VALUE-2.
           EXHIBIT NAMED WS-VALUE-5.
           CALL 'CALLED' USING WS-ITEMS.



           DISPLAY
             'VALUES AFTER CALL STATEMENT IS EXECUTED'.
           EXHIBIT NAMED WS-VALUE-2.
           EXHIBIT NAMED WS-VALUE-5.
```

Filename = CALLED

```
       ID DIVISION.
       PROGRAM-ID.  CALLED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       Ø1 WS-TEST  PIC 9(5) VALUE 22222.
       LINKAGE SECTION.
```

```
Ø1  WS-ITEM.
    Ø5  WS-VALUE-1  PIC 99.
    Ø5  WS-VALUE-2  PIC 9(5).
    Ø5  WS-VALUE-3  PIC X(7).
    Ø5  ANY-NAME    PIC AAA.
PROCEDURE DIVISION USING WS-ITEM.
MAIN-PARA.
    MOVE WS-TEST TO WS-VALUE-2.
    MOVE 'DEF' TO ANY-NAME.
EXIT-PARA.
    EXIT PROGRAM.
OPTIONAL-STOP-PARA.
    STOP RUN.
```

10
Table handling

DEFINITION

Table Handling provides a capability for defining tables of contiguous data items and accessing those items relative to their position in the table. The OCCURS clause is the language facility provided for specifying how many times an item is to be repeated. Each item may be identified through use of a subscript or an index.

DATA DIVISION

The maximum table size permissible in this compiler is 32,767 bytes.

OCCURS

The OCCURS clause eliminates the need for separate entries for repeated data items. Furthur, it supplies information required for the application of subscripts or indexes.

Format

```
OCCURS integer-1 TIMES
[ { ASCENDING  } KEY IS data-name-1 [, data-name-2] ... ]
  { DESCENDING }
[INDEXED BY index-name-1 [, index-name-2] ... ]
```

Syntax rules

1. The OCCURS clause must not be used in a data description entry having a level number 01, 66, 77, or 88.
2. The maximum OCCURS specification (integer-1) is 32,766.
3. The minimum OCCURS specification (integer-1) is 2.
4. Data-name-1 must either be the name of the subject entry containing the OCCURS clause, or the name of an entry subordinate to the subject entry.
5. Data-name-2, etc., must be the name of an entry subordinate to the group item which is the subject of this entry.
6. Data-names in the KEY IS phrase must not contain an OCCURS clause except where data-name-1 is the subject of the entry.
7. There must not be any entry that contains an OCCURS clause between the data-names in the KEY IS phrase and the subject of the entry, except where data-name-1 is the subject of the entry.

8. All data-names used in the OCCURS clause may be qualified; however, they must not be subscripted or indexed.
9. An INDEXED BY phrase is required if the subject of this entry, or an entry subordinates to this entry, is to be referenced by indexing. The index-names identified by this phrase are not defined elsewhere, since their allocation and format are dependent on the hardware (system); not representing data, the index-names cannot be associated with any data hierarchy.
10. Each index-name must be an unique word within the program.

General rules

1. The OCCURS clause defines tables and other homogenous sets of repeated data items. When the OCCURS clause is used, the data-name which is the subject of the entry must be referred to by subscripting or indexing.
2. Except for the OCCURS clause itself, all data description clauses associated with an item containing an OCCURS clause apply to each occurrence of the item being described.
3. The value of integer-1 represents the exact number of occurrences of the subject entry.
4. The KEY IS phrase indicates that the repeated data is arranged in ascending or descending order according to the values contained in data-name-1, data-name-2, etc. The ascending or descending order is determined by the rules for the comparison of operands. (See Section 4, Numeric Comparisons and Non-Numeric Comparisons.) The data-names are listed in their descending order of significance.
5. When the INDEXED BY phrase is omitted, subscripting is used to indicate an individual element within a list, or within a table of like elements which do not have individual data-names.
6. When the INDEXED BY phrase is used, an index is assigned to a table of like elements, with individual items in the table being identified by index-name. For example:

```
          .
          .
          .
    05  MON-TAB OCCURS 12 TIMES INDEXED BY INDX
        ASCENDING KEY MONTH-NO.
        10  MONTH-NO      PIC 99.
        10  MONTH-VALUE   PIC XXX.
          .
          .
          .

FIND-MONTH.
    SEARCH ALL MON-TAB
       WHEN MONTH-NO(INDX) = MONTH-ACCEPT
       MOVE MONTH-VALUE(INDX)   TO PRINT-MONTH.
          .
          .
          .
```

Table initialization

Table initialization, if required, may be achieved either in the Working-Storage Section (explained below) or in the Procedure Division by using appropriate MOVE statements.

In the Working-Storage Section of the Data Division, tables can be initialized in one of two ways:

- If the elements in a table do not need to be individually initialized, then the VALUE clause is specified in the data description entry containing the table name. The subordinate data description entry will then be given an OCCURS clause defining the structure of the table.

  Examples:

```
01  A-TABLE  VALUE ZEROS.
    05  B-TABLE  PIC X(3)  OCCURS 100 TIMES.


01  STATE-TABLE  VALUE 'CALAMAPAVA'.
    05  STATE  PIC XX  OCCURS 5 TIMES.
```

- If the elements in a table need to be individually initialized, then a VALUE clause is specified in each table element entry. The table will then be redefined by using the REDEFINES entry with the subordinate entry containing an OCCURS clause.

  Example:

```
01  WAREHOUSE.
    05  FILLER  PIC 99      VALUE 10.
    05  FILLER  PIC X(22)   VALUE 'BOSTON DISTRICT BRANCH'.
    05  FILLER  PIC 99      VALUE 11.
    05  FILLER  PIC X(22)   VALUE 'NEW YORK CITY BRANCH  '.
    05  FILLER  PIC 99      VALUE 12.
    05  FILLER  PIC X(22)   VALUE 'HOUSTON HOME OFFICE   '.
01  WARE-HOUSE REDEFINES WAREHOUSE.
    05  HOUSES   OCCURS 3 TIMES.
        10  HOUSE-NO      PIC 99.
        10  HOUSE-NAME    PIC X(22).
```

Note

The VALUE clause is not permitted in a data description entry specifying an OCCURS or REDEFINES clause, or in any entry subordinate to one specifying an OCCURS or REDEFINES clause.

Indexing and subscripting

Indexing and subscripting are the two methods of accessing the individual elements in a table established by the OCCURS clause. To specify a desired individual table element, follow the table element's data-name by a parenthesized index or subscript.

An index is an index-name coded in an INDEXED BY phrase in an OCCURS clause. The value of an index corresponds to the occurrence number of the desired element.

A subscript is an integer appended in parenthesis to a data-name. The subscript value represents the occurrence of the desired element.

INDEXED BY phrase: The positioning of the INDEXED BY phrase appears in the OCCURS clause format. As indicated, the INDEXED BY phrase is appended to the OCCURS clause. The INDEXED BY phrase is required if the subject of an entry, or one subordinates to that entry, is to be referred to by indexing. The index-name identified by this phrase is not defined elsewhere; allocation and format are defined by the compiler. In other words, an index-name is declared not by the usual method of level-number, name and Data Description clauses, but implicitly by appearance in the "INDEXED BY index-name" appendage to an OCCURS clause.

The format of the INDEXED BY phrase is:

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

Index-name is equivalent to an index-item; it must be uniquely named. This compiler assigns a full word for each index-name defined.

An index-item may only be referred to by a SET statement, a SEARCH statement, a CALL statement USING list, a Procedure header USING list, as the variation item in PERFORM VARYING and PERFORM UNTIL, or in a relational condition. In all cases, the process is equivalent to dealing with a binary word integer subscript. A maximum of three indexes may be used on any given data-name.

Direct indexing: Direct indexing is specified by using an index-name in the form of a subscript, for example, ELEMENT(INDX-1).

Consider the following illustration:

```
      .
      .
      .
01  TABLE-A.
    05  ELEMENT OCCURS 6 TIMES INDEXED BY INDX-1.
      .
      .
      .
    SET INDX-1 TO 4.
    MOVE ELEMENT(INDX-1) TO PRINT-FIELD.
      .
      .
      .
```

ELEMENT(INDX-1) in the example above would refer to the fourth element of the table. The MOVE statement would move the contents of the ELEMENT to a field called PRINT-FIELD.

Relative indexing: Relative indexing may be specified wherever indexing can be specified. Using the sample TABLE-A defined in the example above, the same results could be achieved with relative indexing; namely,

```
MOVE ELEMENT(INDX-1 + 3) TO PRINT-FIELD.
```

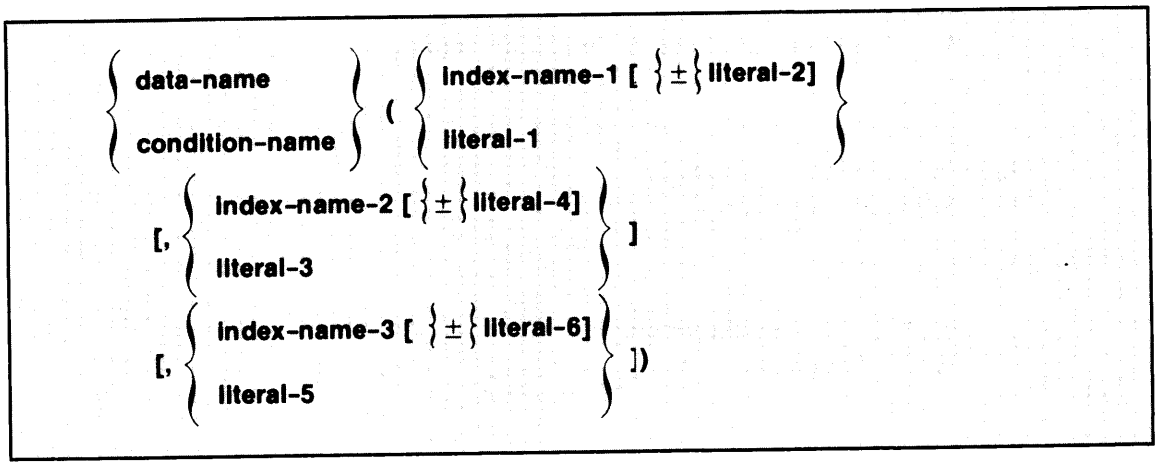
will move the contents of the fourth ELEMENT to a field named PRINT-FIELD (assuming that INDX-1 has a value of 1).

In the instance above, index-name is followed by a space, followed by one of the operators + or –, followed by another space, followed by an unsigned, integer numeric literal, all delimited by the balanced pair of separators left parenthesis and right parenhesis.

The occurrence number resulting from relative indexing is determined by incrementing or decrementing the index by the value of the literal.

When a statement is executed which refers to an indexed table element, the value in the associated index must neither be less than one, nor greater than the highest occurence number of an element in the table. This restriction applies equally to direct indexing and relative indexing.

The general format for direct indexing and relative indexing is:



Subscripting: Subscripting may be used in lieu of indexing. In such instances, the INDEXED BY phrase is omitted.

The format for subscripting is:

data-name (subscript-1 [, subscript-2 [, subscript-3]])

The subscript can be represented either by a positive numeric literal or by a data-name. The data-name must be a numeric elementary item which represents an integer. Further, the data-name as subscript may be qualified but not itself subscripted.

The subscript data-name may be signed, but the value must be positive. The subscript value indicates the position of the item in a table. The lowest value permitted is one, indicating the first position in the table. Subsequent positions are indicated by sequential values 2, 3, 4, etc., up to the highest permissible value, which is the maximum number of occurrences of the item specified in the OCCURS clause.

The subscript can be used on any table. For example:

```
Ø1  ARRAY.
    Ø5  ELEMENT, OCCURS 3, PICTURE S9(4), SIGN TRAILING SEPARATE.
```

The coding in the example above would cause the allocation of storage as shown below:

```
ELEMENT(1)      ARRAY consisting of fifteen
                characters; each item has 4
ELEMENT(2)      digits and a separate sign.

ELEMENT(3)
```

- For literal subscripting, the following MOVE statement could be written:

```
MOVE ELEMENT(2) TO QUANTITY.
```

  This would result in moving the contents of the second ELEMENT in ARRAY (previous example) to a field named QUANTITY.

- For data-name subscripting, additional data description entries are required; an example is illustrated below:

```
      .
      .
      .
Ø1    ARRAY.
      Ø5  ELEMENT, OCCURS 3, PICTURE X(4).
      .
      .
      .
Ø1    SUBSCRIPTNO PIC 99.
Ø1    PART-NO PIC X(4).
      .
      .
      .
      MOVE 2 TO SUBSCRIPTNO.
      GO TO TABLERUN.
      .
      .
      .
TABLERUN.
      MOVE ELEMENT(SUBSCRIPTNO) TO PART-NO.
```

  The MOVE statements in the example above would results in the data-name subscript, SUBSCRIPTNO, being set to a value of 2, and the contents of the second ELEMENT of ARRAY being moved to the field called PART-NO.

  The data-name may not be subscripted if it is being used for any of the following functions:

  - When it is being used as a subscript
  - When it appears as the defining name of a data description entry
  - When it appears as data-name-2 in a REDEFINES clause

A subscript must be delimited by a pair of parenthesis following the table element data-name. When two or more subscripts are required, they are written in the order of successively less inclusive dimensions of the data organization, and should be separated by commas. A maximum of three levels of subscripting is permitted for any given data item.

A subscript value is changed in the Procedure Division via the MOVE, ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE verbs. The SET verb cannot be used on a subscript data-name.

Multi-dimensional tables

The following example presents Data Division entries for a multi-dimensional table, TABLE-PLUS.

