PRIME

THE COBOL PROGRAMMER'S GUIDE PDR3056

P/N MAN3251-001

PRIME SOFTWARE DOCUMENTATION

HIGH LEVEL LANGUAGE PROGRAMMER'S GUIDES

- **FORTRAN IV**
  PDR3057
  PTU47
- **COBOL**
  PDR3056
  PTU48
- **RPGII**
  IDR3031
  FDR3275*
  PTU49
- **BASIC/VM**
  IDR3085
- **INTERPRETIVE BASIC**
  IDR1813

ASSEMBLY LANGUAGE REFERENCE GUIDES

- **SYSTEM ARCHITECTURE INSTRUCTIONS**
  IDR3060
  MAN1812*
- **PRIME MACRO ASSEMBLER**
  PDR3059
  PTU50

OPERATING SYSTEM REFERENCE GUIDES

- **PRIMOS COMMANDS**
  PDR3108
  FDR3250*
- **SYSTEM ADMINISTRATOR GUIDE**
  IDR3109
- **FILE SYSTEM**
  PDR3110
  PTU51
- **SOFTWARE LIBRARY**
  PDR3106
  PTU52

SOFTWARE SUBSYSTEM REFERENCE GUIDES

- **DATA BASE MANAGEMENT**
  IDR3043
  IDR3044
  IDR3045
  IDR3046
  PTU55
- **EDITOR & RUNOFF**
  FDR3104
- **MIDAS**
  PDR3061
  PTU54
- **SPSS**
  PDR3173
- **FORMS**
  IDR3040
  PTU45
  PTU53

COMMUNICATIONS SUBSYSTEM REFERENCE GUIDES

- **RJE/2780**
  PDR3067
- **HASP**
  IDR3107

PRIME DOCUMENTATION TYPES

IDR **Initial Documentation Release:** provides usable, accurate advanced information.

PDR **Preliminary Documentation Release:** provides more complete and accurate information about the product, but is not in final format.

FDR **Final Documentation Release:** a complete product description; edited, formatted and presented in Prime's highest standards. **The Programmer's Companion*** is another type of FDR; a series of pocket-size, quick reference guides on Prime software products.

PTU **Prime Technical Update:** interim updates to existing documents.

PRIME'S COBOL PROGRAMMER'S GUIDE (Rev. A, September 1978)

This guide documents Prime COBOL and all supporting PRIMOS operating system features as implemented at Master Disk Revision Level 14. Enhancements to COBOL, PRIMOS, and supporting utilities at Revision 15 are given in Appendix K. The index reflects this new organization. Typographic errors, corrections, etc. in the Rev. 14 PDR have been incorporated into the text of this guide.

This guide is organized to make life easier for you, the COBOL application programmer.

We assume you know COBOL, and will easily adapt to Prime's implementation and extensions, which are fully defined in the reference sections of this guide.

PRIMOS, on the other hand, is a large and versatile operating system. It is no small task to sift through all the reference documentation for PRIMOS and its file system, libraries, utilities, and supporting software to find what you need to get a COBOL application running.

To save you trouble, we've done all that for you in the early sections of this guide, by:

- Selecting the PRIMOS capabilities that are of key importance to the COBOL programmer.
- Presenting these capabilities in the usual order of COBOL program development.
- Including all the details on the essential tools.
- Summarizing optional, convenience, and advanced features.
- Leaving out what is irrelevant.

The result is a single document containing everything you need to know to write, modify, compile, load, execute, and debug most COBOL application programs.

In exceptional cases, you may need to refer to supporting reference documents. For example, this guide gives enough information on Prime's DBMS, MIDAS, and FORMS subsystems for you to evaluate whether they are useful to your application. To develop applications using these complex subsystems, however, you need access to the complete details in the reference documents.

We hope you will find this to be a helpful guide to the particulars of COBOL programming within the PRIMOS operating system. We invite comments on the organization and philosophy of this guide, as well as its contents, accuracy, and clarity.

All correspondence on suggested changes to this document should be directed to:

Anthony Lewis, Technical Writer
Technical Publications Department
Prime Computer, Inc.
145 Pennsylvania Avenue
Framingham, Massachusetts 01701

Acknowledgements:

We wish to thank the members of the COBOL PROGRAMMER'S GUIDE team and also the non-team members, both customer and Prime, who contributed to and reviewed this PDR.

First Printing November 1977
Corrected and reprinted September 1978

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGMENT

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

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

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

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

The authors and copyright holders of the copyrighted material used herein

> FLOW-MATIC (Trademark of Sperry Rand Corporation, Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator, Form No. F23-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

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

--from the ANSI COBOL Standard

(X3.23-1974)

PART I

AN OVERVIEW OF PRIME'S COBOL

SECTION 1

INTRODUCTION

THIS DOCUMENT

Purpose and Audience

The purpose of this manual is to provide the experienced COBOL programmer with a guide to efficient COBOL usage in the Prime Environment.

Newcomers to Prime will find in Parts 1 and 2 the introduction and guide they require to apply COBOL in the new environment.

The user familiar with Prime may wish to skim Parts 1 and 2 of this manual.

Advanced concepts and reference are geared to all COBOL users.

Organization and Usage

It is envisioned that this manual will be examined from several different viewpoints. For maximum benefit in any application, the user should be familiar with its organization.

In this connection, the Table of Contents is a guide not only to content, but to order as well; while the index will provide the most direct access to specifics.

The reader should also familiarize himself with the kinds of information available in the Appendices, since they represent a capsule form of repeatedly used data. Various versions of tables are here incorporated into one format, error messages are alphabetically stated, COBOL symbols are presented in capsule form, as are Reserved Words.

This manual is cross-referenced and contains frequent pointers to other documentation for in-depth discussion of system features.

This document is organized into four major parts:

1. An overview of Prime's COBOL (Sections 1 through 3).
2. Using the Prime System (Sections 4 through 9).
3. Advanced Concepts (Sections 10-11)
4. Reference (Sections 12-22 and Appendices A-J).

Part 1, An Overview of Prime's COBOL, discusses Prime's system features, the PRIMOS interactive environment, and Prime's COBOL. As an overview, this part is meant to introduce the uninitiated user to a time-shared, multi-user, interactive system, with its potential for COBOL. The experienced user will here find a summary of Revision 14 enhancements to the COBOL language and Prime system facilities.

Part 2, Using the Prime System, is a tutorial. Its sections will take a new Prime user through those stages required to successfully create and execute COBOL programs on a Prime system.

The first concern is system access. System level commands are listed and summarized in Section 4. The system Editor is then presented as a means for entering and modifying data in general, and source programs in particular.

The remainder of Part 2 is organized to reflect the sequence of steps necessary to compile, load, execute and sort COBOL programs. System utility programs useful in this connection are explained in detail, with in-line examples and complete command summaries. These are organized into self-contained sections on the Compiler, the Linking Loader, SEG, and Sort procedures.

When a quick reference rather than a tutorial is wanted, the user will find capsulized versions of the Compiler and SEG sections repeated in Part 4.

Part 3, Advanced Concepts, addresses system and time efficient procedures. Its audience is both the new and the experienced Prime COBOL user.