```
01  TABLE-PLUS.
    05  TYPE OCCURS 10 TIMES.
        10  PART-NO  PIC   X(4).
        10  COLOR    PIC   X  OCCURS 10 TIMES.
        10  CONTROL  OCCURS   7  TIMES.
            15  C1  PIC  X.
            15  C2  PIC XX  OCCURS 4 TIMES.
```

When a table has more than one dimension, the data-name of the desired item is followed by a list of subscripts, one for each OCCURS clause to which the item is subordinate.

In such a list, the first subscript applies to the first OCCURS clause to which the item is subordinate. The second subscript applies to the next most encompassing level. The third, and last, subscript applies to the lowest level OCCURS clause being accessed.

Therefore, using the table depicted in the example above, the statement

```
MOVE C2(8, 6, 4) TO TEMP.
```

would MOVE the contents of the fourth occurrence of the field C2, in the sixth repetition of the field CONTROL, in the eighth occurrence of the field TYPE to a field called TEMP.

Similarly, the statement

```
MOVE C2(10, 7, 4) TO TEMP.
```

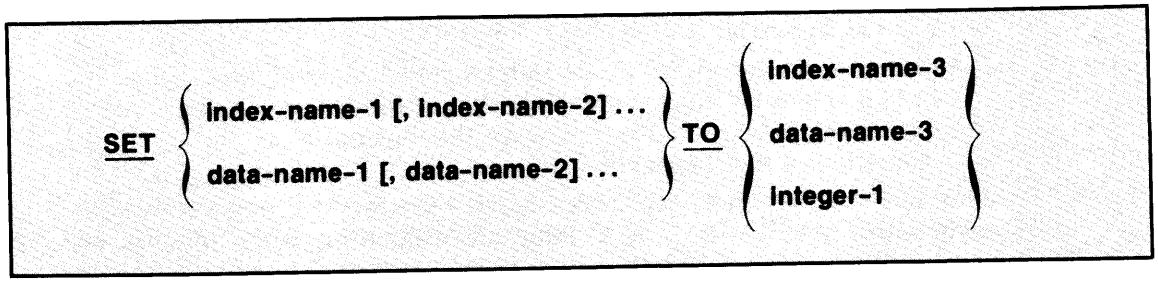
would move the contents of the last occurrence of the field C2 to the field labeled TEMP.

PROCEDURE DIVISION

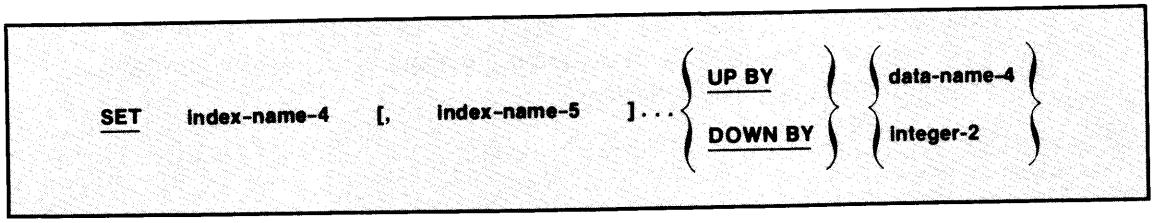
SET

The SET statement permits the manipulation of index-names and index items,for table-handling purposes.

Format one



Format two



Syntax rules

1. All references to index-name-1, data-name-1 and index-name-4 apply equally to index-name-2, data-name-2, and index-name-5, respectively.
2. Data-name-4 must be described as an elementary numeric integer.
3. There must not be a name specifying an index data item after UP BY or DOWN BY option.
4. Integer-1 and integer-2 may be signed. However, integer-1 must have positive value.

General rules

1. In any SET statement, data-names are restricted to binary items, except that a decimal item may follow on the word TO.
2. An index-name should only apply to the OCCURS which defines it.
3. The SET verb cannot be used on a subscripted data-name.
4. Index-names are considered related to a given table and are defined by being specified in the INDEXED BY clause.
5. If inex-name-3 is specified, the value of the index before the execution of th SET statement must not exceed the occurrence number of an element in the associated table.
6. In Format one, the following action occurs:
   - Index-name-1 is set to a value causing it to refer to a table element. That element corresponds in occurrence number to the table element referenced by index-name-3, data-name-3, or integer-1. If data-name-3 is an index data item, or if index-name-3 is related to the same table as index-name-1, no conversion takes place.
   - If data-name-1 is an index data item, it may be set equal to either the contents of index-name-3 or data-name-3, where data-name-3 is also an index data item; no conversion takes place in either case.
   - If data-name-1 is not an index data item, it may be set only to an occurrence number which corresponds to the value of index-name-3. Neither data-name-3 nor integer-1 can be used in this case.
   - The process is repeated for index-name-2, data-name-2, etc., if specified. Each time, the value of index-name-3 or data-name-3 is used as it was at the beginning of the execution of the statement.
7. In Format two, the contents of index-name-4 are incremented (UP BY) or decremented (DOWN BY) by a value corresponding to the number of occurrences represented by the value of integer-2 or data-name-4; thereafter, the process is repeated for index-name-5, etc. Each time the value of data-name-4 is used as it was at the beginning of the execution of the statement.
8. Data in the following table represents the validity of various operand combinations in the SET statement.

Table 10-1. Validity of operand combinations in the SET statement

Sending Item	Receiving Item		
	Integer Data Item	Index-name	Index Data Item
Integer Literal		Valid	
Integer Data Item		Valid	
Index-name	Valid	Valid	Valid*
Index Data Item		Valid*	Valid*

* - No conversion takes place.

SEARCH

The SEARCH statement is used to search a table for a table element which satisfies the specified condition. The associated index-name is adjusted to indicate that table element.

Format one

```
SEARCH identifier-1 [VARYING { identifier-2  } ]
                             { index-name-1  }

[; AT END imperative-statement-1]

                             { imperative-statement-2 }
; WHEN condition-1           { NEXT SENTENCE          }

                             { imperative-statement-3 }
[; WHEN condition-2          { NEXT SENTENCE          } ]
```

Format two

```
SEARCH ALL identifier-1 [; AT END imperative-statement-1 ]

                  { data-name-1      { EQUALS          } { identifier-3 } }
; WHEN            {                  { IS EQUAL TO     } {             } }
                  {                  { IS =            } { literal-1   } }
                  { condition-name-1                                     }

                  { data-name-2      { EQUALS          } { identifier-4 } }
[AND              {                  { IS EQUAL TO     } {             } } ]
                  {                  { IS =            } { literal-2   } }
                  { condition-name-2                                     }

{ imperative-statement-2 }
{ NEXT SENTENCE          }
```

Syntax rules

1. In both Formats one and two, identifier-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause. The description of identifier-1 in Format two must also contain the KEY IS phrase in its OCCURS clause.
2. Identifier-2, when specified, must be described as USAGE IS INDEX, or as a numeric elementary data item without any positions to the right of the assumed decimal point.
3. In Format one, condition-1, condition-2, may be any condition as described under Conditional Expressions in Section 4.
4. In Format two, all referenced condition-names must be defined as having only a single value. The data-name associated with a condition-name must appear in the KEY clause of identifier-1. Each data-name-1, data-name-2 may be qualified. Futher, each data-name-1, data-name-2 must be indexed by the first index-name associated with identifier-1 along with other indices or literals as required.
5. In Format two, when a data-name in the KEY clause of identifier-1 is referenced, or when a condition-name associated with a data-name in the KEY clause of identifier-1 is referenced, all preceding data-names in the KEY clause of identifier-1 or their associated condition-names must also be referenced.

General rules

1. Format one SEARCH statement enables a serial type of search operation, starting with the current index setting.
   - If, at the start of execution of the SEARCH statement, the index-name associated with identifier-1 contains a value which corresponds to an occurrence number greater than the highest permissible occurrence number for identifier-1, the specified imperative-statement-1 is executed; if the AT END phrase is not specified, control passes to the next executable sentence.
   - If, at the start of execution of the SEARCH statement, the index-name associated with identifier-1 contains a value corresponding to an occurrence number not greater than the highest permissible occurrence number for identifier-1, the SEARCH statement operates by evaluating the conditions in the order in which they are written making use of the index settings, wherever specified, to determine the occurrence of those items to be tested. If none of the conditions are satisfied, the index-name for identifier-1 is incremented to obtain reference to the next occurrence. The process is repeated, using the new index-name settings. If the new value of the index-name settings for identifier-1 corresponds to a table element outside the permissible range of occurrence values, the search terminates as indicated in the rule above. If one of the conditions is satisfied upon its evaluation, the search terminates immediately and the imperative statement associated with that condition is executed; the index-name remains set at the occurrence which caused the condition to be satisfied.
2. In Format one, if the VARYING phrase is not used, the index-name which is used for the search operation is the first (or only) index-name

appearing in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

3. In Format one, if the VARYING index-name is specified, and if index-name-1 appears in the INDEXED BY phrase of identifier-1, that index-name is used for this search. If this is not the case, or if the VARYING identifier-2 phrase is specified, the first (or only) index-name given in the INDEXED BY phrase of identifier-1 is used for the search. In addition, the following operations will occur:
   - If the VARYING index-name-1 phrase is used, and if index-name-1 appears in the INDEXED BY phrase or another table entry, the occurrence number represented by index-name-1 is incremented by the same amount as, and at the same time as, the occurrence number represented by the index-name associated with identifier-1 is incremented.
   - If the VARYING identifier-2 phrase is specified, and identifier-2 is an index data item, then the data item referenced by identifier-2 is incremented by the same amount as, and at the same time as, the index associated with identifier-1 is incremented. If identifier-2 is not an index data item, the data item referenced by identifier-2 is incremented by the value one at the same time as the index referenced by the index-name associated with identifier-1 is incremented.
4. In Format two SEARCH statement, results of the SEARCH ALL operation are predictable only when:
   - The data in the table is ordered in the same manner as described in the ASCENDING/DESCENDING KEY clause associated with the description of identifier-1.
   - The contents of the key(s) referenced in the WHEN clause are sufficient to identify an unique table element.
5. When Format two SEARCH ALL is used, a onserial type of search operation may take place; the initial setting of the index-name for identifier-1 is ignored and its setting is varied during the search operation, with the restriction that at no time is it set to a value that exceeds the value which corresponds to the last element of the table, or that is less than the value that corresponds to the first element of the table. The length of the table is discussed in the OCCURS clause at the beginning of this section.

   If any of the conditions specified in the WHEN clause cannot be satisfied for any setting of the index within the permitted range, control is passed to imperative-statement-1 of the AT END phrase, when specified, or to the next executable sentence when this phrase is not specified; in either case the final setting of the index is not predictable. If all the conditions can be satisfied, the index indicates an occurrence that allows the conditions to be satisfied, and control passes to imperative-statement-2.
6. If imperative-statement-1, imperative-statement-2, or imperative-statement-3, does not terminate with a GO TO statement, control passes to the next executable sentence.
7. In Format two, the index-name that is used for the search operation is the first (or only) index-name that appears in the INDEXED BY clause of identifier-1. Any other index-names for identifier-1 remain unchanged.
8. If identifier-1 is a data item subordinate to another data item containing an OCCURS clause (providing for a two or three dimensional table), an

index-name must be associated with each dimension of the table. This is accomplished through the INDEXED BY phrase of the OCCURS clause. Only the setting of the index-name associated with identifier-1 (and identifier-2 or index-name-1, if present) is modified by the execution of the SEARCH statement. To search an entire two or three dimensional table, it is necessary to execute a SEARCH statement several times. Prior to each execution of a SEARCH statement, SET statements must be executed to adjust index-names to appropriate settings.

9. A flowchart of the Format one SEARCH operation contaning two WHEN phrases is presented in Figure 10-1.

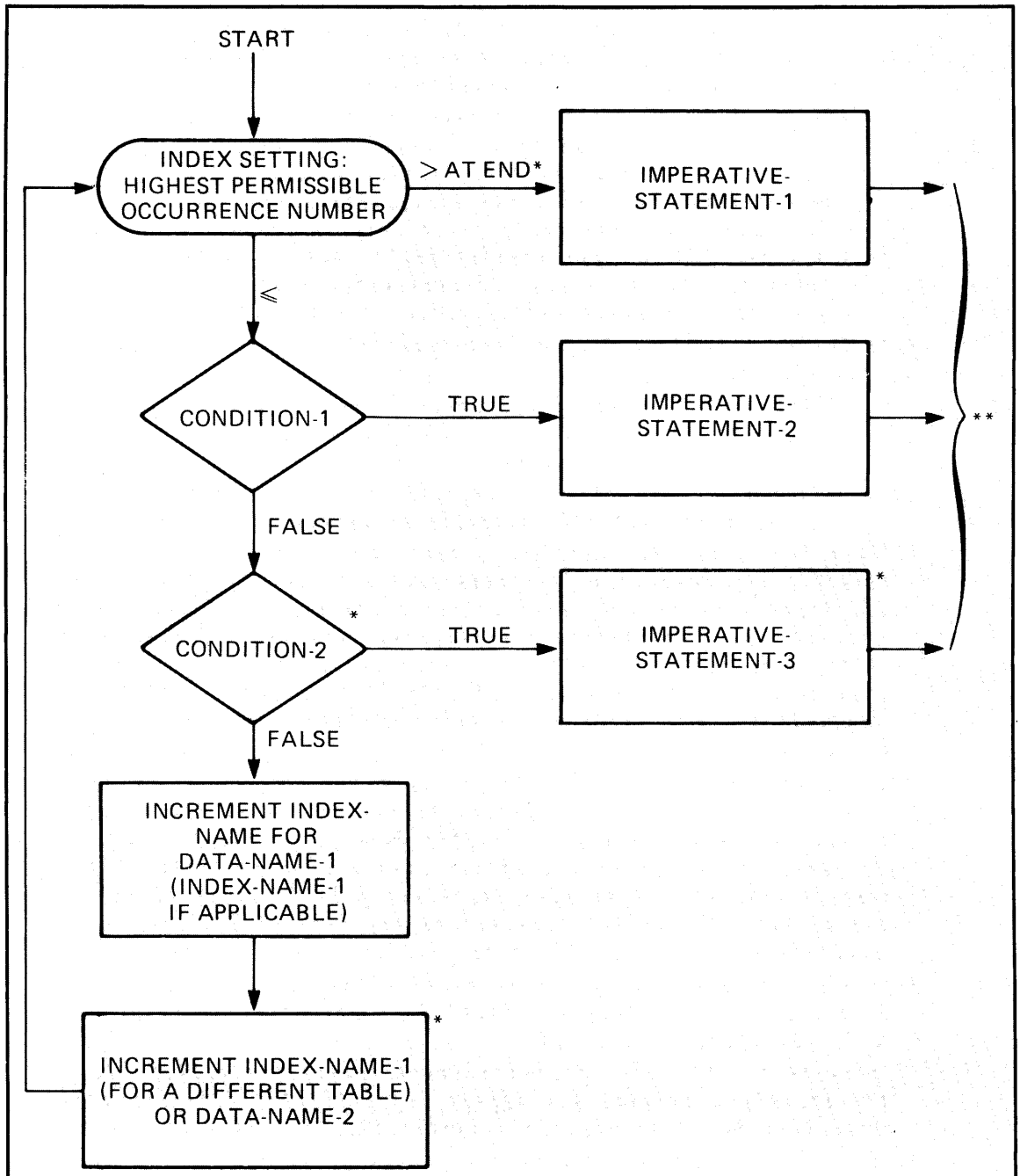


Figure 10-1. Format One SEARCH Operation Flowchart

11
Sort module

DEFINITION

The Sort facility of Sort Module is capable of ordering one or more record files, according to a set of user-specified keys contained within each record.

To accomplish the Sort, the user must specify the File-Control SELECT clause in the Environment Division, the sort file description (SD) entry in the Data Division, and the SORT statement in the Procedure Division. The basic elements of the Sort, however, are the SD entry with its associated record description entries and the SORT statement.

Note

Prime COBOL does not currently support the Merge facility of the ANSI standard Sort/Merge Module.

DATA DIVISION

File section

An SD file description gives information about the sizes and the names of the data records associated with the file to be sorted. There are no label procedures which the user can control, and the rules for blocking and internal storage are peculiar to the SORT statement.

SORT file description

The sort file description furnishes information concerning the physical structure, identification, and record names of the file to be sorted.

Format

```
SD file-name

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

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

Syntax rules

1. The level indicator **SD** identifies the beginning of the sort file description and must precede the **file-name** of each sort-file. Note, an FD level indicator must precede the file-name of each file providing input or output to the sort operation.
2. The clauses which follow the file-name are optional, and their order of appearance is immaterial.
3. One or more record description entries must follow the SD entry; however, no READ, WRITE, OPEN or CLOSE statements may be executed for this file.

4. The file must be specified in a SELECT clause.

PROCEDURE DIVISION

RELEASE

The RELEASE statement transfers records to the initial phase of a SORT operation.

Format

```
RELEASE record-name [ FROM identifier]
```

Syntax rules

1. A RELEASE statement may be specified only within an input procedure associated with a SORT statement for a file whose SD entry contains record-name.
2. **Record-name** must be the name of a logical record in the associated SD entry. Record-name may be qualified.
3. Record-name and **identifier** must not refer to the same storage area.

General rules

1. The execution of a RELEASE statement causes the record-name to be released to the initial phase of a SORT operation.
2. If the **FROM** phrase is specified, the contents of the identifier are moved to the record-name, then the contents of the record-name are released to the sort file. Moving takes place according to the rules for the MOVE statement without the CORRESPONDING phrase. The information in the record-name is no longer available, but the information in the identifier is still available.
3. After the execution of the RELEASE statement, the information in record-name is no longer available, unless the associated sort file is named in a SAME RECORD AREA clause, in which case record-name is still available as a record of other files specified in the clause. When control passes from the input procedure, the file consists of all those records placed in it by the execution of RELEASE statements.

RETURN

The RETURN statement obtains sorted records from the final phase of a SORT operation.

Format

```
RETURN file-name RECORD [INTO identifier]

    AT END imperative-statement
```

Syntax rules

1. **File-name** must be described by an SD entry in the Data Division.
2. A RETURN statement may be specified only within an output procedure associated with a SORT statement for file-name.
3. The **INTO** phrase must not be used if the input file contains logical records of various sizes.

4. The record areas associated with **identifier** and file-name must not be the same storage area.

General rules

1. If more than one record description is associated with file-name, these records automatically share the same storage area; that is, the area is implicitly redefined. After the execution of the RETURN statement, any data items which lie beyond the range of the current record are undefined.
2. When the RETURN statement is executed, the next record from file-name is made available for processing in the record areas associated with the sort-file.
3. If the INTO phrase is specified, the current record is moved from the input area to the area specified by identifier according to the rules for the MOVE statement without the CORRESPONDING phrase. The implied MOVE does not occur if there is an AT END condition. Any subscripting or indexing associated with identifier is evaluated after the record has been returned and immediately before it is moved to the identifier.
4. When the INTO phrase is used, the data is available in both the input record area and the data area associated with identifier.
5. After all the records have been returned from the file-name, the **AT END** condition occurs. The contents of the record areas associated with the file are undefined when that condition occurs. After the execution of the **imperative-statement** in the AT END phrase, no RETURN statement may be executed as part of the current output procedure.

SORT

The SORT statement creates a sort-file by executing input procedures or by transferring records from another file, sorts the records in the sort-file on a set of specified keys, and, in the final phase of the sort operation, makes available each record from the sort-file, in sorted order, to some output procedures or to an output file.

Format

```
SORT file-name-1 ON { ASCENDING  } KEY data-name-1 [, data-name-2] . . .
                    { DESCENDING }

{ INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2] }
                                      { THRU    }
{ USING file-name-2                                               }

{ OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
                                       { THRU    }
{ GIVING file-name-3                                              }
```

Syntax rules

1. SORT statements may appear anywhere except in the Declaratives portion of the Procedure Division or in an input or output procedure associated with a SORT statement.
2. **File-name-1** must be described in an SD entry in the Data Division.
3. If the **USING** phrase is specified and the file-name-1 contains variable-length records, the size of the records contained in the **file-name-2** must not be less than the smallest record nor larger than the largest record described for file-name-1. If file-name-1 contains fixed-length records, the size of the records contained in file-name-2 must not be larger than the largest record described for file-name-1.
4. **Data-name-1, data-name-2**, etc., are **KEY** data-names and are subject to the following rules:
   - The data items identified by KEY data-names must be described in records associated with file-name-1.
   - KEY data-names may be qualified.
   - The data items identified by KEY data-names may not be variable-length data items, nor may they name group items which contain variable-occurrence data items.
   - If file-name-1 has more than one record description, then the data items identified by KEY data-names need be described in only one of the record descriptions. In other words, the same character positions referenced by a KEY data-name in one record description entry are taken as the KEY in all records of the file-name-1.
   - The data items identified by KEY data-names may not contain an OCCURS clause or be subordinate to an item which contains an OCCURS clause.
5. **Section-name-1** specifies the first or the only section in an input procedure. **Section-name-2**, if specified, indentifies the last section of an input procedure.

   Section-name-3 and **section-name-4** apply to an output procedure.
6. The words **THRU** and **THROUGH** are equivalent.
7. In the Data Division, file-name-2 and file-name-3 must be described in an FD entry, not in an SD entry.
8. If the **GIVING** phrase is specified and the **file-name-3** contains variable-length records, the size of the records contained in the file-name-1 must not be less than the smallest record nor larger than the largest record described for file-name-3. If file-name-3 contains fixed-length records, the size of the records contained in file-name-1 must not be larger than the largest record described for file-name-3.

General rules

1. If file-name-1 contains only fixed-length records, any record in file-name-2 released to file-name-1 is left justified, and any unused character positions at the right end of the record will be filled with blanks.
2. The data-names following the word KEY are listed in order of decreasing significance no matter how they are divided into KEY phrases. For example, data-name-1 is the major key, data-name-2 is the next most significant key, etc.

- When the **ASCENDING** phrase is specified, the sorted sequence will be from the lowest key value to the highest key value.
- When the **DESCENDING** phrase is specified, the sorted sequence will be from the highest key value to the lowest key value.
- The key values are compared according to the rules for comparison of operands in a relation condition. (See Conditional Expressions in Section 4 and IF Statement in Section 8.)

3. If the contents of all KEY data items associated with two or more data records are equal, then the order of return for the records is undefined.
4. The input procedure must consist of one or more sections that are written consecutively and do not form a part of any output procedure. In order to transfer records to file-name-1, the input procedure must include at least one RELEASE statement. Control must not be passed to the input procedure except when a related SORT statement is being executed.

   The input procedure can include any procedures needed to select, create, or modify records. There are three restrictions on the procedural statements within the input procedure:

   - The input procedure must not contain any SORT statements.
   - The input procedure must not contain any explicit transfers of control to points outside the input procedure; GO TO and PERFORM statements in the input procedure are not permitted to refer to procedure-names outside the input procedure. COBOL statements are allowed that will cause an implied transfer of control to Declaratives.
   - The remainder of the Procedure Division must not contain any transfers of control to points inside the input procedure; GO TO and PERFORM statements in the remainder of the Procedure Division must not refer to procedure-names within the input procedure.

5. If an input procedure is specified, control is passed to the input procedure before the file-name-1 is sequenced by the SORT statement. Before control passes the last statement in the input procedure, the file-name-3 must not be open. The compiler inserts a return mechanism at the end of the last section in the input procedure and when control passes the last statement in the input procedure, the records that have been released to the file-name-1 are sorted.
6. During the execution of the input procedure, the output procedure or any USE AFTER EXCEPTION procedures, no statement manipulating the files referenced by, or accessing the record areas associated with file-name-2 or file-name-3 may be executed.
7. If the USING phrase is specified, all the records in file-name-2 are automatically transferred to file-name-1. At the time of execution of the SORT statement, file-name-2 must not be open. For file-name-2, the execution of the SORT statement causes the following actions to be taken:

   - The processing of the file is initiated. The initiation is performed as if an OPEN statement with the INPUT phrase had been executed.

- The logical records are obtained and released to the sort operation. Each record is obtained as if a READ statement with the NEXT and the AT END phrase had been executed.
- The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

8. The output procedure must consist of one or more sections that are written consecutively and do not form a part of any input procedure. In order to make sorted records available for processing, the output procedure must include at least one RETURN statement. Control must not be passed to the output procedure except when a related SORT statement is being executed. The output procedure may consist of any procedures needed to select, modify or copy the records that are being returned, one at a time in sorted order, from the sort file. There are three restrictions on the procedural statements within the output procedure:
   - The output procedure must not contain any SORT statements.
   - The output procedure must not contain any explicit transfers of control to points outside the output procedure; GO TO and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure. COBOL statements are allowed that will cause an implied transfer of control to Declaratives.
   - The remainder of the Procedure Division must not contain any transfers of control to points inside the output procedure; GO TO and PERFORM statements in the remainder of the Procedure Division must not refer to procedure-names within the output procedure.
9. If an output procedure is specified, control passes to it after file-name-1 has been sequenced by the SORT statement. The file-name-2 must not be open. The compiler inserts a return mechanism at the end of the last section in the output procedure and when control passes the last statement in the output procedure, the return mechanism terminates the sort, and then passes control to the next executable statement after the SORT statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record in sorted order, when requested. The RETURN statements in the output procedure are the requests for the next record.
10. If the GIVING phrase is specified, all the sorted records are automatically written on file-name-3 as the implied output procedure for the SORT statement. At the time of the execution of the SORT statement, file-name-3 must not be open. For file-name-3, the execution of the SORT statement causes the following actions to be taken:
    - The processing of the file is initiated. The initiation is performed as if an OPEN statement with the OUTPUT phrase had been executed.
    - The sorted logical records are returned and written onto the file. The records are written as if a WRITE statement without any optional phrases had been executed.
    - The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

11. If file-name-3 contains only fixed-length records, any record in file-name-1 containing less character positions is padded with blanks at the right end of the record when the record is returned from file-name-3.

A listing file for sample program SAMPLE.SORT is presented below.

```
Rev 17.0  COBOL     Source File:  SAMPLE.SORT                    08/13/79  14:25
 (0001)          ID DIVISION.
 (0002)          PROGRAM-ID.  SORTIT.
 (0003)          ENVIRONMENT DIVISION.
 (0004)          CONFIGURATION SECTION.
 (0005)          SOURCE-COMPUTER. PRIME.
 (0006)          OBJECT-COMPUTER. PRIME.
 (0007)          INPUT-OUTPUT SECTION.
 (0008)          FILE-CONTROL.
 (0009)              SELECT NET-FILE-IN ASSIGN TO PFMS.
 (0010)              SELECT NET-FILE-OUT ASSIGN TO PFMS.
 (0011)              SELECT NET-FILE-WORK ASSIGN TO PFMS.
 (0012)          DATA DIVISION.
 (0013)          FILE SECTION.
 (0014)          SD  NET-FILE-WORK.
 (0015)          01  SALES-RECORDS.
 (0016)              05  EMPL-NO              PIC 9(6).
 (0017)              05  DEPT                 PIC 99.
 (0018)              05  NET-SALES            PIC 9(7)V99.
 (0019)              05  NAME-ADR             PIC X(61).
 (0020)              05  MONTH                PIC XX.
 (0021)          FD  NET-FILE-IN
 (0022)              LABEL RECORDS ARE STANDARD
 (0023)              VALUE OF FILE-ID 'FILEIN'.
 (0024)          01  NET-CARD-IN.
 (0025)              05  EMPL-NO-IN           PIC 9(6).
 (0026)              05  DEPT-IN              PIC 99.
 (0027)                  88  OFF-SITE-LOCATION   VALUE 7, 9.
 (0028)              05  NET-SALES-IN         PIC 9(7)V99.
 (0029)              05  NAME-ADDR-IN         PIC X(61).
 (0030)              05  MONTH-IN             PIC 99.
 (0031)          FD  NET-FILE-OUT
 (0032)              LABEL RECORDS ARE STANDARD
 (0033)              VALUE OF FILE-ID 'FILEOUT'.
 (0034)          01  NET-CARD-OUT.
 (0035)              05  EMPL-NO-OUT          PIC 9(6).
 (0036)              05  DEPT-OUT             PIC 99.
 (0037)              05  NET-SALES-OUT        PIC 9(7)V99.
 (0038)              05  NAME-ADDR-OUT        PIC X(61).
 (0039)              05  MONTH-OUT            PIC 99.
 (0040)          WORKING-STORAGE SECTION.
 (0041)          77  SUM-DEPT             PIC S9(14)V99 VALUE ZEROS.
 (0042)          01  MONTH-ACCEPT         PIC 99 VALUE ZERO.
 (0043)              88  VALID-MONTH      VALUE 01 THRU 12.
 (0044)          01  TABLE-VALUE.
 (0045)              02  FILLER           PIC X(5) VALUE '01JAN'.
 (0046)              02  FILLER           PIC X(5) VALUE '02FEB'.
 (0047)              02  FILLER           PIC X(5) VALUE '03MAR'.
 (0048)              02  FILLER           PIC X(5) VALUE '04APR'.
 (0049)              02  FILLER           PIC X(5) VALUE '05MAY'.
 (0050)              02  FILLER           PIC X(5) VALUE '06JUN'.
 (0051)              02  FILLER           PIC X(5) VALUE '07JUL'.
```

```
(0052)              02  FILLER          PIC X(5) VALUE '08AUG'.
(0053)              02  FILLER          PIC X(5) VALUE '09SEP'.
(0054)              02  FILLER          PIC X(5) VALUE '10OCT'.
(0055)              02  FILLER          PIC X(5) VALUE '11NOV'.
(0056)              02  FILLER          PIC X(5) VALUE '12DEC'.
(0057)          01  MONTH-TABLE REDEFINES TABLE-VALUE.
(0058)              02  MON-TAB  OCCURS 12 TIMES INDEXED BY INDX
(0059)                  ASCENDING KEY MONTH-NO.
(0060)                  03  MONTH-NO    PIC 99.
(0061)                  03  MONTH-VALUE PIC XXX.
(0062)          01  TABLE-AREA.
(0063)              03  SITE    OCCURS 2 TIMES INDEXED BY INDX1.
(0064)                05  MONTHS    OCCURS 12 TIMES INDEXED BY INDX2.
(0065)                  07  DEPT-TOTAL    OCCURS 7 TIMES INDEXED BY INDX3
(0066)                                     PIC S9(14)V99 COMP-3.
(0067)         *
(0068)          01  DISPLAY-TOTALS.
(0069)              02  FILLER          PIC XX VALUE SPACE.
(0070)              02  PRINT-MONTH     PIC XXX VALUE SPACE.
(0071)              02  FILLER          PIC X(16) VALUE '  TOTAL SALES = '.
(0072)              02  PRINT-SUM       PIC ZZ,ZZZ,ZZZ,ZZZ,ZZZ.99-.
(0073)          PROCEDURE DIVISION.
(0074)          START-PARA.
(0075)              PERFORM INT-PARA
(0076)                  VARYING INDX1 FROM 1 BY 1
(0077)                    UNTIL INDX1 > 2
(0078)                  AFTER INDX2 FROM 1 BY 1
(0079)                    UNTIL INDX2 > 12
(0080)                  AFTER INDX3 FROM 1 BY 1
(0081)                    UNTIL INDX3 > 7.
(0082)              GO TO SORT-PARA.
(0083)         *
(0084)          INT-PARA.
(0085)              MOVE ZEROS TO DEPT-TOTAL(INDX1, INDX2, INDX3).
(0086)         *
(0087)          SORT-PARA.
(0088)              SORT NET-FILE-WORK
(0089)                  ASCENDING KEY DEPT
(0090)                  DESCENDING KEY NET-SALES
(0091)                  INPUT PROCEDURE SCREEN-DEPT
(0092)                  GIVING NET-FILE-OUT.
(0093)              GO TO GET-TOTALS.
(0094)         *
(0095)          SCREEN-DEPT SECTION.
(0096)          S-DEPT1.
(0097)              OPEN INPUT NET-FILE-IN.
(0098)          S-DEPT2.
(0099)              READ NET-FILE-IN
(0100)                  AT END GO TO S-DEPT-FINAL.
(0101)              SET INDX1 TO 2.
(0102)              IF NOT OFF-SITE-LOCATION
(0103)                 MOVE NET-CARD-IN TO SALES-RECORDS
(0104)                 RELEASE SALES-RECORDS
(0105)                 SET INDX1 TO 1.
```

```
(0106)              SET INDX2 TO MONTH-IN.
(0107)              SET INDX3 TO DEPT-IN.
(0108)              ADD NET-SALES-IN TO DEPT-TOTAL (INDX1, INDX2, INDX3).
(0109)              GO TO S-DEPT2.
(0110)           S-DEPT-FINAL.
(0111)              CLOSE NET-FILE-IN.
(0112)           S-DEPT-END.
(0113)              EXIT.
(0114)          *
(0115)           GET-TOTALS SECTION.
(0116)           GET-TOTAL.
(0117)              DISPLAY 'ENTER MONTH XX (01-12) OR ENTER 99 TO QUIT'.
(0118)              ACCEPT MONTH-ACCEPT.
(0119)              IF MONTH-ACCEPT = 99
(0120)                  GO TO DONE-PARA.
(0121)              IF NOT VALID-MONTH
(0122)                  GO TO GET-TOTAL.
(0123)              PERFORM FIND-MONTH.
(0124)              SET INDX1 TO 1.
(0125)              SET INDX2 TO MONTH-ACCEPT.
(0126)              DISPLAY 'IN STATE'.
(0127)           GET-NEXT.
(0128)              SET INDX3 TO 1.
(0129)              PERFORM ADD-TOTALS 7 TIMES.
(0130)              MOVE SUM-DEPT TO PRINT-SUM.
(0131)              DISPLAY DISPLAY-TOTALS.
(0132)              MOVE ZEROS TO SUM-DEPT.
(0133)              SET INDX1 UP BY 1.
(0134)              IF INDX1 > 2
(0135)                  GO TO GET-TOTAL.
(0136)              DISPLAY 'OUT OF STATE'.
(0137)              GO TO GET-NEXT.
(0138)           *
(0139)           ADD-TOTALS.
(0140)              ADD DEPT-TOTAL(INDX1, INDX2, INDX3) TO SUM-DEPT.
(0141)              SET INDX3 UP BY 1.
(0142)           *
(0143)           FIND-MONTH.
(0144)              SEARCH ALL MON-TAB
(0145)                  WHEN MONTH-NO(INDX) = MONTH-ACCEPT
(0146)                  MOVE MONTH-VALUE(INDX) TO PRINT-MONTH.
(0147)           *
(0148)           DONE-PARA.
(0149)              STOP RUN.

                 .
                 .
                 .
```

No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.00.12 <SORTIT>

12

Indexed sequential files

DEFINITION

The indexed sequential system incorporates the concept of accessing data selectively in a sequentially structured file. (Only the index which points to the data is sequential.) The data base is created in ascending sequential order on a direct access device, and concurrently a hierarchy of indices is constructed. The indices can be used to directly locate a given record within the file.