Treatement of COBOL program environments is here expanded, with discussion including command files, phantom users, CX mode, and shared procedures.

Management systems are introduced and discussed in relation to COBOL interface. Those aspects of MIDAS (Multiple Keyed Index Data Access System) most commonly utilized in COBOL applications are treated in detail.

Throughout Part 3, the approach remains tutorial, including many examples. To accomodate the large spectrum of user applications, frequent reference is made to additional sources of information.

Part 4, Reference, provides syntactical and general COBOL specifications: it is patterned after the ANSI standards. Its four main subdivisions are:

Fundamental Concepts of COBOL
Nucleus
Functional Processing Modules
Utilities

Fundamental Concepts of COBOL defines and enhances the Nucleus and Functional Progressing Modules. The Nucleus sets forth the structure and governing rules for COBOL's four divisions: Identification Environment, Data, and Procedure. The Functional Processing Modules include Inter-program Communication, Table Handling, Indexed I-O, and Relative I-O. Utilities is a reference presentation of the Prime COBOL Compiler and the SEG utility program.

Effective usage of the Reference requires considerable knowledge of its organization:

In Fundamental Concepts, the user will find a generalized COBOL program summary, together with a skeletal component structure for such a program. These are expanded in the example program, SAMPLE which follows follows with its Listing File. This summary is a thumb-nail presentation of required and optional program structure, which is expanded throughout the Nucleus.

The Nucleus presents information related to the Identification, Environment, Data and Procedure Divisions (Section 13 through 16 respectively).

Each sections begins with a thumb-nail, skeletal component format for the program division it discusses. This is expanded throughout the section in the sequence in which it must appear.

COBOL verbs are presented alphabetically in Section 16, the Procedure Division. A quick verb index precedes this data and appears also in Appendix D.

At the close of each division section, the user will find an example of source coding for that given division. These examples form a functional program, REF2, which illustrates the interrelationship of component parts. The COMPILE, LOAD, CREATK, and EXECUTE sequences for REF2 are presented immediately following the program example at the close of Section 16. These, and the compiled Listing File which accompanies them, form an integrated COBOL picutre. They relate both to program division discussion in Part 4, and to their corresponding tutorial sections in Part 2.

A VMODE Expanded Listing for REF2 appears in Appendix J.

Functional Processing Modules are self-contained, often restating concepts, data descriptions, and COBOL statement formats elsewhere described. The reader will here find 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.

This Version

This is a Preliminary Documentation Release (PDR). It represents a SECOND draft, providing more complete and accurate information about the product than previously available, but not in itself complete.

Thus, those sections still incomplete are listed in the Table of Contents and outlined in place. Such sections will be finalized and incorporated into a Final Documentation Release (FDR). The FDR will represent the complete product description up to the stated software revision number and be produced in typeset format.

PRIME CONVENTIONS

Symbols, abbreviations, special characters and conventions frequently used in Parts 1, 2, and 3 of this document are defined below.

Terminal Functions:

Character	Function
(CR)	Carriage return.
\	Backslash. Used as Editor's tab character.
"	Delete or erase symbol to delete one preceding character.
?	Kill character to delete all characters in current command line.

Prime Filename Conventions:

Filename	Function
B←filename	Binary (object) file.
L←filename	Listing file.
C←filename	Command file.
filename	Source file.
*filename	SAVED (Executable) file.
M←filename	Map file.
#filename	SEG runfile.

NOTES:

1. New file partitions accept a maximum of 32 characters for filename.

2. The two character sequence ← in a filename represents only a single character. This character is the back-arrow on the terminal keyboard which prints as an underscore on output devices. Since this is inconvenient in text examples which use underlining, the "←" convention has been adopted.

Text Conventions:

ALL CAPS	An item which must be included verbatim. Underlines indicate acceptable abbreviations.
Underlining	Indicates user input in examples.

RELATED DOCUMENTS

The following documents include information on the PRIMOS system and Prime Utility programs. They will be important adjuncts to this release:

Title	Manual No.
Reference Guide, PRIMOS Commands	PDR 3108
The System Administrator's Guide	IDR 3109
FORMS Management System (FORMS)	IDR 3040
User Guide for the Data base Administrator	IDR 3043
Reference Guide for DBMS Schema DDL	IDR 3044
COBOL Reference Guide for DBMS	IDR 3046
The PMA Programmer's Guide	PDR 3059
Reference Guide, Multiple Index Data Access System (MIDAS)	PDR 3061
The New User's Guide to Editor and Run off	FDR 3104
Reference Guide, Software Library	PDR 3106
Reference Guide, File Management System (FMS)	PDR 3110

SECTION 2

PRIME COBOL

FEATURES

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

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

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

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

Module	Features Available in Prime COBOL
Nucleus	All of level 1, plus these features of level 2: Levels 77, 01-30, 88; Value series or range, level 88 conditions; AND OR = < > in conditions; Procedure-names consisting of digits only; COMPUTE with multiple receiving fields; PERFORM VARYING one index; Mnemonic-names for ACCEPT or DISPLAY devices; Qualification of Names (Procedure Division); Sign test; STRING; (supported in V-Mode only) UNSTRING; (supported in V-Mode only) ACCEPT { DAY / TIME / DATE } .
Sequential I-O	All of level 1 plus these features of level 2: RESERVE clause and variable form of BLOCK; Multiple operands in OPEN & CLOSE, with individual option per file.
Relative I-O	All of level 1 plus: RESERVE clause; DYNAMIC access mode (with READ next); START (with key relations EQUAL, GREATER, or NOT LESS).

Module	Features Available in Prime COBOL
Indexed I-O	All of level 1 plus: RESERVE clause; DYNAMIC access (with READ next); RANDOM access mode with READ by KEY; START (with key relations EQUAL, GREATER, NOT LESS).
Library	Level 1
Table Handling	All of level 1 plus: SEARCH
Inter-program Communication	Level 1

SYSTEM FILES

To utilize COBOL, the following files must be available on the system in the UFD's specified:

UFD	FILE-NAME
CMDNCO	COBOL
SYSOVL	C$$DAT C$$DAR C$$GEN C$$FIN C$$END C$$64V
LIBRARY	COBLIB COBKID VCOBLB FTNLIB (RMODE) PFTNLB, IFTNLB } (VMODE)

VCOBLB

The new 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 = Hundreth of seconds

C$INSP = INSPECT statement

C$UNS1/C$UNS2 = UNSTRING statement

C$STR1/C$STR2 = STRING statement

C$IN = File assignment initialization

C$OS = Open sequential file

C$CS = Close sequential file

C$RS = Read sequential file

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

SECTION 3

COBOL AND PRIMOS

OPERATING SYSTEM MODES

64R Mode, Prime 300, 400, 500

64V Mode, Prime 400, 500

FILE SYSTEM SUMMARY

PROGRAM ENVIRONMENTS

Interactive

Queued Jobs Using Command Files

Phantom Users

CX, Sequential Job Processor

Shared Procedures

SYSTEM RESOURCES SUPPORTING COBOL

The portions of SECTION 3 outlined above were incomplete at this printing.

PART II

USING THE PRIME SYSTEM

SECTION 4

SYSTEM ACCESS

This section treats the following topics:

- Accessing the system
- PRIMOS command summary
- Program and data entry from other media
- Entering and modifying data: the Editor

SYSTEM CONCEPTS

Certain Prime system conventions and concepts are employed throughout this document. They are basic to effective usage of COBOL and PRIMOS.

Basic Concepts

file — An organized collection of information stored on a disk (or a peripheral storage medium such as tape). Each file on a disk has an identifying label called a filename.

UFD — A User File Directory. A special type of file containing a list of filenames and the location of the corresponding files. A file whose name is on this list is said to be "in this directory".

MFD — The Master File Directory. A special UFD which contains the names of the UFDs on a particular disk. There is one MFD for each logical disk.

sub-UFD — A User File Directory in a UFD or other sub-UFD.

Note

File directories with names in the MFD are UFDs; all other file directories are sub-UFDs.

logical disk — A division of the computer's disk memory. It may be all or part of a physical disk. The logical disk is labelled by an octal number called the logical disk number (obtainable from STATUS DISKS command).

volume name — A literal name corresponding to a logical disk, e.g., logical disk 4 may have volume name DOCUMN. (Also obtainable from STATUS DISKS command).

treename — An extended form of the filename which completely describes the location of a file in the directory structure. Treenames may be one of the following:

- filename
- ufd-name [password]>...>sub-ufd-name [password]>filename
- <volume-name>ufd-name [password]>...sub-ufd-name [password]>filename
- <logical-disk>ufd-name [password]>...sub-ufd-name [password]>filename

where:

filename — is the name of the file.

ufd-name
sub-ufd-name — is the name of the UFD or sub-UFD in which the file (or sub-UFD) to right of it on the line is located.

password — is the password of the UFD or sub-UFD, if it has been protected with a password.

volume-name — is the literal name of the disk on which the file is located; if volume-name is specified as <*>, this is the same as using the name of the disk the user is currently attached to.

logical-disk — is the octal number of the logical disk.

source file — The program file created by the user consisting of text, program statements, comments, etc.

binary file
object file — A translation of the source file generated by the COBOL compiler. Such files are in the format required as input to the linking loader or segmented loader.

runfile — The loaded, executable version of a program consisting of the binary file, subroutines and library entries used by the program, COMMON areas, initial settings, etc. This file is created using LOAD and SEG.

mode — An addressing scheme. The mode used determines the construction of the computer instructions by the compiler. Modes available to the COBOL programmer are relative-addressed (RMODE or 64R) and segmented-addressed (VMODE or 64V). (The number is the user memory size in K's of 16-bit words.)

identity — The addressing mode plus its associated repertoire of computer instructions. Programs compiled in 64R mode execute in the R-identity; programs compiled in 64V mode execute in the V-identity.

byte 8 bits; 1 ASCII character.

word 16 bits; 2 bytes; 2 ASCII characters.

ACCESSING THE SYSTEM

The most basic commands for an interactive session with PRIMOS are outlined below; a PRIMOS command summary follows.

This is not intended as an exhaustive presentation. The reader should consult REFERENCE GUIDE, PRIMOS COMMANDS (PDR31Ø8) for a complete discussion.

Logging In

'Logging In' identifies the user to the system and establishes initial contact; it provides access to a work area, files, and all the general resources of the computer.

The format of the LOGIN command is:

```
LOGIN ufd-name [password]
```

where ufd-name is the name of a User File Directory in the system. PRIMOS will respond with the message:

```
ufd-name (job or term#) LOGGED IN AT time date
```

Example:

```
LOGIN PENNY PROTECT

PENNY (3Ø) LOGGED IN AT 1Ø'33 Ø12478
```

The number in parenthesis is the PRIMOS-assigned job number. The LOGIN time (here 1Ø'33) is expressed in the 24-hour system. The date (here Ø12478) is expressed as Month Day Year.

Accessing Files

Having logged in, the user now has a work space and direct access to a User File Directory.

The LISTF Command: The UFD contains the files to 'work on'. To list these files, the LISTF command should be used. Its format is:

```
LISTF
```

For example, when logged into a UFD named STAFF which contains the files FRANK, MARTHA, ALBERT, giving the LISTF command would have this effect:

```
LISTF

UFD=STAFF 3 Ø

FRANK MARTHA ALBERT

OK,
```

The number after the ufd-name identifies the logical device; the letter following the number will be either O (for owner) or N (for non-owner). See the PASSWD and PROTEC commands for details.

If there were no files in the STAFF UFD, a LISTF would have this result:

```
OK, LISTF

UFD=STAFF 3 Ø

.NULL.

OK,
```

Logging into the system has the effect of 'attaching' the user to a UFD. At any time, the UFD to which you are currently attached is known as the current UFD.

The ATTACH Command: When access is required to files in other UFDs, the ATTACH command should be used:

```
ATTACH ufd-name [password]
```

At this point, the ufd-name specified in the ATTACH command becomes the current UFD.

Spelling errors or other improper data cause the system to respond with one of the following messages:

```
ufdname NOT FOUND

     or

BAD PASSWORD

     or

NO UFD ATTACHED
```

In such instances, correct the ufd-name or password and reissue the ATTACH command.

Sub-UFDs

A sub-UFD is a selection of files in a UFD which have been grouped together. This grouping has a name called sub-UFD-name. Once logged into a UFD, the ATTACH command can be used to access any sub-UFD within it. This sub-UFD would then be the current directory. At this juncture, giving a LISTF command provides a list of only those files within the current directory (sub-UFD).

The ATTACH Command: When applied to a sub-UFD, the ATTACH command requires at least one additional parameter:

ATTACH sub-UFD-name [password] [ldisk] [key]

where ldisk is logical disk number and is usually omitted for sub-UFDs. When ldisk is omitted, key is specified with a numeric positional character, i.e., 1/n. 1/n means, "skip over one numerical parameter and use what follows as the second parameter"; n is the value of key.

For most applications, the COBOL programmer will set the value of key at 2. A complete list of key values and their meanings is provided in REFERENCE GUIDE, PRIMOS COMMANDS (PDR31Ø8).

Example:

```
OK, LISTF

UFD=STAFF 3 Ø

JAMES JOHNSTON JONES

OK, ATTACH JOHNSTON 1/2
OK,
```

The CREATE Command: Sub-UFDs can be created within UFDs or sub-UFDs. A sub-UFD can contain files and sub-UFDs.

The command to define and name a sub-UFD is the CREATE command. This command creates a sub-UFD in the current directory.

The format of the CREATE command is:

CREATE sub-ufd-name

Two files or sub-UFDs with the same name are not permitted in the current UFD. Should this be attempted inadvertently, PRIMOS will respond with the message: ALREADY EXISTS.

Example:

```
OK, ATTACH SALES
OK, LISTF

UFD=SALES 3 Ø

.NULL.

OK, CREATE EAST.COAST
OK, CREATE WEST.COAST
OK, CREATE EAST.COAST
ALREADY EXISTS
ER!  LISTF

UFD=SALES 3 Ø

EAST.COAST WEST.COAST

OK,
```

Examining Files

Contents of a file can be examined with the SLIST and SPOOL commands. The SLIST command displays a file on the terminal; the SPOOL command has a file printed out on the line printer.