The sequence of the indices relating to a record depends on a field within the data records which is specified by the programmer, in a RECORD KEY clause. The record key(s) are the elements which identify each record in a file.

FILE CONTROL

Format

```
SELECT file-name ASSIGN TO PFMS
    ; ORGANIZATION IS INDEXED
    [; ACCESS MODE IS { SEQUENTIAL | RANDOM | DYNAMIC } ]
    ; RECORD KEY IS data-name-1
    [; ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES] ] . . .
    [; FILE STATUS IS data-name-3].
```

General rules

SELECT file-name

1. The SELECT clause specifies the name of the indexed sequential file. Refer to ENVIRONMENT DIVISION for rules.

ORGANIZATION IS INDEXED

2. This clause specifies that the file named in the SELECT statement contains data organization by indices, and that it is to be processed by the Multiple Index Data Access System, MIDAS. (See Appendix B.)

[ACCESS MODE IS { SEQUENTIAL | RANDOM | DYNAMIC }]

3. The ACCESS MODE clause specifies how an indexed file is written or retrieved.

   - **SEQUENTIAL**: If access mode is not specified, the default is SEQUENTIAL. This access mode specifies that records will be written or retrieved sequentially. When a WRITE statement is used, the record must be submitted in ascending sequence by RECORD KEY value. A READ statement retrieves the record sequentially.
   - **RANDOM**: When the RANDOM is specified, the records are to be written or retrieved randomly, based on the value placed in the RECORD KEY field prior to a READ or WRITE. The complete RECORD KEY value must be placed in data-name-1, prior to READ, otherwise the record will not be found. Random mode precludes a sequential READ or WRITE.
   - **DYNAMIC**: When DYNAMIC access method is specified, a program can read or write randomly or sequentially.

RECORD KEY IS data-name-1

4. The RECORD KEY clause specifies the data item within each record which is used for the primary index.

   - **Data-name-1** must be defined in the Record Description associated with the FD entry for the file.
   - Data-name-1 must be the first entry in the Record Description. Multiple Record Descriptions must have the same corresponding data description for the record key.
   - Data-name-1 must not be specified with an OCCURS clause, or be contained within a group affected by an OCCURS clause.
   - Data-name-1 must not be specified with a P character in its PICTURE clause, or be described with a separator sign (/).
   - Data-name-1 must have the same description and relative location as when the file was created.
   - Data-name-1 cannot exceed 32 characters.
   - The value contained within data-name-1 must be unique, duplicates are invalid.

[ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] . . .

5. The ALTERNATE RECORD KEY clause specifies a data item, which is used for a secondary index, within each record. There may be up to five alternate record keys. Alternate record key cannot be embedded within the primary record key. See rules under RECORD KEY.

   Specification of **WITH DUPLICATES** allows keys containing the same value to be placed in the file. If WITH DUPLICATES is specified, duplicates must be allowed for the corresponding secondary index when the MIDAS template is created; the admissability of duplicates cannot be changed at the program level.

[FILE STATUS IS data-name-3]

6. The **FILE STATUS** is a two-character (one-word) unsigned field described in the Working-Storage Section. The operating system moves a value into data-name-3 following the execution of every statement which explicitly or implicitly references the file. This value indicates the execution status of the statement to the program. Following a successful READ or WRITE, etc., data-name-3 contains 00. The complete status codes are described in Table C-4, Appendix C.

PROCEDURE DIVISION

The COBOL statements listed in this section apply to their application in indexed file processing.

A complete description of all COBOL verbs, their functions, formats, and rules, is provided in Section 8, PROCEDURE DIVISION.

The INVALID KEY clause may be written for indexed files in the START, READ, WRITE, REWRITE or DELETE statements. Its format is:

. . . [INVALID KEY imperative-statement]

The INVALID KEY clause is executed if there is an error status code condition, in which case control is transferred to imperative-statement. If this clause is not present, control is passed to the DECLARATIVES section for the corresponding file. If neither is specified, the program will abort during execution. The result for the INVALID condition is returned via the ERROR STATUS code. See Table C-4.

CLOSE

Format

CLOSE file-name-1 [, file-name-2] . . .

General rule

This is the only option possible for an indexed file.

DELETE

Format

DELETE file-name RECORD [; INVALID KEY imperative-statement]

General rules

1. The DELETE statement logically removes a data record from the indexed file together with all the indices.
2. In SEQUENTIAL access, the record to be deleted must have been successfully read before a delete can be executed. The primary RECORD KEY cannot be changed between the READ and DELETE statement, otherwise the **INVALID KEY** clause will be activated.
3. RANDOM and DYNAMIC access modes only need to place the value of the record to be deleted in the RECORD KEY field. If that record does not exist in the file, the INVALID KEY statement is executed and the ERROR STATUS field will contain a value of 23.

OPEN

Format

```
OPEN { { I-O    }                                  }
     { { INPUT  } file-name-1 [, file-name-2] ... } ...
     { { OUTPUT }                                  }
```

General rules

1. A file opened as **INPUT** can only be accessed in a READ statement.
2. A file opened as **OUTPUT** can only be accessed in a WRITE statement.
3. A file opened as **I/0** can be either read or written with lock record.

Note

Table C-5 in Appendix C specifies the types of OPEN statements which are permissible with the different ACCESS modes.

READ

Format one (SEQUENTIAL or DYNAMIC)

```
READ file-name [NEXT] RECORD [INTO data-name-1]

 AT END imperative-statement]
```

Format two (SEQUENTIAL, RANDOM or DYNAMIC)

```
READ file-name RECORD  INTO data-name-1]

 KEY IS data-name-2]

 INVALID KEY imperative-statement]
```

General rules

1. Format one, Option one (SEQUENTIAL ACCESS only):

```
READ file-name RECORD  INTO data-name-1]

     [AT END imperative-statement]
```

A file is read sequentially based on the primary index (RECORD KEY). If one of the secondary index sequences is to be used, the index must be established via a Format two, Option two READ statement. Thereafter, the file can be read with a Format one, Option one format. If the **INTO** clause is used, the data record is automatically moved into **data-name-1.** When **AT END** is specified, control is passed when the complete file has been read.

2. Format one, Option two (DYNAMIC and SEQUENTIAL ACCESS):

READ file-name [NEXT] RECORD [INTO data-name-1]

[AT END imperative-statement]

- For **DYNAMIC** access: This option allows the programmer to change from a random mode to sequential reading with the NEXT record clause. The **INTO** clause automatically moves the data-record into **data-name-1**. The **AT END** clause transfers control at the end of the file.
- If the **NEXT RECORD** option is not specified, the value of the record to be retrieved must be placed in the RECORD KEY data-name.
- For **SEQUENTIAL** access: The NEXT RECORD is not required with sequential access; it is automatically accessed.

3. Format two, Option one:

READ file-name RECORD [INTO data-name-1]

[INVALID KEY imperative-statement]

- For **SEQUENTIAL** access: The format will read the file sequentially based on the specified index, or be defaulted to the primary index. The **INTO** moves data into **data-name-1**. INVALID KEY transfers control if any of the status codes listed in Table C-4 are encountered.
- For **DYNAMIC** and **RANDOM** access: The format will retrieve data based on the value contained in data-name

(primary or secondary index). If the record is not found or, any other error status is encountered, control is passed to the INVALID KEY (refer to Table C-4). The INTO clause moves data to data-name-1.

4. Format two, Option two:

READ file-name RECORD [INTO data-name-1]

[KEY IS data-name-2]

[INVALID KEY imperative-statement]

This format is used to perform keyed access, allowing the file to be retrieved based on the RECORD KEY or ALTERNATE RECORD KEYS (secondary indexes) via the **KEY IS** clause. Once this format is executed, the Format one READ statement should be used. The index is used for each READ until another secondary index is specified via the KEY IS clause of a READ statement.

REWRITE

Format

```
REWRITE record-name [FROM data-name]

    [INVALID KEY imperative-statement]
```

General rules

1. The REWRITE statement physically replaces an existing record.
2. The REWRITE statement can change any or all data-fields in the record except the prime record key.
3. The file must be opened for I-O for all access methods.
4. A record must have been READ successully prior to the REWRITE. This is required to lock the record and ensure that it cannot be updated by another program running concurrently.
5. In the **FROM data-name** option, the primary RECORD KEY must equal the key from the previous READ or the INVALID KEY conditions will occur. The FROM option allows the record to be created in another area. It is equivalent to MOVE data-name TO record-name prior to the execution of the REWRITE statement.
6. Control is passed to the **INVALID KEY** statement if the primary key is changed. If this statement is not present, control is then passed to the USE DECLARATIVES. One or the other of these statements must be present, or the program will terminate if the invalid statement is activated. Refer to Table C-4 for status codes.

START

Format

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

    [INVALID KEY imperative-statement]
```

General rules

1. The START statement enables an indexed organized 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 START verb is not allowed with the random access.
2. Option one:

   START file-name.

   This option positions the file to the value contained in the RECORD KEY data-name. If that record is not present in the file, control is passed to the DECLARATIVES section if present; otherwise, the program terminates.

3. Option two:

START file-name KEY IS data-name.

This option will position the file to the value contained in data-name (data-name is the name of either RECORD KEY or one of the ALTERNATE RECORD KEYs). If the record is not contained in the file, control is passed to the DECLARATIVES section if present; otherwise, the program terminates.

4. Option three:

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

[INVALID KEY imperative-statement]

If the option **GREATER** or **NOT LESS** is specified, the file is positioned for the next access to be greater than or less than the value specified in the **data-name**.

The VALID clause or DECLARATIVES is taken if there is no data satisfying data-name, and the STATUS code returned is 23.

5. START does not retrieve a record, but only positions to a desired record.

Consider the following short indexed file. Each record contains just two fields: A NAME field which serves as primary key, and a COMPANY field:

```
|  NAME  |  COMPANY  |
```

Source coding relating to the file might be:

```
ENVIRONMENT DIVISION.
     .
     .
     .
      SELECT FILE-1 ASSIGN TO PFMS
           ORGANIZATION IS INDEXED
           ACCESS IS DYNAMIC
           RECORD KEY IS NAME.
     .
     .
     .
DATA DIVISION.
FILE SECTION.
FD  FILE-1 LABEL RECORDS ARE STANDARD
           VALUE OF FILE-ID IS 'FILE-1'.
    01  FILE-1-RECORD.
        05  NAME    PIC X(10).
        05  COMPANY PIC X(25).
```

A pictoral view of this file is presented below.

	data-name PICTURE	NAME PIC X(10)	COMPANY PIC X(25)
Values:		BLYE	REPORTCO
		CLAPP	MERGANTHALER
		GRIER	AUTOMATION
		HARPER	DESIGNERS
		KEANE	REPORTCO

5. If a sequential traverse of this file is performed, records are returned in sequence based on primary key:

```
BLYE      REPORTCO
CLAPP     MERGANTHALER
GRIER     AUTOMATION
HARPER    DESIGNERS
KEANE     REPORTCO
```

To obtain specific records with a START statement, the key field (NAME) has to be initialized.
If the intent is to obtain records of people whose name begins with the characters F, G, H, and I, program actions should include the following type of logic:

```
MOVE 'F' to NAME.
```

Place value in key field.

```
START FILE-1 KEY IS NOT LESS THAN NAME.
        .
        .
        .
```

Find the first record whose key is not less than 'F'. This positions the file to the records.

```
READ FILE-1 NEXT RECORD.
        .
        .
        .
```

This action will retreive the desired record. In this example it will be the record 'GRIER AUTOMATION'.

```
READ FILE-1 NEXT RECORD.
        .
        .
        .
```

This action will retreive the next sequential record 'HARPER DESIGNERS'.

```
READ FILE-1 NEXT RECORD.
```

This action will retreive the next sequential record, 'KEANE REPORTCO'. Examination will indicate that all desired records have been obtained.

WRITE

Format

```
WRITE record-name [FROM data-name-1]

      [INVALID KEY imperative-statement]
```

General rules

1. The WRITE function releases a logical record for an output or I-O file.
2. Prior to the WRITE statement, a valid, unique value must be in the primary RECORD KEY data-name. If the **FROM** option is used, the unique value in RECORD KEY data-name must be in the relative location of **data-name-1**. If the primary key is not unique, the invalid statement or the DECLARATIVE section will be executed. Refer to Table 18-1 for error conditions.

13

Relative file processing

DEFINITION

Relative file organization is permitted only with disk storage devices. Records are stored and retrieved based on a relative record number. For example, the 10th record is the one addressed by relative record number 10 and is the 10th record area whether or not records 1 through 9 have been written.

FILE CONTROL

Format

```
SELECT file-name ASSIGN TO PFMS
    ; ORGANIZATION IS RELATIVE
    [; ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name-1] }
                      { {RANDOM }                                  }
                      { {DYNAMIC} , RELATIVE KEY IS data-name-1    } ]
    [; FILE STATUS IS data-name-2].
```

General rules

SELECT file-name

1. This clause specifies the name of the relative file. Refer to ENVIRONMENT DIVISION for rules.

ORGANIZATION IS RELATIVE

2. This clause pecifies that the file named in the SELECT statement contains data organized by record number and processed by the File Processing facility of the operating system.