The SLIST Command: The format of the SLIST command is:

```
SLIST treename
```

This command causes the treename specified (in the current UFD or sub-UFD) to be displayed.

The format of the SPOOL command is:

```
SPOOL filename
```

PRIMOS makes a copy of filename in the Spool Queue List for the line printer, and displays the message:

```
YOUR SPOOL FILE IS PRTxxx (length)
```

where xxx is a 3-digit number wich identifies the file in the Spool Queue List. The reason there is a list, rather than just having each file spooled out as the request comes, is that some requests are very long - hundreds of pages. PRIMOS spools out the shorter files as soon as possible, rather than make the user wait while the long files are printed. The word (SHORT or LONG) which follows the SPOOL message indicates the category to which the file has been assigned.

It is possible to check the status of a SPOOL request by giving the command:

SPOOL -LIST

Example:

```
OK, SPOOL POEM
GO
YOUR SPOOL FILE IS PRT013 (LONG) REV 14.00

OK, SPOOL -LIST
GO
```

USER	FILE	DATE/TIME	OPTS	SIZE	NAME	FORM
COSMO	PRT005	11/09 10:03	S	5	RTN168	WHITE
MARTHA	PRT007	11/09 15:34	S	3	RTN172	WHITE
HAMPSO	PRT009	11/11 10:25	S	6	PAGTUR	
LAWLER	PRT010	11/11 10:26	S	9	L_AM9600	
RANDI	PRT011	11/11 10:26	S	1	WKINFO	
MORRIS	PRT012	11/11 10:27	S	3	L_CMP$SR	
ALICE	PRT013	11/11 10:28	L	17	POEM	

To cancel a spool request, the command format is:

SPOOL -CANCEL PRTxxx

where xxx is the number of your Spool File.

For example:

```
OK, SPOOL -CANCEL PRT013
GO
PRT013 CANCELLED.

OK,
```

Notes

1. If PRIMOS has already begun SPOOLing a file, it is not possible to cancel the SPOOL request. Instead, PRIMOS will display the message:

   CAN'T CANCEL REQUEST - FILE IS OPEN OR CURRENTLY PRINTING

2. If the file has already been spooled, or the PRT number given is incorrect, the message will read:

   PRTxxx NOT IN QUEUE

3. If the name of the file is specified instead of the PRT number, the message will read:

   BAD PRINT FILE NAME

Renaming and Deleting Files and Empty Sub-UFDs

The CNAME Command: Files and sub-UFDs can be renamed with the CNAME command. The format is:

CNAME old-name new-name

If new-name already exists, PRIMOS will display the message:

ALREADY EXISTS

An incorrect old-name prompts the message:

NOT FOUND
ER!

The DELETE Command: The DELETE command enables the deletion of files and empty sub-UFDs. Its format is:

DELETE { filename / sub-UFd-name }

Sub-UFDs containing files cannot be deleted with this command. Instead, all files contained within the sub-UFD must first be DELETEd.

PRIMOS warns the user if an attempt is made to DELETE sub-UFDs containing files; it displays the message:

DIRECTORY NOT EMPTY

Closing Files and Logging Out

The CLOSE Command: Files are automatically opened by PRIMOS when accessed by the user. Certain conditions, such as quitting out of EDITOR via the BREAK key or CONTROL-P, will leave these files open. If an attempt is made to re-access them, PRIMOS will respond with the message:

```
FILE IN USE
```

The CLOSE command must be given in such instances. Its format is:

```
CLOSE { filename        }
      { ALL             }
      { file-unit-number }
```

Once closed, the file is ready to be reacessed by the user.

The CLOSE command is discussed again in this section under Entry From Other Media.

The LOGOUT Command: When finished with a session at the terminal, give the LOGOUT command. Its format is:

```
LOGOUT
```

PRIMOS acknowledges the command with the following message:

```
UFD-name (user-#) LOGGED OUT AT (time) (date)
TIME USED = terminal-time CPU-time I/O-time
```

user-# is the same as the one assigned at LOGIN

terminal-time is the amount of elasped clock time between logging in and l gging out (hours and minutes).

CPU-time refers to Central Processing Unit time consumed (minutes and seconds).

I/O-time is the amount of input/output processing time used (minutes and seconds).

Sample PRIMOS Session

```
LOGIN STAFF

STAFF (15)  LOGGED IN AT 10'35 110579
LISTF

UFD=STAFF 3 0

FRANK  MARTHA  ROSEMARY

OK, ATTACH SALES
OK, LISTF
UFD=SALES 3 0
.NULL.

OK, CREATE EAST.COAST
OK, CREATE WEST.COAST
OK, CREATE EAST.COAST
ALREADY EXISTS.
ER! LISTF

UFD=SALES 3 0

EAST.COAST  WEST.COAST

OK, ATTACH EAST.COAST 1/2
OK, CREATE NORTH
OK, CREATE SOUTH
OK, CREATE CENTRAL
OK, LISTF

UFD=EAST.COAST 3 0

NORTH  SOUTH  CENTRAL

OK, ATTACH NORTH 1/2
OK, CREATE MAIN
OK, CREATE BRANCH1
OK, CREATE BRANCH3
OK, DELETE BRANCH3
OK, CREATE BRUNCH2
OK, CNAME BRUNCH2 BRANCH2
OK, ATTACH SALES
OK, ATTACH EAST.COAST 1/2
OK, ATTACH NORTH 1/2
OK, LISTF

UFD=NORTH 3 0

MAIN  BRANCH1  BRANCH2

OK, LOGOUT
```

```
STAFF (15) LOGGED OUT AT 11'28 110579
TIME USED=0'12 0'06 0'032
```

PRIMOS COMMAND SUMMARY

An alphabetic list of PRIMOS commands of special interest to the COBOL programmer follows (acceptable abbreviations are underlined):

/*	Command line comment designator
ASSIGN	Obtains exclusive control of a peripheral device
ATTACH	Attaches to UFD or sub-UFD
AVAIL	Gives records available on specified disk
BINARY	Opens a file for writing on PRIMOS unit 3 (Obs.)
CLOSE	Closes files
CMPF	Compares ASCII files
CMPRES	Compresses ASCII file
CNAME	Changes a filename
COBOL	Invokes COBOL compiler
COMINPUT	Switches command stream from terminal file and vice-versa
COMOUTPUT	Switches terminal output to file and vice-versa
CPMPC	Punch cards on parallel interface card punch
CREATE	Creates a sub-UFD in the current UFD
CREATK	Defines MIDAS template file.
CRMPC	Reads cards from the parallel interface card reader
CRSER	Reads cards from the serial interface card reader
CSUBS	Interfaces COBOL with DBMS (data base management system)
CX	Invokes the sequential phantom job execution utility
DATE	Prints system time and date at terminal
DELETE	Deletes a filename from the UFD
DELSEG	Deassigns segments assigned by SEG.
ED	Invokes Prime's text editor
EDB	Invokes the binary editor (for library building)
EXPAND	Expands a file previously compressed with CMPRES
FAP	Updates and maintains FORMS directory.
FDL	Converts data descriptors for FORMS into format usable by routine program.
FILMEM	Fills the user memory space with zeros
FILVER	Compares two binary files for equivalence and prints differences
FUTIL	Invokes Prime's file manipulation utility
INPUT	Opens file for reading on PRIMOS unit 1
KBUILD	Builds MIDAS data file.
KIDDEL	Deletes MIDAS file.
LABEL	Creates an ANSI COBOL level-1 volume label on a magnetic tape.
LISTF	Prints list of entries in current UFD
LISTING	Open a file for writing on PRIMOS unit 2
LOAD	Invokes the Linking Load (R-identity)
LOGIN	Logs the user into the system
LOGOUT	Logs the user off the system
MAGNET	Invokes the magtape/disk transfer/translation utility
MAGRST	Transfers files from 9-track tape to disk
MAGSAV	Transfer files from disk to 9-track tape

MDL	Punches paper tape of specified locations of memory in self-loading format
MESSAGE	Transmits message from user terminal to system console
MRGF	Merges ASCII files.
OPEN	Opens a file by name on a specified PRIMOS unit for specified operations
PASSWD	Sets passwords for current UFD
PHANTOM	Spawns a user to execute the specified command file
PM	Prints program start and end addresses, register contents
PRERR	Prints error message in ERRVEC
PRMPC	Prints on parallel interface driven line printer
PROTEC	Sets owner/non-owner rights for files and sub-UFDs
PRSER	Print on serial interface driven line printer
PSD	Invokes the Prime Symbolic Debug utility
PTCPY	Duplicates and verifies paper tapes
REMAKE	Maintains MIDAS file.
REPAIR	Rebuilds damaged MIDAS file.
RESTORE	Restores a file from disk to user's memory space
RESUME	Restores a file to user's memory and begins execution
RUNOFF	Invokes Prime's text output formatter
SAVE	Writes memory into a disk runfile with the address values and register settings
SEG	Invokes the segmented-address (V-identity) utility
SIZE	Gives size of file
SLIST	Prints contents of file to user's teminal
SORT	Sorts an ASCII file
SPOOL	Spools output files to line printer
START	Sets registers and keys and begins program execution
STATUS	Prints status of specified system parameters
TA	Attaches to UFD with treename specified as in FUTIL
TAP	Invokes octal mode debugging routine
TERM	Sets/Displays terminal kill and erase characters, set duplex
TIME	Prints connect time, compute time, and disk I/O time at terminal
UNASSIGN	Relinquishes control of a peripheral device
UPCASE	Reformats files by changing lower-case letters to upper-case
USERS	Prints number of users currently logged in
VPSD	Invokes Debugging utility for V-identity
VPSD16	Used when the program is so large that it overlays VPSD.

For a complete treatment of all commands, see the REFERENCE GUIDE, PRIMOS COMMANDS, PDR31Ø8.

CREATING AND ENTERING SOURCE PROGRAMS

Entry From Other Media

Source programs existing on punched cards, magnetic tape, or punched paper tape can easily be read onto disk files using PRIMOS-level utilities. In addition, the punched card and magnetic tape transfer utilities will translate from BCD or EBCDIC representation into ASCII representation, saving considerable time and effort.

Subroutines and other installation-dependent operations may be altered to conform to PRIMOS using the Editor (described later in this section).

The general order of operations for input from a peripheral device is:

1. Obtain exclusive use of the device (Assigning at PRIMOS level).

2. Transfer programs with appropriate utility.

3. Release device to other users (Unassigning at PRIMOS level).

Assigning A Device: Assigning a device gives the user exclusive control over that peripheral device. The PRIMOS-level ASSIGN command is given at the terminal:

```
ASSIGN device [-WAIT]
```

where device is a mnemonic for the appropriate peripheral:

CARDR	Serial Card Reader
CRn	Parallel Card Reader n (Ø-3)
MTn	Magnetic Tape Unit n (Ø-7)
PTR	Paper Tape Reader

and -WAIT is an optional parameter. If included, it queues the ASSIGN command if the device is already in use. The assignment request remains in the queue until the device becomes available or the user types the CONTROL-P or BREAK key at the terminal; both occurrences return the user to PRIMOS. If the requested device is not available and the -WAIT parameter has not been included, the error message:

```
DEVICE IN USE
```

will be printed at the terminal.

After all I/O operations are completed, exclusive use if relinquished by the command:

```
UNASSIGN device
```

where device is the same mnemonic used in the ASSIGN command.

Reading Punched Cards: Assign use of the parallel interface card reader with the ASSIGN command:

```
AS CRØ -WAIT
```

To read cards from the card reader, load the card deck into the device and enter the command:

```
CRMPC treename
```

where treename is the name of the file into which the card images are to be loaded.

Source deck header control cards are set up as follows:

Source deck representation	Columns 1 and 2 of deck header card
BCD	$6
EBCDIC	$9
ASCII	no header card

Reading continues until a card with $E in columns 1 and 2 is encountered (end of deck). Control returns to PRIMOS and the file is closed. If the cards are exhausted (or the reader is halted by the user), control returns to PRIMOS but the file is not closed.

If more cards are to be read into the file at this point, the reader should be reloaded. Reading is resumed by the START command given at the terminal: START.

The format of the command to close the file is:

```
CLOSE { filename }
      { ALL      }
```

To close all files and units, the CLOSE command should be given in the form:

```
CLOSE ALL
```

Example of a card reading session:

```
OK, AS CRØ -WAIT
OK, CRSER old-program-1
OK, UN CRØ
OK,
```

If a serial interface card reader is used, the process is similar with slightly different reader commands. CARDR may be abbreviated to CAR.

```
OK, AS CARDR -WAIT
OK, CRSER old-program-2
OK, UN CAR
OK,
```

Reading Magnetic Tape/The MAGNET Utility: Assign use of the magnetic tape drive by:

```
AS MTx -WAIT
```

where x is the tape drive unit number: Ø,1...7.

Mount the tape on the selected drive unit and read the tape with PRIMOS' MAGNET utility:

```
OK, MAGNET
GO

MAGNET 14.Ø 19-MAY-77

OPTION: READ

MTU# = unit number [/tracks]
```

where:

unit-number	is the number of the magnetic tape drive unit which was previously assigned and
tracks	is either 7 or 9; if this parameter is omitted, 9-track tape is assumed.

MAGNET than asks a series of questions about the tape format (user responses are underlined):

Prompt	Response	Remarks
MTFILE# =	tape-file-number	This is the number on the tape. A positive integer causes the tape to be rewound and then positioned to the file number; a Ø causes no repositioning of the tape.
LOGICAL RECORD SIZE =	number	This is the number of bytes/line image; normally this is 8Ø for COBOL source program

BLOCKING FACTOR =	blocking factor	Blocking-factor is the number of logical records per record.
ASCII, BCD, BINARY, OR EBCDIC?	ASCII	indicates that no translation is to occur between tape and disk. The data is written to the disk file in ASCII format.
	EBCDIC	indicates that the data on the tape is to be translated from EBCDIC to ASCII before written to the disk file.
	BCD	specifies that the data is to be translated from BCD (6-bit) to ASCII before being written to the disk file. This option is only meaningful when used with a 7-track tape. Note that no 6-6-5 unpacking is done by Magnet when this option is specified
	BINARY	Indicates that the data is to be written verbatim to a binary disk file. The record size is the specified logical record size. No translation occurs.
FULL OR PARTIAL RECORD TRANSLATION?	FULL PARTIAL	The question is asked only for BCD or EBCDIC representations. PARTIAL allows specified bytes in the record to be transferred to disk without translation to ASCII. This is useful when transferring data files. Most source programs will be transferred with the FULL option.
OUTPUT FILENAME:	filename	This is the name of the file in the UFD into which the magnetic tape is read.