```
[; ACCESS MODE IS { SEQUENTIAL } ]
                  { RANDOM     }
                  { DYNAMIC    }
```

3. This clause specifies how a relative file is written or retrieved.
   - SEQUENTIAL: If access mode is not specified, the access mode will default to SEQUENTIAL. This access mode specifies that records will be written or retrieved sequentially. A READ statement retrieves the records sequentially.

- **RANDOM**: Specifies that the records are to be written or retrieved randomly based on the value placed in the RELATIVE KEY field prior to a READ or WRITE. When RANDOM access is used, the complete RELATIVE KEY value must be placed in RELATIVE KEY, or the record will not be found. Random mode precludes a sequential READ or WRITE.
- **DYNAMIC**: When this access method is specified, the program can read or write randomly or sequentially.

RELATIVE KEY IS data-name-1

4. The RELATIVE KEY clause, whose value is the relative record number of the record to be accessed, specifies the data item within Working-Storage Section.
   - **Data-name-1** must not be defined in the Record Description.
   - Data-name-1 must not be specified with an OCCURS clause, or be contained within a group affected by an OCCURS clause.
   - Data-name-1 must not be specified with a P character in its PICTURE clause, or be described with a separator sign (/).
   - Data-name-1 must be a valid numeric integer, and cannot contain a value greater than 999,999.
   - The value contained within data-name-1 must be unique; duplicates are invalid.

   The RELATIVE KEY is optional if access is sequential. However, in the creation of the MIDAS template, a RELATIVE KEY size equal to the maximum (48 bits), must be given.

[FILE STATUS IS data-name-2]

5. The FILE STATUS is a two-character (one word), unsigned field described in the Working-Storage Section. The operating system moves a value into data-name-2 following the execution of every statement which explicitly or implicitly references the file. This value indicates the execution status of the statement o the program. Following a successful READ or WRITE, etc., data-name-2 contains 00. For complete status codes, see Table C-4 in Appendix C.

PROCEDURE DIVISION

The COBOL statements listed in this section apply to relative file processing.

A complete description of all COBOL verbs, their functions, formats, and rules, is provided in Section 8, PROCEDURE DIVISION.

The INVALID KEY clause may be written for relative files in the START, READ, WRITE, REWRITE or DELETE statements. Its format is:

... [INVALID KEY imperative-statement]

The INVALID KEY clause is executed if there is an error status code condition, in which case control is transferred to imperative-statement. If this clause is not present, control is passed to the DECLARATIVES section for the corresponding file. If neither is specified, the program will abort during execution. The result for the INVALID condition is returned via the ERROR STATUS code (see Table C-4).

CLOSE

Format

```
CLOSE file-name-1 [, file-name-2] ...
```

General rule

This is the only option possible for a relative file.

DELETE

Format

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

General rules

1. The DELETE statement logically removes a data record from the relative file.
2. In SEQUENTIAL access, the record to be deleted must have been successfully read before a DELETE can be executed. The RELATIVE KEY cannot be changed between the READ and DELETE statement, otherwise the INVALID KEY clause will be activated.
3. RANDOM and DYNAMIC access modes only need to place the value of the record to be deleted in the RELATIVE KEY field. If that record does not exist in the file, the INVALID KEY statement is executed and the ERROR STATUS field will contain a value of 23.

OPEN

Format

```
OPEN { { I-O    } filename-1 [, file-name-2] ... } ...
     { { INPUT  }
     { { OUTPUT }
```

General rules

1. A file opened as **INPUT** can only be accessed in a READ statement.
2. A file opened as **OUTPUT** can only be accessed in a WRITE statement.
3. A file opened as **I-O** can be either read or written.

Note

Table C-5 in Appendix C specifies the types of OPEN statements which are permissible with the different ACCESS modes.

READ

Format one (SEQUENTIAL or DYNAMIC)

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

Format two (SEQUENTIAL, RANDOM or DYNAMIC)

```
READ file-name RECORD [INTO data-name-1]

     [INVALID KEY imperative-statement]
```

General rules

1. Format 1, Option 1 (SEQUENTIAL only):

```
READ file-name RECORD [INTO data-name-1]

     [AT END imperative-statement]
```

   For a sequential read, the file is read sequentially. If the **INTO** clause is used, the data record is automatically moved into **data-name-1**. When **AT END** is specified, control is passed to the **imperative-statement** when the complete file has been read.

2. Format 1, Option 2 (DYNAMIC and SEQUENTIAL):

```
READ file-name [NEXT] RECORD [INTO data-name-1]

     [AT END imperative-statement]
```

   - For **DYNAMIC** access: This option allows the programmer to change from a random mode to sequential reading with the NEXT option. The **INTO** clause automatically moves the data-record into **data-name-1**. The **AT END** clause transfers control at the end of the file.
   - If the **NEXT** option is not specified, the value of the record to be retrieved must be placed in the RELATIVE KEY data-name.
   - For **SEQUENTIAL** access: The NEXT option is not required.

3. Format 2, Option 1:

```
READ file-name RECORD [INTO data-name-1]

     [INVALID KEY imperative-statement]
```

   - For **SEQUENTIAL** access: The format reads the file sequentially. The RELATIVE KEY is updated with the record number after each successful READ. The **INTO** moves data into **data-name-1**. The **INVALID KEY** transfers control if any of the status codes listed in Table C-4 are encountered.
   - For **DYNAMIC** and **RANDOM** access: This format retrieves data based on the value contained in the RELATIVE KEY. If the record is not found, or any other error status is encountered, control is passed to the INVALID KEY clause. Refer to Table C-4. The INTO clause moves data to data-name-1.

REWRITE

Format

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

General rules

1. The REWRITE statement physically replaces an existing record.
2. The REWRITE statement can change any or all data-fields in the record.
3. The file must be opened for I/O for all access methods.
4. A record must have been READ successfully prior to the REWRITE statement. This ensures that the record cannot be updated by another program running concurrently.
5. The **FROM data-name** option allows the record to be created in another area. It is equivalent to a MOVE data-name TO record-name prior to the execution of the REWRITE statement.
6. Control is passed to the **INVALID KEY** statement if the RELATIVE KEY is changed since the successful read. If this statement is not present, control is then passed to the USE DECLARATIVES. One or the other of these statements must be present. Refer to Table C-4 for status codes.

START

Format