OK TO DELETE OLD filename?	{ YES / NO }	This question will be asked only if the filename specified already exists in a UFD. A YES will cause the transfer to being

Upon completion of the dialogue, the following message will be printed:

```
DONE, tape-records RECORDS READ, disk-records DISK RECORDS OUTPUT
OK,
```

Use of the tape drive unit should then be relinquished by the command: UN MTx.

Reading Punched Paper Tape: Source programs punched on paper tape in ASCII representation can be read onto a disk file with the Editor utility.

```
OK, AS PTR -WAIT          assign tape reder
OK, ED                    invoke Editor
GO
INPUT
(CR)                      switch to EDIT mode
EDIT
INPUT (PTR)               input from tape reader
 tape is being read
EDIT
FILE filename             file input under filename
OK, UN PTR
```

The EDITOR/Entering and Modifying Programs

Programs are normally entered into the computer using Prime's Text Editor (ED). This editor is a line-oriented text processor whose line pointer is always located at the last line processed (whether the processing is printing, locating, moving pointer, etc). The Editor operates in two modes, INPUT and EDIT.

Using the Editor: When creating a new file, the Editor is invoked by the command:

```
ED
```

which places the Editor in the INPUT mode. To modify an existing file, use the expanded command format:

```
ED filename
```

This places the Editor in the EDIT mode.

A CARRIAGE RETURN with no preceding characters on that line switches the Editor from one mode to another.

Input Mode: The INPUT mode is used when entering text information into a file (e.g., creating a program). The word INPUT is displayed at the user's terminal to indicate the Editor has entered that mode. The RETURN key terminates the current line and prepares the Editor to receive a new line. Tabulation is performed with the backslash (\) character. Each backslash represents the first, second, etc. tab setting; the default tabs are at positions 6, 15, and 3Ø. These setting may be overridden, and up to 8 tab settings may be specified by the user with the TABSET command (described later). A RETURN with no text preceding it puts the Editor into EDIT mode.

Edit Mode: The EDIT mode is used when the contents of the file are to be modified. More than fifty commands are available, however, a small subset of these will suffice for most purposes. Commands are listed and described later in this section.

In EDIT mode, the Editor maintains an internal line pointer at the current line (the last line processed). Commands such as TOP, BOTTOM, FIND, and LOCATE, move this pointer. WHERE prints out the current line number; POINT moves the pointer to a specified line number. The MODE NUMBER command causes the line number to be printed out whenever a line of text is printed. All commands for location and modification begin processing with the current line.

A CARRIAGE RETURN without any preceding characters on that line puts the Editor into INPUT mode.

Special Characters: Unless modified at the user's installation, the Editor's erase and kill symbols are those of PRIMOS. The Editor's default erase character is the double-quote ("), and the default kill character is the question-mark (?). For each " typed, a character is erased (from right to left). The entire current line may be deleted by typing the kill character. A line followed by a ? is null, and a RETURN at that point will switch the EDITOR into the other mode.

The semicolon character (;) is interpreted as a carriage return by the Editor in INPUT mode. While this places restrictions on entering semicolons as part of a file, it does provide a 'brief' format for inputting multiple short entries or blank lines.

Example:

```
INPUT
*;* TEST-FILE;*
```

will become:

```
*
* TEST-FILE
*
```

A more detailed discussion of special characters and how to manipulate them is provided in the New User's Guide to EDITOR and RUNOFF, FDR31Ø4.

Saving Files: Orderly termination of an Editor session is done from EDIT mode. The command:

FILE filename

writes the current version of the edited file to the disk under the name filename. The specified file will be created if it did not previously exist, or overwritten if did exist. If an existing file is being modified, the command should be given as:

FILE

This writes the new version to the disk under the old filename. After execution of the filing command, control is returned to PRIMOS.

Useful Techniques: The following are highlights of some Prime Editor techniques which will be of particular interest to the COBOL user:

- Tab Settings: When entering source code, much time can be saved using the TABSET command. In INPUT mode, each backslash character (\) is interpreted as one tab setting; the default values are positions 6, 15, and 30. Tabs may be set to whatever values each programmer finds useful.

- Column Display: Entering source code and other data is also facilitated by the Editor's column display feature. A banner of column numbers can be displayed across the top of the terminal screen providing alignment guides. The command MODE COLUMN, given in Edit mode, causes the column header display to be printed each time Input mode is entered during an Editor session.

- Moving Lines of Code: Several Editor commands enable the transfer of coded lines to and from Editor work files.

  The LOAD command inserts (load) a copy of filename into the Editor's work file below the current line, repositioning the pointer just below the end of the LOADed text.

  The UNLOAD command copies (unloads) the specified number of lines in the Editor work file into filename

  The DUNLOAD command copies (unloads) the specified lines the work file into filename, and then deletes those lines from the work file.

- Finding A Line By Statement Label: The FIND command may be used to locate a statement label in a COBOL program.

- Modifying A Line Without Changing Character Positions: The MODIFY command is used when a line must be modified but the absolute column alignment must remain the same.

Sample Editing Session 1

See the list following these examples for an explanation of the commands.

```
OK, ED
GO
INPUT
RETURN
EDIT
C"MODE COLUMN
RETURN
INPUT
         1         2         3         4         5         6         7
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST2.                  Source coding is keyed in,
       INSTALLATION. PRIME.                aligned by column.
\ *
           .                               The first tab default is
           .                               position 6. A space after the
           .                               backslash character positions
\ *                                        the asterisk in the continua-
                                           tion column 7.
       PROCEDURE DIVISION.
           .
           .
           .
       OPEN-FILES.
           OPEN INPUT  INPUT-FILE.
           .
           .
           .
RETURN

EDIT                                       The PRINT command in EDIT
                                           mode displays entered source
P2Ø                                        statements.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TEST2.
       INSTALLATION. PRIME.
      *
           .
           .
           .
      *
       PROCEDURE DIVISION.
           .
           .
           .
       OPEN-FILES.
           OPEN INPUT  INPUT-FILE.
           .
           .
```

```
.
```

RETURN

```
INPUT
        1         2         3         4         5         6         7
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
       DONE.
           CLOSE INPUT-FILE.
           STOP RUN.
```

Each time INPUT is invoked, there is an automatic column number display. This convention may be used at terminal scrolling (when the column display scrolls or roll off the screen).

RETURN

```
EDIT
```

FILE TEST2

The FILE command writes the contents of the edited file to the filename specified (in this case, TEST2). The FILE command also causes a return from the Editor to PRIMOS.

```
OK,
```

Sample Editing Session 2:

```
OK, ED
GO
INPUT
RETURN
EDIT
TABSET 8 17                          Set tabs
RETURN
INPUT
*                                    Begin entering data
* THIS IS A RECORD STREAM FOR
*\A COBOL PROGRAM                    Use backslash character for
*\\USING FORMS                       tabulation
*
RECORD STREAM
*\MACRO DEFF"INITIONS                Erase errors
FI\DEF\FIELD
V\DEF\VALLIDATE
            .
            .
            .
\END\STREAM
*;*;*;*                              Semicolons enable multiple
                                     entries with a single line
RETURN
EDIT
T                                    Position pointer at beginning
                                     of file.

FIND FI                              Locate statement to be modified.
FI      DEF     FIELD
MODIFY/FI/F
F       DEF     FIELD                Modification complete;
                                     Absolute alignment remains
                                     intact

FIND V
V       DEF     VALLIDATE
C/LL/L
V       DEF     VALIDATE             Change is complete; relative
                                     alignment remains intact.
T
P30                                  Print corrected file (30
                                      lines).
.NULL.
*
* THIS IS A RECORD STREAM FOR
*       A COBOL PROGRAM
*               USING FORMS
*
*
RECORD STREAM
        MACRO DEFINITIONS
F       DEF     FIELD
```

```
V       DEF       VALIDATE
           .
           .
           .
        END       STREAM
*
*
*
*
FILE FORMSTREAM
OK,
```

Editor Command Summary

The following is an alphabetic list of each Editor command and its function. Acceptable command abbreviations are underlined. Especially useful commands are indicated with a bullet (•). For a detailed description of all commands, see the Editor Reference Section of THE NEW USER'S GUIDE TO EDITOR AND RUNOFF, FDR31Ø4.

Note

The string parameter in a command is any series of ASCII characters including leading, trailing, or embedded blanks.

Command	Function
• APPEND string	Appends string to the end the current line.
• BOTTOM	Moves the pointer beyond the last line of the file.
BRIEF	Speeds editing by suppressing the (default) verification responses to certain Editor commands.
• CHANGE/string-1/string-2/[G] [n]	Replaces string-1 with string-2 for n lines. If G is omitted, only the first occurrence of string-1 on each line is changed; if G is present, all occurrences on n lines are changed.
• DELETE [n]	Delete n lines, including the current line (default n=1).
DELETE TO string	Deletes all lines up to but not including line containing string.
• DUNLOAD filename [n]	Deletes n lines and writes them into filename. (default n=1.)
DUNLOAD filename TO string	Same as DELETE...TO, but writes deleted lines into filename.
ERASE	Resets current erase character to character.

• FILE filename	Writes the contents of the current file into filename and exits to PRIMOS.
• FIND string	Moves the pointer down to next line beginning with string.
FIND(n) string	Moves the pointer down to next line with string beginning in column n.
GMODIFY	Allows the user to enter a string of subcommands which modify characters within a line.
INPUT {(ASR) (PTR) (TTY)}	Reads text from the specified input device: ASR (Teletype paper tape reader), PTR high-speed paper tape reader) or TTY (terminal). Default is TTY.
• INSERT string	Inserts string after current line.
KILL character	Sets kill character to character.
LINESZ n	Changes maximum line length.
• LOAD filename	Loads filename into text following the current line.
• LOCATE string	Moves pointer forward to the next line containing string, which may contain leading and trailing blanks.
• MODE COLUMN	Displays column numbers whenever INPUT mode is entered. (The command is given in EDIT mode.)
MODE COUNT start increment width {PRINT BLANK SUPPRESS}	Turns on the automatic incremented counter.
MODE NCOLUMN	Turns off the column display (default). (The command is given in EDIT mode.)

MODE NCOUNT	Disables the MODE COUNT function.
MODE NUMBER	Displays line numbers in front of printed line.
MODE NNUMBER	Turns off the line number display (default).
(Case Modes)	Case-flagging is done by preceding each new case with either ^U (for Upper-Case) or ^L (for Lower-Case). PRUPPER and PRLOWER are intended for use on Upper-Case-only Terminals.
MODE PRALL	Prints letters without case flagging (default).
MODE { PRUPPER }	Accepts/prints case-flagged Upper-Case letters. Each line of input/output begins implicitly flagged as Upper-Case.
MODE { PRLOWER }	Accepts/prints case-flagged Upper-Case Letters. Each line of input/output begins implicitly flagged as Lower-Case.
MODE PROMPT	Prints prompt characters for INPUT and EDIT modes.
MODE NPROMPT	Inhibits printing of INPUT and EDIT prompt characters (default).
• MODIFY/string-1/string-2/ [n] [G]	Superimposes string-2 onto string-1 for n lines. If G is omitted, only the first occurrence of string-1 on each line is modified, otherwise all occurrences of string-1 are modified.
MOVE buffer-1 buffer-2 string	Move string or contents of buffer-2 into buffer-1.
• NEXT [n]	Moves the pointer n lines forward or backward (default n=1).

• NFIND string	Moves the pointer down to next line NOT beginning with string.
NFIND(n) string	Moves pointer to next line in which string does not start in column n.
OVERLAY string	Superimposes string on current line. Use tabs to start in middle of line. An ! forces a space in its corresponding column.
PAUSE	Returns to operating system without changing the Editor state. Type START to continue.
POINT line-number	Relocates the pointer to line-number.
• PRINT n	Prints the current line or n lines beginning with the current line.
PSYMBOL	Prints a list of current symbol characters and their function.
PTABSET tab-1...tab-8	Provides for a setup of tabs on devices that have physical tab stops.
PUNCH { (ASR) / (PTP) } n	Punches n lines on high- or low-speed paper-tape punch.
• QUIT	Returns control to PRIMOS.
RETYPE string	Replace current line by string.
SYMBOL name character	Changes a symbol name to character. Current default values are:

Name	Default Characters
KILL	?
ERASE	"
WILD	!
BLANK	#
TAB	\
ESCAPE	^
SEMICO	;
CPROMPT	$
DPROMPT	&

• TABSET tab-1...tab-8	Sets up to eight logical tabstops to be invoked by the tab symbol.
• TOP	Moves the pointer one line before the first line of text.
• UNLOAD filename n	Copies n lines into filename.
UNLOAD filename TO string	Unload lines from current file into filename until string is found.
• VERIFY	Displays each line after completion of certain commands. (default.)
• WHERE	Prints the current line number.
XEQ buffer	Executes the contents of buffer as a command line. See MOVE.
* n	Repeat symbol. Causes preceding command to be repeated n times as in: F /;D;*1Ø which deletes the next ten lines beginning with /. If n is omitted, the command repeats until the bottom of file is reached.

Listing Programs

Techniques for listing files are also discussed earlier in this section under Examining Files.

Terminal Listing: Programs may be listed at the terminal by the PRIMOS command:

```
SLIST treename
```

where treename is the name of the file to be listed. Upon completion of the listing, control is returned to PRIMOS.

Line Printer Listing: To obtain a copy of a source file on the system line printer, enter the command:

```
SPOOL filename [-option-1...-option-n]
```

which creates a copy of the user's file filename in the line printer spool queue. The options are mnemonics specifying printer options. The most useful options for COBOL programmers are:

-LNUM	Prefixes a line number to the left of the file contents; these numbers are enclosed in parentheses.
-DEFER 'time'	Defers printing of the file until the specified time. The time may be entered in 24-hour format (13:Ø5) or 12-hour format (9:25 PM).