```
START file-name [KEY IS [ GREATER THAN
                          NOT LESS THAN
                          EQUAL TO      ] data-name]
        [INVALID KEY imperative-statement]
```

General rules

1. The START statement enables a relative 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 START verb is not allowed with RANDOM access (see INVALID KEY).
2. Option 1:

START file-name

This option positions the file to the value contained in the data-name as defined in RELATIVE KEY. If that record is not present in the file, control is passed to the DECLARATIVES section if present; otherwise, the program terminates.

3. Option 2:

START file-name KEY IS data-name

This option is synonymous to option 1 since there is only one key, RELATIVE KEY, in a relative file.

4. Option 3:

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

[INVALID KEY imperative-statement]

The option **GREATER** or **NOT LESS** is specified, the file is positioned for the next access to be greater than or less than the value specified in the data-name.

The **INVALID** clause or DECLARATIVES is taken if there is no data satisfying data-name, and the STATUS code returned is a 23.

5. START does not retrieve a record, but only positions to a desired record.

WRITE

Format

WRITE record-name [FROM data-name-1]

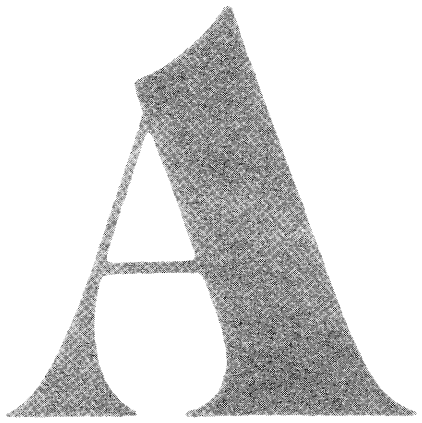
[INVALID KEY imperative-statement]

General rules

1. The WRITE statement releases a logical record to a file.
2. In the **FROM** option, **data-name-1** and **record-name** cannot reference the same memory location.
3. The file must be open for OUTPUT, or I-O.
4. The **INVALID KEY** clause must be specified if the DECLARATIVE section is not applicable. The program will terminate if an error code condition arises. Refer to Table C-4 for error codes.
5. For **SEQUENTIAL** access: If the file is opened as OUTPUT, the records are placed in the file in sequential order. The first record would have a position of 1, and the record number returned into the RELATIVE KEY data-name would be 1, etc.
6. For **DYNAMIC** and **RANDOM** access: The value of the record number must be placed in the RELATIVE KEY data-name.

IV

APPENDICES



File organization

ACCESS METHODS

Sequential Access Method (SAM)

SAM files require that all entries in a file preceding a desired entry be accessed in order to reach that entry. In other words, the file must be read sequentially. This is most useful for files in which information is normally entered into the file sequentially and retrieved from it in the same manner.

Direct Access Method (DAM)

DAM files (RELATIVE) permit access to a specific entry in a file by specification of physical disk record number. This permits the user to locate an entry within a known position in the file more quickly than does the SAM file structure. The size is restricted to 999,999 entries.

Indexed Sequential Access Method (INDEXED)

INDEXED method locates file entries through a key field search. The user may retrieve a data entry with only a few disk accesses, regardless of the position of the entry in the file. The primary index is based on the description of the record key. The key value is embedded in the first data field in the record. The secondary indexes are referenced by alternate record keys; up to five additional indexes may be specified. The user must show in advance which index is to be used to locate a data entry.

B

Creating ISAM and relative files - the MIDAS template

To initiate an Indexed Sequential or Relative file, a user must run a conversational program called CREATK to create a corresponding MIDAS template for the file. (For more information, refer to the MIDAS Reference Guide.)

Two sets of typical CREATK dialog, generated for INDEXED and DAM files, are shown below.

All user responses are underlined.

DIALOG FOR INDEXED FILE

Prompt	Response	Remarks
OK.	**CREATK**	
[CREATK rev 17.2]		
MINIMUM OPTIONS?	**YES**	If minimum options is selected, all index level keys will have the same length as the full key for the last index level. The primary key will be stored with the data and not in the index entries of the secondary indices. All index blocks will default to a length of 440 words.
FILE NAME? [volume name>ufd passwd [disk>] filename		Volume name>UFD: specifies the name of the disk and the User File Directory (UFD) on which the file is to be created. Filename is the user assigned filename.
NEW FILE?	**YES**	
DIRECT ACCESS?	**NO**	For a new indexed file.
DATA SUBFILE QUESTIONS		Prime Index or Record Key.
KEY TYPE:	{ **A** / **B** }	**A** specifies an ASCII key. **B** specifies a binary key.
KEY SIZE = :	{ **B** / **W** } **number**	**B number** defines the number of bytes for an ASCII key or bits for a binary key. **W number** defines the number of words for A or B key. For example, if there are 2 characters in the key, **number** should be 16 bits, 2 bytes, or 1

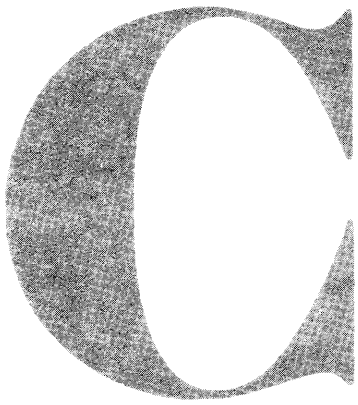
		word, depending on the key type. The maximum key size for an indexed file is 256 bits, 32 characters (bytes), or 16 words.
DATA SIZE = :	**number**	**Number** is the number of words for a data record, where number equals the record length divided by 2 plus the remainder factor of 1 if it applies.
SECONDARY INDEX		This section is repeated for each alternate record key.
INDEX NO.?	{ **1-5** / **(CR)** }	The numeric variable is the number of the alternate record key. Carriage return **(CR)** will exit from CREATK, specifying no alternate indexes.
DUPLICATE KEYS PERMITTED?	{ **YES** / **NO** }	**YES** allows the data in this key field to be duplicated. **NO** indicates that if the data in the key field is duplicated the file will not be updated and the INVALID KEY clause or the use DECLARATIVES section will be activated.
KEY TYPE:	{ **A** / **B** }	
KEY SIZE = :	{ **B** / W } **number**	Enter the size of the alternate key.
USER DATA SIZE = :	**0** **(CR)**	No data may be entered for secondary keys. The response must be 0, **(CR)**, or 0 (CR). Either option will return the user to the prompt INDEX NO.? above, from which he may exit from CREATK, or continue with alternate key specifications.

DIALOG FOR DAM FILE

Prompt	Response	Remarks
OK,	**CREATK**	
[CREATK rev 17.2]		
MINIMUM OPTIONS?	**YES**	If minimum options are selected, all index level keys will have the same length as the full key for the last index level. The primary key will be stored with the data and not in the index entries of the secondary indices.
FILE NAME? [volume name>ufd passwd ldisk>] filename		Same as the first dialog.
NEW FILE?	**YES**	
DIRECT ACCESS?	**YES**	For a new relative file.
DATA SUBFILE QUESTIONS		
KEY TYPE:	{ **A** / **B** }	
KEY SIZE = :	{ **B** / **W** } **number**	Enter the size of key; see the first dialog. The maximum key size for a relative file is 48 bits, six characters (bytes), or three words. In sequential mode, key must always be specified at maximum size.
DATA SIZE = :	**number**	Same as the first dialog.
NUMBER OF ENTRIES TO ALLOCATE?	**number**	**Number** is the number of entries to allocate in the new MIDAS. Entries are numbered 1-n inclusive; any reference outside this range results in an error.
INDEX NO.?	**(CR)**	This concludes template creation and returns to command level.

Note

If an invalid response is entered by the user, the question (prompt) will be repeated.



Reference tables

WHAT IS IN THIS APPENDIX

The following tables are included in this appendix.

- COBOL Verb Index
- COBOL Reserved Words
- ASCII Character Set
- File Status Key Definitions
- Permissible Input/Output Statements
- Permissible Moves
- Numeric Conversion Tables

Table C-1. COBOL Verb Index

VERB	CATEGORY (Depending on Format)	SPECIAL APPLICATIONS
ADD COMPUTE DIVIDE MULTIPLY SUBTRACT	Arithmetic	
COPY ENTER USE	Compiler Directing	Inter-program Communication
ADD COMPUTE DELETE DIVIDE IF (a) MULTIPLY		File Handling
READ	Conditional	File Handling
REWRITE		File Handling
START		File Handling
STRING SUBTRACT UNSTRING		
USE		File Handling
WRITE		File Handling
INSPECT MOVE STRING UNSTRING	Data Movement	
STOP	Ending	
ACCEPT		Terminal Input
CLOSE		File Handling
DELETE		File Handling
DISPLAY		Terminal Output
EXHIBIT		Terminal Output

OPEN	Input-Output	File Handling
READ		File Handling
REWRITE		File Handling
START		File Handling
STOP		
USE		File Handling
WRITE		File Handling
CALL	Inter-program Communicating	Inter-program Communication
RELEASE		
RETURN	Ordering	
SORT		
ALTER		
CALL		Inter-program Communication
EXIT		
EXIT PROGRAM	Procedure Branching	Inter-program Communication
GO TO		
PERFORM		
SEARCH	Table Handling	
SET		
READY TRACE	TRACE MODE	Debugging
RESET TRACE	Directing	Debugging

Note

(a)-IF is a verb in COBOL, although not a verb in the grammatical sense in English.

Table C-2. COBOL Reserved Words

ACCEPT	HIGH-VALUE	REFERENCES
ACCESS	HIGH-VALUES	RELATIVE
ADD	I-O	RELEASE
ADVANCING	I-O-CONTROL	REMARKS *
AFTER	ID *	REMOVAL
ALL	IDENTIFICATION	RENAMES
ALPHABETIC	IF	REPLACING
ALTER	IN	RERUN
ALTERNATE	INDEX	RESERVE
AND	INDEXED	RESET
ARE	INITIAL	RETURN
AREA	INPUT	RESTART-FILE *
AREAS	INPUT-OUTPUT	REVERSED
ASCII *	INSPECT	REWIND
ASSEMBLER *	INSTALLATION	REWRITE
ASSIGN	INTO	RIGHT
AT	INVALID	ROUNDED
AUTHOR	IS	RUN
BEFORE	JUST	SAME
BLANK	JUSTIFIED	SEARCH
BLOCK	KEY	SECTION
BY	LABEL	SECURITY
CALL	LEADING	SELECT
CHARACTER	LEFT	SENTENCE
CHARACTERS	LENGTH	SEPARATE
CLOSE	LESS	SEQUENTIAL

COBOL
CODE
CODE-SET
COMMA
COMP
COMP-3 *
COMPUTATIONAL
COMPUTATIONAL-3 *
COMPUTE
CONFIGURATION
CONSOLE *
CONTAINS
COPY
CORR
CORRESPONDING
COUNT
CURRENCY
DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC
ELSE
END
ENTER
ENVIRONMENT
EQUAL
ERROR
EVERY
EXCEPTION
EXHIBIT *
EXIT
EXTEND
FD
FILE
FILE-CONTROL
FILE-ID *
FILLER
FIRST
FOR
FROM
GIVING
GO
GREATER
LINE
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES
MODE
MOVE
MT9 *
MULTIPLY
NAMED *
NATIVE
NEGATIVE
NEXT
NOT
NUMBER
NUMERIC
OBJECT-COMPUTER
OCCURS
OF
OFF
OFFLINE-PRINT *
OMITTED
ON
OPEN
OR
ORDS *
ORGANIZATION
OUTPUT
OVERFLOW
OWNER *
PAGE
PERFORM
PFMS *
PIC
PICTURE
POINTER
POSITION
POSITIVE
PRINTER *
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
PUNCH *
QUOTE
QUOTES
RANDOM
READ
READER *
READY *
RECORD
RECORDS
REDEFINES
REEL
SET
SIGN
SIZE
SORT
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
START
STATUS
STOP
STRING
SUBSCHEMA *
SUBTRACT
SYNC
SYNCHRONIZED
TABLE
TALLYING
TAPE
TERMINAL
THAN
THROUGH
THRU
TIME
TIMES
TO
TRACE *
TRAILING
UNCOMPRESSED *
UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING
VALUE
VALUES
VARYING
WHEN
WITH
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS
+
–
*
/
>
<
=

The Prime COBOL collating sequence conforms to the American Standard Code for Information Interchange (ASCII) collating sequence. The octal value associated with each character in the Prime computer is the basis for the sequence, where the characters are arranged in ascending value from top to bottom

Table C-3. ASCII Character Set

ASCII Character	PRIME REPRESENTATION Hexadecimal	Octal	Punched Cards
NUL (low value)	80	200	
(space)	A0	240	No punch
! (Exclamation)	A1	241	12-8-2
" (Quote)	A2	242	7-8
# (Number)	A3	243	8-3
$	A4	244	11-3-8
' (Apostrophe)	A7	247	5-8
(	A8	250	12-5-8
)	A9	251	11-5-8
*	AA	252	11-4-8
+	AB	253	12-6-8
, (Comma)	AC	254	0-3-8
– (Minus)	AD	255	11
. (Period)	AE	256	12-3-8
/ (Virgule, slash, stroke)	AF	257	0-1
0 (Zero)	B0	260	0
1	B1	261	1
2	B2	262	2
3	B3	263	3
4	B4	264	4
5	B5	265	5
6	B6	266	6
7	B7	267	7
8	B8	270	8
9	B9	271	9
: (Colon)	BA	272	8-2
; (Semicolon)	BB	273	11-6-8
<	BC	274	12-4-8
=	BD	275	6-8
>	BE	276	0-6-8
?	BF	277	0-7-8
@ (at)	C0	300	8-4
A	C1	301	12-1
B	C2	302	12-2
C	C3	303	12-3
D	C4	304	12-4
E	C5	305	12-5
F	C6	306	12-6
G	C7	307	12-7
H	C8	310	12-8
I	C9	311	12-9
J	CA	312	11-1
K	CB	313	11-2

L	CC	314	11-3
M	CD	315	11-4
N	CE	316	11-5
O	CF	317	11-6
P	D0	320	11-7
Q	D1	321	11-8
R	D2	322	11-9
S	D3	323	0-2
T	D4	324	0-3
U	D5	325	0-4
V	D6	326	0-5
W	D7	327	0-6
X	D8	330	0-7
Y	D9	331	0-8
Z	DA	332	0-9
a	E1	341	
b	E2	342	
c	E3	343	
d	E4	344	
e	E5	345	
f	E6	346	
g	E7	347	
h	E8	350	
i	E9	351	
j	EA	352	
k	EB	353	
l	EC	354	
m	ED	355	
n	EE	356	
o	EF	357	
p	F0	360	
q	F1	361	
r	F2	362	
s	F3	363	
t	F4	364	
u	F5	365	
v	F6	366	
w	F7	367	
x	F8	370	
y	F9	371	
z	FA	372	
0 (+zero)	FB	373	12-0
0 (–zero)	FD	375	11-0
DEL (High Value)	FF	377	

Note

Characters with no Punched Cards code are not supported for punched card entry.

Table C-4. File Status Key Definitions

FILE ORGANIZATION	STATUS KEY 1	STATUS KEY 2
SEQUENTIAL	0-Successful completion	0-No further information
	1-End of file (a)	0-No further information
	3-Permanent I/O error (b)	0-No further information
		4-Boundary violation (c)
RELATIVE	0-Successful completion	0-No further information
	1-End of file (a)	0-No further information
	2-Invalid key	1-Sequence error (f)
		3-No record found (e)
		4-Boundary violation (c)
	3-Permanent I/O error (b)	0-No further information
	9-Implementor defined	0-Locked record (g)
		1-Unlocked record (h)
		2-Record already exists on Data Base
		6-Space relative key contains larger value than used when CREATK was used.
		9-System error, call analyst
INDEXED	0-Successful completion	0-No further information
	1-End of file (a)	0-No further information
	2-Invalid key	1-Sequence error (f)
		2-Duplicate key (d)
		3-No record found (e)
		4-Boundary violation (c)
	3-Permanent I/O error (b)	0-No further information
	9-Implementor defined	0-Locked record (g)
		1-Unlocked record (h)
		2-Value in key already in the data base and duplicates not specified when CREATK was run. (d)
		3-Indices specified in the program do not match the indices specified when CREATK was run.
		4-MIDAS multi-user concurrency bug. Index file has been altered by another user. Pointers are bad.
		5-Index does not match size used on creation.
		6-The disk is full.
		9-System error, call analyst.

Note

(a)-End of file — A READ statement was unsuccessful because there was no logical next record in the file.

(b)-Permanent I/O error	An I/O statement was unsuccessful because of an I/O error, such as a data check, parity error, or transmission error. For sequential file only, a boundary violation.
(c)-Boundary Violation	Attempt was made to read or write beyond the externally defined boundaries of a file. Disk space full.
(d)-Duplicate key	Attempt was made to write (or, for an indexed file, rewrite) a record which would create a duplicate key in the file. For an indexed file, when the file status is 92, a duplicate condition exists if the key value of the current key of reference is equal to the value of that same key in the next record within the current key of reference.
(e)-No Record Found	Attempt was made to access a record, identified by key, but the record does not exist in the file.
(f)-Sequence error	For relative file: trying to write beyond the predefined boundaries of the file. For an indexed file: trying to write a record containing a key which already exists in the file.
(g)-Locked record	The record is locked and being updated by another program.
(h)-Unlocked record	The record is not locked by a READ prior to a REWRITE

Table C-5. Permissible Input/Output Statements-OPEN Statements vs. Access Modes

File Organization	File Access Mode	Procedure Statement	OPEN Option in Effect: INPUT	OUTPUT	I-O	EXTEND
Sequential	SEQUENTIAL	READ	X		X	
		WRITE		X		X
		REWRITE			X	
	SEQUENTIAL	READ				
		WRITE				
		REWRITE				
		START				
		DELETE				
Indexed Relative	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	

Table C-6. Permissible Moves

SENDING DATA ITEM \ RECEIVING DATA ITEM	ALPHABETIC	BINARY	ALPHANUMERIC EDITED	NUMERIC INTEGER NUMERIC NON-INTEGER	NUMERIC EDITED	ALPHANUMERIC
ALPHABETIC	X		X			X
BINARY		X		X	X	X(1)
ALPHANUMERIC EDITED	X		X(3)			X
NUMERIC INTEGER		X	X	X	X	X(2)
NUMERIC NON-INTEGER				X	X	
NUMERIC EDITED			X(3)			X(3)
ALPHANUMERIC	X		X	X	X(4)	X

Note

1. If receiving operand length L is less than or equal to 18, target Picture 9(L) is assumed. Otherwise, the MOVE is disallowed.
2. The source is converted to DISPLAY form with separate trailing sign (blank for positive), then moved as a character string source, subject to truncation or blank padding depending on receiving its length.
3. The source is considered as a character string.
4. If source length L is less than or equal to 18, source Picture 9(L) is assumed. Otherwise, the MOVE is disallowed.

Table C-7. Hexadecimal and Decimal Conversion

	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
	0	0	0	0	0	0	0	0
	1	4096	1	256	1	16	1	1
	2	8192	2	512	2	32	2	2
	3	12288	3	768	3	48	3	3
	4	16384	4	1024	4	64	4	4
	5	20480	5	1280	5	80	5	5
	6	24576	6	1536	6	96	6	6
	7	28672	7	1792	7	112	7	7
	8	32768	8	2048	8	128	8	8
	9	36864	9	2304	9	144	9	9
65,536	A	40960	A	2560	A	160	A	10
	B	45056	B	2816	B	176	B	11
	C	49152	C	3072	C	192	C	12
	D	53248	D	3328	D	208	D	13
	E	57344	E	3584	E	224	E	14
	F	61440	F	3840	F	240	F	15
	16^3		16^2		16^1		16^0	

Table C-8. Octal and Decimal Conversion

	OCT	DEC	OCT	DEC	OCT	DEC	OCT	DEC	OCT	DEC
	0	0	0	0	0	0	0	0	0	0
	1	4096	1	512	1	64	1	8	1	1
	2	8192	2	1024	2	128	2	16	2	2
	3	12288	3	1536	3	192	3	24	3	3
	4	16384	4	2048	4	256	4	32	4	4
32,768	5	20480	5	2560	5	320	5	40	5	5
	6	24576	6	3072	6	384	6	48	6	6
	7	28672	7	3584	7	448	7	56	7	7
	8^4		8^3		8^2		8^1		8^0	

Table C-9. Hexadecimal Addition Table.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Note

All numbers in hexadecimal.

D

COBOL symbols

COBOL SYMBOLS

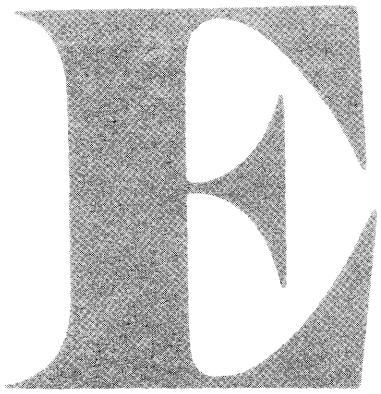
PUNCTUATION SYMBOLS — Used to punctuate program entries.	
. period	1. Used to terminate entries. Usually required. 2. Used to signify the decimal in numeric literals.
, comma	1. Used to separate operands or clauses in a series. Usually optional. 2. "European" notation for the decimal in numeric literals.
; semicolon	Used to separate operands or clauses in a series. Usually optional.
" quotation mark ' apostrophe	Used to enclose non-numeric literals.
CODING SYMBOLS — Campiler symbols.	
* asterisk	Denotes an explanatory comment line when inserted in column 7 of a source program line.
/ virgule	Denotes a skip to the top of a new page during a compiler listing. This is coded in column 7 of a source program line.
- hyphen	Denotes a continuation-line for non-numeric literals when coded in column 7 of a source program line.
SIGN SYMBOLS/UNARY OPERATORS — Found in numeric literals and arithmetic formulas.	
+ positive	1. Used as a sign character in the high-order (left-most) position of a numeric literal. 2. As a unary operator, the effect of multiplication by numeric literal + 1.
− negative	1. Used as a sign character in the high-order (left-most) position of a numeric literal. 2. As a unary operator, the effect of multiplication by numeric literal − 1.

COBOL SYMBOLS

ARITHMETIC SYMBOLS — Found in arithmetic formulas.	
+ plus	Addition.
− minus	Subtraction
* asterisk	Multiplication
/ virgule	Division
= equal	"Make equal to"
( ) parenthesis	Used to enclose expressions to control the sequence in which they are performed.
CONDITION SYMBOLS — Used in conditional test expressions.	
= equal	Denotes "is equal to".
> greater than	Denotes "is greater than".
< less than	Denotes "is less than"
( ) parenthesis	Used to enclose expressions to control the sequence in which conditions are evaluated.
EDITED ITEM OR EDIT SYMBOLS — Used in edited item picture clauses	
. decimal point (insertion character)	Used to insert an actual decimal in the indicated position of an edited item.
, comma (insertion character)	Used to insert a comma in the indicated position(s) of an edited item. (May be used in conjunction with floating characters.)
$ dollar sign (floating character)	Used to float an actual dollar sign (from left to right) in an edited item, so that exactly one $ is developed immediately to the left of the most significant nonzero digit in any position where the symbol has been used.

COBOL SYMBOLS

REPORT ITEM OR EDIT SYMBOLS (continued . . .)	
= equal (insertion character)	Used to insert an actual equal symbol in the indicated position of an edited item.
/ virgule (insertion character)	Used to insert an actual slash in the indicated position(s) of an edited item.
* asterisk (replacement character)	Used to replace leading zeros with an actual asterisk. Each * represents a digit position in an edited item.
+ plus − minus or dash (fixed sign control, or floating character)	1. Used as a fixed sign control character in the low-order (right-most) position of an edited item picture. The symbol does not replace a digit position. 2. Used to float an actual plus or minus character (from left to right) in an edited item, so that exactly one + or − is developed immediately to the left of the most significant nonzero digit in any position where the symbol has been used.
B letter B (insertion character)	Used to insert blanks in the indicated position(s) of an edited item.
0 ZERO (insertion character)	Used to insert zero(s) in the indicated position(s) of an edited item.
Z ZED (replacement character)	Used to replace leading zero(s) with blank(s) in the indicated position(s) of an edited item.
CR credit (fixed sign control character)	Used as a fixed sign control character in the low-order (right-most) position of an edited item picture. It occupies 2 character positions in the picture.
DB debit (fixed sign control character)	Used as a fixed sign control character in the low-order (right-most) position of an edited item picture. It occupies 2 character positions in the picture.
P letter P (decimal scaling character)	Used to position the assumed decimal point away from the number; e.g., an item whose actual value is 25 will be treated as 25000 if its picture is 99PPPV.



Error messages

TYPES OF ERROR MESSAGES

This appendix contains the following categories of errors:

- COMPILE-TIME ERROR MESSAGES
- COMPILE-TIME WARNING MESSAGES
- RUN-TIME ERROR MESSAGES

Error messages appear alphabetically within each category.

COMPILE-TIME ERROR MESSAGES

▶ ")" REQUIRED AFTER SUBSCRIPTS.

The close parenthesis following a subscript has been omitted. Correct the coding and recompile.

▶ AREA-A VIOLATION; RESUMES AT NEXT PARAGRAPH/ SECTION/DIVISION/VERB.

Data was ignored.

▶ BLANK WHEN ZERO IS DISALLOWED.

The BLANK WHEN ZERO clause is not permitted here. Use zero suppression or other editing functions as indicated. Recompile.

▶ CONDITIONAL I/O STATEMENT DISALLOWED WITHIN "IF".

Implied conditional such as READ, IF A=B READ FILE AT END is invalid.

▶ DATA DIVISION ASSUMED.

DATA DIVISION omitted in application program; insert DATA DIVISION and recompile.

▶ DELETE/START NOT VALID FOR THIS FILE.

See Table C-5, OPEN Statements and Access Modes. Correct coding, recompile, i.e., the DELETE/START statement must not be used for a sequential file.

▶ DISPLAY LIMITED TO 72 ON CONSOLE, 132 ON PRINTER.

Cannot display more than 72 characters on most terminals or print more than 132 characters per line for print files.

▶ ERRONEOUS ASSIGNMENT.

ASSIGN TO device clause does not match FD; correct and recompile.

▶ ERRONEOUS FILE-NAME.

SELECT file-name does not match FD file-name.

ERRONEOUS QUALIFICATION; LAST DECLARATION USED.

Data-name not unique, needs qualification to the group level.

ERRONEOUS SELECT-SENTENCE; RESUMES AT NEXT SELECTOR AREA-A.

The flagged SELECT was ignored because the proper SELECT file-name, Key word ASSIGN, or device-name was missing. Correct errors, recompile.

ERRONEOUS SUBSCRIPTING; STATEMENT DELETED.

Refer to rules governing subscripting, Section 7 and subscripting OCCURS clause. Correct errors, recompile.

ERROR IN USING SORT, RELEASE, RETURN.

- SD has not been defined for the sort-file.
- Sort-file has no corresponding SELECT clause.
- Sort keys are not in SD description.
- RELEASE has not been used in the input procedure.
- RETURN has not been used in the output procedure.

EXCESSIVE OCCURS NESTING IS IGNORED.

Only up to three levels of OCCURS clause are allowed.

FD-VALUE IGNORED SINCE LABELS OMITTED.

VALUE OF FILE-ID or OWNER ID specified with labels omitted assumed. Correct and recompile.

FILE SECTION ASSUMED.

Missing FILE SECTION in DATA DIVISION. Insert and recompile.

GROUP SIZE > 32,766; SET TO 1.

Group and/or record size exeeds maximum. Reduce to less than 32,766 bytes. Correct and recompile.

ILLEGAL MOVE OR COMPARISON IS DELETED.

Check rules governing IF and MOVE statements. Correct errors, recompile.

IMPROPER FILENAME IGNORED.

FD entry has no corresponding SELECT statement. Correct and recompile.

IMPROPER OCCURS COUNT IGNORED.

OCCURS specification is greater than 32,787. Check rules for OCCURS clause; correct and recompile.

IMPROPER REDEFINITION IGNORED.

Check rules for REDEFINES clause. Correct errors; recompile.

INCOMPLETE/TOO LONG STATEMENT DELETED.

Check syntax; correct and recompile.

▶ INVALID BLOCKING IS IGNORED.

BLOCK CONTAINS clause exceeds maximum supported; correct and recompile.

▶ INVALID RECORD SIZE(S) IGNORED.

RECORD CONTAINS clause in error; correct and recompile.

▶ ITEM ASSUMED TO BE BINARY.

Elementary item with no PICTURE clause assumed binary. Check coding.

▶ KEY DECLARATION OF THIS FILE IS INCORRECT.

KEY IS clause in SELECT does not match FD description for indexed files, or Working-Storage description for relative files.

▶ KEY MUST BE DECIMAL OR CHARACTER ITEM, MAX. 255 BYTES. STATEMENT DELETED.

Key specification in error. Maximum is 255 bytes for indexed files, 6 bytes for relative files. Correct and compile.

▶ LABEL RECORDS OMITTED ASSUMED FOR UNIT-RECORD FILE.

Check LABEL clause, change to LABEL RECORD OMITTED.

▶ LABELS ASSUMED FOR DISK FILE.

Check LABEL clause, change to LABEL RECORD STANDARD.

▶ LEVEL 01 ASSUMED.

Check previous statements; correct and recompile.

▶ MISORDERED/REDUNDANT SECTION PROCESSED AS IS.

Correct coding sequence; recompile.

▶ NAME OMITTED; ENTRY BYPASSED.

Unrecognizable data-name, not defined in Data Division, is a syntax error. Correct and recompile.

▶ NON-UNIQUE SUBSCRIPT; LAST DECLARATION USED.

Non-unique data-name. Qualification is required; recompile.

▶ OCCURS DISALLOWED AT LEVEL 01.

Correct and recompile.

▶ PARAGRAPH DECLARATION REQUIRED HERE.

A paragraph-name is required as the first item in the Procedure Division; recompile.

▶ PERIOD ASSUMED AFTER PROCEDURE-NAME DEFINITION.

Period missing after a paragraph-name or a section-name. Correct and recompile.

▶ PICTURE IGNORED FOR INDEX ITEM.

PICTURE disallowed on USAGE IS INDEX. Correct and recompile.

RECORD MIN/MAX DISAGREES WITH RECORD CONTAINS; LATER SIZES PREVAIL.

Either delete RECORD CONTAINS clause and use default, or use the proper record size. Recompile.

REDUNDANT CLAUSE IGNORED.

Remove and recompile.

REDUNDANT FD.

Non-unique file-name in the same program.

"SECTION" ASSUMED HERE.

Insert SECTION and recompile.

SINGLE-SPACING ASSUMED DUE TO IMPROPER ADVANCING COUNT

WRITE BEFORE/AFTER ADVANCING count is greater than 62. Correct and recompile.

SOURCE BYPASSED UNTIL NEXT FD/SECTION.

This relates to previous error. Correct previous error(s), recompile.

STATEMENT DELETED DUE TO ERRONEOUS SYNTAX.

Illegal, non-standard syntax; correct and recompile.

STATEMENT DELETED DUE TO OMISSION OF RELATIONAL SYMBOL.

Correct and recompile.

STATEMENT DELETED DUE TO NON-NUMERIC OPERAND.

Incompatible data types must be reconciled; recompile.

STATEMENT DELETED; OPERAND IS NOT A FILE-NAME.

Illegal attempt to READ, WRITE, OPEN or CLOSE an undefined file. Correct syntax and recompile.

UNIT-RECORD FILE BLOCKING IS IGNORED.

Device and BLOCK CONTAINS clause are incompatible.

UNRECOGNIZABLE ELEMENT IS IGNORED.

Correct and recompile.

UNRESOLVED PROCEDURE-NAME; STATEMENT DELETED.

Illegal attempt to GO TO an undefined procedure-name. Correct and recompile.

USING-LIST LEVELS MUST BE 01/77.

Using-lists in subprograms must begin on 01 or 77 level to be forced on word boundaries. Correct and recompile.

VALUE CLAUSE IGNORED.

Delete and recompile.

▶ VALUE DELETED DUE TO TYPE CONFLICT.

PICTURE and VALUE disagree in size. Correct and recompile.

▶ VALUE DISALLOWED DUE TO OCCURS/REDEFINES.

Remove VALUE clause and recompile.

▶ VALUE DISALLOWED IN FILE/LINKAGE SECTION.

Remove VALUE clause and recompile.

▶ VARYING ITEM MAY NOT BE SUBSCRIPTED.

Correct and recompile.

COMPILE-TIME WARNING MESSAGES

▶ "COMP" IGNORED DECIMAL ITEM.

COMP has been specified, although the item appears to be decimal; the compiler is ignoring the COMP designation. Results may be incorrect. Determine the correct specification and recompile.

▶ DATA RECORDS CLAUSE WAS INACCURATE.

The DATA RECORDS clause does not agree with record description entries for the file. Correct and recompile.

▶ FILE NEVER CLOSED.

Include a CLOSE statement for the file, recompile.

▶ FILE NEVER OPENED.

Include an OPEN statement for the file, recompile.

▶ INCONSISTENT READ USAGE.

OPEN statement and USAGE do not agree. Illegal attempt to write to an input file.

▶ INCONSISTENT WRITE USAGE.

OPEN statement and USAGE do not agree. Illegal attempt to read an output file.

▶ ITEM IS UNSIGNED.

The item in this statement is unsigned, but appears to require sign designation. Results may be indeterminate.

▶ LITERAL TRUNCATED TO ITEM SIZE.

- VALUE OF FILE-ID has been specified by a literal greater than 8 characters.
- OWNER IS has been specified by a literal greater than 6 characters.
- The size of the literal in a VALUE clause is greater than the size defined in the PICTURE clause.

▶ MOVE IS DONE WITHOUT CONVERSION.

Data respresentation does not agree. Conversion will not occur; results are indeterminate.

▶ PERIOD ASSUMED ABOVE.

Statement syntax suggests a period; one has been generated by the compiler.

▶ NO CORRESPONDENCE FOUND.

Check rules for MOVE CORRESPONDING. Correct and recompile.

RUN-TIME ERROR MESSAGES

The general format for run-time I/O errors generated by a COBOL program is:

```
KI/DA FILE SYSTEM ERROR n, FILE-STATUS CODE f

FILE-ID: file-id  OWNER-ID: owner-id  DEVICE: device-name

FATAL RUN-TIME I-O ERROR (C$ER)
ER!
```

The first line of the message is omitted unless the error was caused by an indexed or relative I/O operation which involved a call to the MIDAS file system. If printed, **n** represents the error code returned from MIDAS. For a complete dissussion of MIDAS error messages, refer to The MIDAS Reference Guide. Further, **f** is the COBOL file-status code, as defined in this manual.

The diagnostic message is one-line which briefly describes the probable cause of the error. Most of the time the message will point directly to the problem. A list of diagnostics and further explanations are provided below.

The next line identifies the file on which the error occurred. Information printed includes **file-id** and **owner-id**, if specified, and **device-name (specified in SELECT clause)**.

A list of the COBOL run-time I/O error messages follow.

▶ ATTEMPTED DELETE FROM UNOPENED FILE

The user attempted to delete a record from an unopened file.

▶ ATTEMPTED READ FROM ILLEGAL DEVICE

The user attempted to read a record from the printer.

▶ ATTEMPTED READ FROM UNOPENED FILE

The user attempted to read a record from an unopened or a write-only file.

▶ ATTEMPTED REWRITE TO NON-DISK FILE

The user attempted to rewrite a record to a non-disk file (a file not assigned to Prime File Management System).

▶ ATTEMPTED REWRITE TO UNOPENED FILE

The user has attempted to rewrite a record to an input-only or an unopened file.

▶ ATTEMPTED START ON UNOPENED FILE

The user program executed a START statement on an unopened file.

▶ ATTEMPTED WRITE TO UNOPENED FILE

The user attempted to write a record to an unopened or a read-only file.

END OF FILE ENCOUNTERED

An EOF mark was encountered on a sequential READ statement.

ERROR ADDING SECONDARY INDEX, UNABLE TO DELETE PRIMARY

An error occurred adding a secondary index to an index file on a WRITE statement. When the error was noticed by the COBOL run-time package, an attempt was made to remove the primary index entry which failed. This error is always fatal and may indicate a problem with the MIDAS file structure or the COBOL run-time package.

ERROR PROCESSING DELETE STATEMENT

An error occurred attempting to delete a record from an indexed or a relative file.

ERROR PROCESSING START STATEMENT

An unexpected error occured while executing a START statement on an indexed or relative file.

ERROR UNLOCKING RECORD

A MIDAS error occurred (from UPDAT$) in an attempt to unlock a record.

FILE READ ERROR

General message indicating a sequential file read error.

FILE REWRITE ERROR

General message indicating a sequential file re-write error.

FILE WRITE ERROR

General message indicating a sequential file write error.

NO READ PRIOR TO DELETE

A READ statement must be executed prior to a DELETE on an indexed or relative file in sequential access mode.

NO READ PRIOR TO REWRITE

A READ statement must be executed prior to a REWRITE when an indexed or relative file is used in sequential access mode.

NO UNITS AVAILABLE

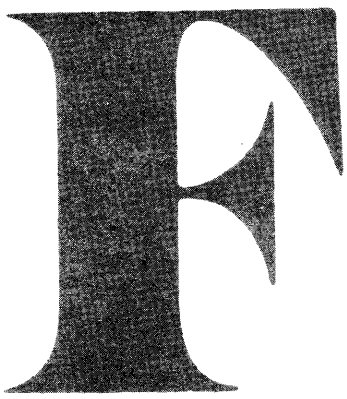
All available file units are in use. Note that units 13-16 are reserved for use by MIDAS and FORMS.

REDUNDANT OPEN ATTEMPTED

The program tried to open a file which was already open.

SEQUENTIAL WRITE TO RANDOM FILE OPENED IN I/O MODE

Attempt to use the sequential WRITE statement on a file opened in I/O mode for random access is not permitted.



Expanded listing

EXPANDED LISTING

In 64V mode (Prime 350 and up), COBOL can optionally generate an expanded listing following the errors and warnings section in the listing file. The expanded listing is fairly 'PMA-like', easily readable, and is obtained by employing the mnemonic parameter -EXPLIST.

Example:

```
COBOL program-name -EXPLIST
```

For the expanded listing, instead of using source code identifiers, Prime COBOL uses machine-generated labels in the listing. The general format of these labels is:

< TYPE > $XXXX[±N character offset]

where

XXXX	is the hexadecimal identifier.
TYPE	is the label type.

Label types fall into the following category:

A	Paragraph or section
B	Alter or perform indirect word
C	Perform count variable
D	Decimal constant
E	Picture string (const)
F	Character string (const)
G	Generate label for branch instruction
H	Passed parameter
S	Generate label - any usage allowed
Y	File control block
Z	File buffer

A description of these labels is shown below:

A$XXXX - Paragraph or section label

Will appear in expanded code associated with the previous source line.

Note

When TRACE MODE is used, code will exist to accomplish paragraph and section name displays.

B$XXXX - Alter or perform indirect

Will be used to key a return for a PERFORM statement. If when examined, the variable contains some non-zero value, a branch to that non-zero location is performed.

C$XXXX - Perform count variable

Used to contain value for perform loop.

Note

Stored as single precision integer for maximum value of 32767.

D$XXXX - Decimal constant

1. Item is placed in literal pool for explicit numeric constant references.
2. Item is placed in literal pool for implicit numeric constant references.
   - Condition names with value associated with numeric constant.

Note

All numeric constants in literal pool are examined prior to the addition of a new numeric constant. This prevents duplicates.

E$XXXX - Edit picture string

This item is generated for use with edit capability on data strings. Insertion characters, zero suppression, etc., are represented by this string.

F$XXXX - Character string constant

1. All explicit character string constants are placed in literal pool.
2. All implicit character string constants:
   - Condition names may generate a reference to a character constant.
   - An EXHIBIT Statement.
   - Entering a new paragraph with TRACE MODE set.

G$XXXX - Generated label for branch

Produced by IF Statements to bypass code based on conditions results.

H$XXXX - Passed parameter

Produced for each item in Linkage Section.

S$XXXX - Generated label - no specific reason

Produced for branching code associated with a READY TRACE.

Y$XXXX - File control block

Each defined file will generate an FCB.

Z$XXXX - File buffer

1. Each FCB will reference a file buffer area.
2. Size of file buffer may contain room for alternate buffers.

Other labels used are:

SB%	Stack base relative - used for temporary storage.
XB%	Temporary base relative - used for Linkage Section address.
WRKST$	Working-Storage
WSEXT$	Working-Storage extension, etc. under index, tallying and work area as needed by the compiler.

Example:

```
@003233:      001310      EAFA      1,Z$0027+72C
@003232:001000.000725L
```

The example above says: At relative location '3233 in the procedure area, EAFA 1, file buffer (ID=$0027 with a +72 character offset). Note that the word offset is '725 in the link frame.

An expanded listing example is presented below. It represents a portion of an actual listing for SAMPLE presented in Section 4.

For additional information pertaining to expanded code and the Program Statistics which follows it, see The Assembly Language Programmer's Guide and Section 2 of this manual respectively.

Example:

```
OK, COBOL SAMPLE -L TTY -EXPLIST
                  .
                  .
                  .
 (0001)           ID DIVISION.
 (0002)           PROGRAM-ID. SAMPLE.
                  .
                  .
                  .
 (0056)           BEGIN SECTION.
 (0057)           CREATE-FILE.
 (0058)               OPEN INPUT CARD-FILE.
 (0059)               OPEN OUTPUT PRINT-FILE, DIRECTORY-FILE.
 (0060)               WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
 (0061)           READ-NEXT.
 (0062)               READ CARD-FILE AT END GO TO LIST-DIRECTORY.
                  .
                  .
                  .
 (0076)               DISPLAY 'END TEST SEQUENTIAL READ AFTER A START'.
 (0077)               STOP RUN.
 (0078)           LIST.
 (0079)               WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
 (0080)           READ-NEXT-DIRECTORY-RECORD.
 (0081)               READ DIRECTORY-FILE NEXT RECORD AT END GO TO LIST-DONE.
 (0082)               MOVE DIRECTORY-RECORD TO PRINT-LINE.
 (0083)               WRITE PRINT-LINE.
 (0084)               GO TO READ-NEXT-DIRECTORY-RECORD.
 (0085)           LIST-DONE.
 (0086)               EXIT.


E X P A N D E D    L I S T I N G    F O R    --   SAMPLE
        000000:       001300            EAFA  0,WRKST$+2C
        000001: 001000.001413L
        000003:       001320            STFA  0,Y$002B+20C
        000004: 001000.000433L
        000006:       001300            EAFA  0,WRKST$
        000007: 001000.001412L
        000011:       001320            STFA  0,Y$002B+90C
        000012: 001000.000476L
        000014: 061432.000376L            PCL   =C$IN  ,*
                                A$0001 EQU   *
                                A$0002 EQU   *
```

```
* SOURCE LINE 58
        000016: 061432.000374L  A$0006 PCL   =C$OS  ,*
        000020: 001100.000650L         AP    Y$0013,S
        000022: 000100.000000F         AP    ='1,S
        000024: 000300.000000F         AP    F$80E2,SL
* SOURCE LINE 59
        000026: 061432.000374L         PCL   =C$OS  ,*
        000030: 001100.001076L         AP    Y$0001,S
        000032: 000100.000000F         AP    ='2,S
        000034: 000300.000025F         AP    F$80E2,SL
        000036:      02.000000F        LDA   ='0
        000037:      04.000467L        STA   Y$002B+76C
        000040: 061432.000372L         PCL   =C$OR  ,*
        000042: 001100.000421L         AP    Y$002B,S
        000044: 000300.000033F         AP    ='2,SL
* SOURCE LINE 60
        000046:        001300          EAFA  0,WRKST$+4C
        000047: 001000.001414L
        000051:        001310          EAFA  1,Z$0001
        000052: 001000.001204L
        000054: 001313.000204A         LFLI  1,132
        000056: 001303.000106A         LFLI  0,70
        000060:        001114          ZMV
        000061: 061432.000370L         PCL   =C$WS  ,*
        000063: 001100.001076L         AP    Y$0001,S
        000065: 000100.000000F         AP    S$0000,S
        000067: 000100.000000F         AP    ='102,S
        000071: 000300.000000F         AP    ='40000,SL
                                S$0000 EQU   *
* SOURCE LINE 62
        000073: 061432.000366L  A$000D PCL   =C$RS  ,*
        000075: 001100.000650L         AP    Y$0013,S
        000077: 000100.000000F         AP    S$0002,S
        000101: 000300.000000F         AP    S$0001,SL
        000103:      01.000000F S$0001 JMP   G$0014
        000104:      01.000000F S$0002 JMP   A$0013
                      .
                      .
                      .
* SOURCE LINE 77
        000233: 061432.000354L         PCL   =EXIT  ,*
* SOURCE LINE 79
        000235:        001300   A$0028 EAFA  0,WRKST$+4C
        000236: 001000.001414L
        000240:        001310          EAFA  1,Z$0001
        000241: 001000.001204L
        000243: 001313.000204A         LFLI  1,132
        000245: 001303.000106A         LFLI  0,70
        000247:        001114          ZMV
        000250: 061432.000370L         PCL   =C$WS  ,*
        000252: 001100.001076L         AP    Y$0001,S
        000254: 000100.000000F         AP    S$0009,S
        000256: 000100.000140F         AP    ='102,S
        000260: 000300.000072F         AP    ='40000,SL
                                S$0009 EQU   *
```

```
* SOURCE LINE 81
        000262: 061432.000352L  A$002B PCL    =C$RR  ,*
        000264: 001100.000421L         AP     Y$002B,S
        000266: 000100.000210F         AP     ='1,S
        000270: 000100.000000F         AP     S$000B,S
        000272: 000300.000000F         AP     S$000A,SL
        000274:     01.000000F  S$000A JMP    G$0016
        000275:     01.000000F  S$000B JMP    A$0039
* SOURCE LINE 82
        000276:       001300    G$0016 EAFA   0,Z$002B
        000277: 001000.000527L
        000301:       001310           EAFA   1,Z$0001
        000302: 001000.001204L
        000304: 001313.000204A         LFLI   1,132
        000306: 001303.000120A         LFLI   0,80
        000310:       001114           ZMV
* SOURCE LINE 83
        000311: 061432.000370L         PCL    =C$WS  ,*
        000313: 001100.001076L         AP     Y$0001,S
        000315: 000100.000000F         AP     S$000C,S
        000317: 000100.000257F         AP     ='102,S
        000321: 000300.000142F         AP     ='20001,SL
* SOURCE LINE 84
        000323:     01.000262  S$000C JMP    A$002B
* SOURCE LINE 86
        000324:       140040   A$0039 CRA
        000325:     13.000400L         IMA    B$0039
        000326:       100040           SZE
        000327:     41.000001A         JMP#   1,*
        000330: 061432.000354L         PCL    =EXIT  ,*
        000332:       000046
                        .
                        .
                        .
        000343:       305.316  F$8138
        000344:       304.240
                        .
                        .
                        .
        000352> 000000.000000E         IP     C$RR
        000354> 000000.000000E         IP     EXIT
        000356> 000000.000000E         IP     C$CR
                        .
                        .
                        .
        000400> 000000           B$0039
        000401>                  SAMPLE ECB    0,'352,'12,0,50
        001410>                  WSEXT$ EQU    *
        001412>                  WRKST$ EQU    *
        000421>       142255     Y$002B OCT    142255
        000422>       143311           OCT    143311
                        .
                        .
                        .
```

```
000473>       000000           OCT   0
000474>       000000           OCT   0
                .
                .
                .
000501>       004005           OCT   4005
000527>                 Z$002B DATA  81('  ')
000650>       144716    Y$0013 OCT   144716
000651>       142301           OCT   142301
000652>       152301           OCT   152301
                .
                .
                .
000654>       000000           OCT   0
                .
                .
                .
000756>                 Z$0013 DATA  80('  ')
001076>       000007    Y$0001 OCT   7
001077>       000007           OCT   7
                .
                .
                .
001160>       000000           OCT   0
001161>       000000           OCT   0
001204>                 Z$0001 DATA  132('  ')
```

P R O G R A M S T A T I S T I C S

Executable Code Size: 218 Words.
Constant Pool Size: 49 Words.
Total Pure Procedure Size: 267 Words.

Working-Storage Size: 74 Bytes.
Total Linkframe Size: 581 Words.

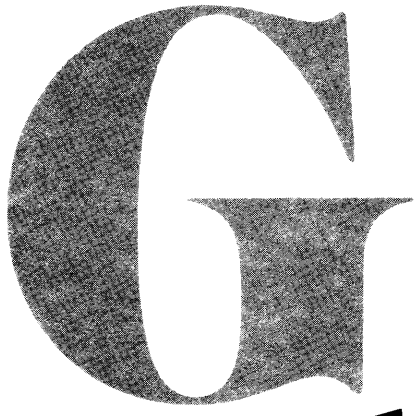
Stack Size: 51 Words.

Trace Mode: Off.

No Arguments Expected.

86 Source Lines.

No Errors, No Warnings, Prime V-Mode COBOL, Rev 17.2 <SAMPLE>



LABEL command

OVERVIEW OF LABEL

PRIMOS has an utility called LABEL which initializes magnetic tapes. LABEL writes either IBM (9-track EBCDIC or 7-track BCD) or ANSI (9-track ASCII) level 1 volume labels followed by dummy HDR1 and EOF1 labels. LABEL can also be used to read existing VOL1 and HDR1 labels.

ANSI labels are written in accordance with the American National Standards Institute standard ANSI X3.27-1978. IBM labels are written in accordance with IBM's specifications (IBM manual GC28-6680-5).

Any non-standard labels such as 7-track ASCII or user-defined labels *cannot* be read or written.

USING LABEL

To read existing labels type the command:

LABEL MTn [-TYPE type]

To write labels type the command:

LABEL MTn [-TYPE type] -VOLID vol [-OWNER own] [-ACCESS acc] [-INIT]

The arguments have the following meanings:

MTn is the tape drive where the tape to be labeled is located. **n** is a number between 0 and 7. This keyword is required and must be the first on the command line.

type is the type of label desired:

-TYPE A	9-track ASCII	(ANSI) (Default)
-TYPE B	7-track BCD	(IBM)
-TYPE E	9-track EBCDIC	(IBM)

vol is a 1-6 character string which uniquely identifies this tape reel. If less than 6 characters are specified, they are blank-padded on the right. The keywords "-VOLUME" or "VOL" may be substituted for the keyword "-VOLID".

own is 1-14 characters long for ANSI labels, 1-10 characters long for IBM labels. If less than 14 (or 10) characters is specified, they are blank-padded on the right. If this keyword is omitted, the default is the user's login name. The keyword "-OWN" may be substituted for the keyword "-OWNER".

acc is a single character defining access to this tape. ACCESS is not used by Prime software but is included for completeness. If it is omitted, it is left blank on ANSI labels. ACCESS is ignored for IBM labels.

INIT is necessary if the tape is brand new.

On read operations, LABEL prints out the volumne and owner ids, creation date, access (ANSI tapes only), and other information.

If LABEL successfully writes a label, the message "TAPE LABEL WAS WRITTEN SUCCESSFULLY" is displayed.

ERRORS USING LABEL

Improper use of the LABEL command causes an error message to be printed. These errors are the result of bad syntax in the LABEL command itself or a system magnetic tape I/O error.

Syntax errors

▶ ***DUPLICATE KEYWORD DETECTED

The same keyword was typed more than once.

▶ ***INVALID TAPE UNIT SPECIFIED

Something other than MT0-MT7 was typed.

▶ ***VOLUME ID SPECIFIED IS TOO LONG

The volume id cannot be longer than 6 characters.

▶ ***OWNER ID SPECIFIED IS TOO LONG

The owner id cannot be longer than 14 characters.

▶ ***INVALID LABEL TYPE SPECIFIED

Label type must be one of the characters "A", "E", or "B".

▶ ***NO MAGNETIC TAPE UNIT SPECIFIED

A magnetic tape unit is required.

▶ ***VOLUME ID WAS NOT SPECIFIED

When writing labels, a volume id is required.

▶ ***OWNER ID SPECIFIED IS TOO LONG FOR TYPES B OR E

The owner id for IBM labels cannot be longer than 10 characters.

▶ ***UNABLE TO WRITE TAPE LABEL ON THIS TAPE

A magnetic tape write error occurred and the label was not written.

▶ ***UNABLE TO READ TAPE LABEL ON THIS TAPE

A mag tape read error occurred and the label was not read.

▶ ***VOL1 LABEL ALREADY EXISTS

ANSI standards prohibit the re-writing of VOL1 labels.

▶ ***LABEL READ WAS NOT TYPE x

The label read was not of the type specified.

▶ ***LABEL OPERATION ABORTED

Any one of the prededing four errors aborts.

▶ ***ACCESS IGNORED FOR IBM LABELS (WARNING ONLY)

This is a warning only - processing continues.

▶ ***UNRECOGNIZED KEYWORD. string (CMDL$A)

An invalid keyword (string) appeared on the command line.

System errors

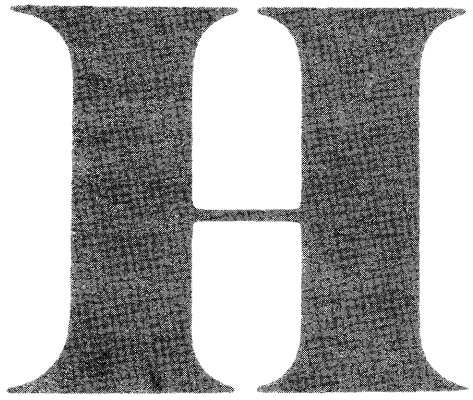
MTn NOT ASSIGNED	Tape drive must be ASSIGNed before using LABEL
SUBR EOF	END-OF-FILE on the magnetic tape
SUBR EOT	END-OF-TAPE
SUBR MTNO	Tape drive is not operational
SUBR PERR	Parity error on the tape drive
SUBR HERR	Tape drive hardware error
SUBR BADC	LABEL improperly called mag tape subroutines

In the above errors, SUBR is the name of the magnetic tape subroutine that reported the error. See the PRIMOS Subroutines Reference Guide for more information regarding these errors.

HELP FACILITY

The command **LABEL -HELP** causes LABEL to print out an abbreviated description of the command on the terminal.

For a complete description of tape labels and their use, refer to the IBM publication GC28-6680, OS TAPE LABELS and the ANSI publication X3.27-1969, "American National Standard Magnetic Tape Labels for Information Interchange".



COBOL system files

SYSTEM FILES

To utilize COBOL, the following files must be available on the system in the UFD's specified:

UFD	FILE-NAME	
CMDNC0	COBOL	(shared COBOL compiler)
	NCOBOL	(non-shared COBOL compiler)
SYSOVL	C$$COD	(code generator)
LIB	NVCOBLB	(non-shared library)
	VCOBLB	(shared library)
	PFTNLB	(pure FORTRAN library)
	IFTNLB	(impure FORTRAN library)
SYSTEM	C02016	(shared compiler segment)

VCOBLB Library

The VCOBLB library contains the following common COBOL subroutines.

C$ADAT	Returns current data in format YMMDD
C$ADAY	Returns Julian date in format YYDDD
C$ATIM	Returns current time in format HHMMSSFF H = Hour M = Minutes S = Seconds F = Hundredths of seconds
C$CA	Close opened files
C$INSP	INSPECT statement
C$UNS1/C$UNS2	UNSTRING statement
C$STR1/C$STR2/ C$STR3	STRING statement
CIN/CIN1/ C$IN2	File assignment initialization
C$OS	Open sequential file
C$CS	Close sequential file
C$RS	Read sequential file
C$UN	Unlock an indexed or relative file entry
C$XS	Rewrite sequential file
C$WS	Write sequential file
COI/COR	Open indexed/relative file
CCI/CCR	Close indexed/relative file
CRI/CRR	Read indexed/relative file
CWI/CWR	Write indexed/relative file

CXI/CXR	Rewrite indexed/relative file
CSI/CSR	Start indexed/relative file
CDI/CDR	Delete indexed/relative file
C$AU	Find next available file unit
C$NCLT	Numeric class test
C$KE	Error processing
C$ER	Error processing
CER	Error processing
	Note
C$PRTN	Calling COBOL subprograms, non-shared library only

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z