After a file has been spooled; the system returns the message:

```
YOUR SPOOL FILE IS PRTxxx
```

where xxx is a 3-digit number identifying the file on the spool queue. If a file has been spooled in error, it may be removed from the spool queue by the command:

```
SPOOL -CANCEL PRTxxx
```

where xxx is the identifying number of the spooled file.

The contents of the spool queue may be examined by the command:

```
SPOOL -LIST
```

A complete description of the SPOOL COMMAND with all its options will be found in the documentation on the PRIMOS system.

Renaming and Deleting Program Files

Renaming Program Files: Program files can be renamed or deleted in the manner of other files. Use the PRIMOS-level command, CNAME:

```
CNAME oldname newname
```

where oldname is the current name of the file and newname is the new name of the file. The user must have owner status in the UFD in order to use this command.

Deleting: Program files can be deleted with the PRIMOS-level command:

```
DELETE filename
```

where filename is the name of the file to be deleted; the user must have owner status in order to use this command.

Note

You cannot use the DELETE command to delete a UFD, sub-UFD, or segmented runfile (see Section 7).

SECTION 5

COMPILING A SOURCE 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.

Object code generated by the compiler in 64R mode is in a format suitable for loading by Prime's Linking Loader (LOAD) (see Section 6). The COBOL compiler can also generate object code in the segmented-addressing (64V) mode suitable for processing by Prime's segmented-addressing loader (SEG) utility on Prime 400 (or higher) computers.

USING THE COMPILER

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

```
COBOL Treename [-parameter-1 -parameter-2 ... -parameter-n]
                              or
COBOL [-parameter-1 ...] -I treename [... -parameter-n]
```

where treename is the treename 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.

EXAMPLE:

```
COBOL MYPROG -64V -L PRGLST

          or its equivalent

COBOL -64V -I MYPROG -L PRGLST
```

The mnemonics, e.g., -64V, 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 either as the first expression following the command COBOL, or as -I treename, but not both.

End of Compilation Message

After the compiler has done a pass at the specified input file, and generated code and listing output to the devices specified by the mnemonic parameters, it prints a message at the user's terminal. The message formats are:

64R mode

```
xxxx ERRORS    yyyy WARNINGS    (VER 04)
```

64V mode

```
xxxx ERRORS    yyyy WARNINGS P400/500, COBOL  VER 14.0 <PROGRAM>
```

where 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 syntax, an omission or the like 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.

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 G.

[ ] when stated, this is a variable describing the problem.

EXAMPLE:

```
112:UNRESOLVED PROCEDURE NAME; STATEMENT DELETED. [READ-PAYROLL]
```

An in-line error message takes the format:

```
** SYNTAX ERROR ** variable - in-line-message
```

Compiler Warning Messages

The general format of the message is:

line#/W/message

where line# 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 G.

EXAMPLE:

150/W/MOVE ID DONE WITHOUT CONVERSION.

Program Statistics (64V Mode Only)

When programs are compiled in 64V mode, program statistics are appended to the listing. These statistics relate to storage allocations. They take the form:

EXECUTABLE CODE SIZE: (in words)

CONSTANT POOL SIZE: (in words)

TOTAL PURE PROCEDURE SIZE: (in words)

WORKING-STORAGE SIZE: (in bytes)

TOTAL LINKFRAME SIZE: (in words)

STACK SIZE: (in words)

The trace mode status is given by (on or off).

TRACE MODE:

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 yyyy is the number of lines in the source program.

COMPILER FUNCTIONS

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

- Specify Input/Output Devices

  BINARY
  INPUT
  LISTING

- Addressing Mode

  64R
  64V

- Enable Expanded Listings (64V mode only)

  EXPLIST
  NOEXPLIST

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

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	Define input file/device (example -I TEST).
-I treename	The source program file is treename.
-BINARY	To override default, define binary (object file device.
-B treename	The binary file will be created with the treename 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.

NOTE: If the BINARY parameter is not included in the command line, it is equivalent to -B YES.

-LISTING	To override default, define listing file.
-L treename	The listing file will be created with the treename 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 is printed at the user's terminal.
-L SPOOL	The listing file is spooled directly to the line printer.

NOTE: If the LISTING parameter is not included in the command line, it is equivalent to -L YES.

Addressing Mode

-64R	Generates relative-addressed code suitable for loading with Prime's Linking Loader for the Prime 300 or higher.
-64V	Generates segmented-addressed code suitable for loading with SEG's loader. This mode must be used for programs exceeding 64K words, and/or for programs intended to be loaded as shared procedure. Code is suitable for execution on a Prime 350 or higher.

Listings

There are two forms of listing; regular and expanded.

The regular listing consists of source code with line numbers appended for reference purposes. This may be obtained in both 64R and 64V mode by the mnemonic parameter -NOEXPLIST.

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

The expanded listing is a combination of a regular listing and machine-generated code. The expanded listing is only valid for compilation in 64V mode; it may be obtained by the mnemonic parameter -EXPLIST.

-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. The label format is:

<TYPE>$HHHH[+N Character Offset]

HHHH = is the HEXADECIMAL IDENTIFIER

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

Other labels used:

SB% = Stack base relative - used for temporary storage
XB% = Temporary base relative - used linkage section address
WRKST$ = Working storage
WSEXT$ = Working storage extension, etc. under indexes, tallying and work area as needed by the compiler.

FOR EXAMPLE:

```
003233:      001310      EAFA      1,Z$0027+72C
003234:001000,000725L
```

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.

In order to utilize this expanded listing, a knowledge of PMA is necessary (see: PDR3059, PMA User Guide).

A complete list of all the compiler mnemonic parameters with more detailed comments on the consequences of their usage will be found in Section 21.

SECTION 6

LOADING AND LINKING

INTRODUCTION

The Prime Linking Loader utility (LOAD) operates on code produced by the COBOL compiler in the 64R mode; code produced in the 64V (segmented addressing) mode must be processed by the SEG utility (Section 7).

The Linking Loader combines into an executable program a number of program units or subroutines which have been independently compiled. Some of the subroutines may reside in a library; the Linking Loader provides the facility for incorporation of any library subroutines which have been referenced in the main program, as well as resolving the addresses between them.

Prime's Linking Loader offers the following features:

- The loader is capable of loading code anywhere within the 64K in which it resides, except on top of itself or in its symbol table.
- The location of COMMON is moveable by a keyboard command. (COMMON)
- Partial or full load maps can be displayed on the user terminal or written to a disk file. (MAP)
- An indefinite number of base areas can be specified; the loader automatically uses the first available area which can be reached, in preference to the sector 0 linkage area. (AUTOMATIC)
- The user can specify the instruction execution hardware available in the CPU on which the loaded program will execute. This is coordinated with the UII object blocks in load modules so that the proper UII library routines will load automatically. (HARDWARE) (UII - Unimplemented Instruction Interrupt)
- The user can execute the program from the keyboard in the loader without having to return to the PRIMOS command level. (EXECUTE)

Desectorization

The loader performs a function during loading called desectorization. The need for this function arises because one-word memory reference instructions cannot directly reference all of memory. The loader compensates for this by generating a pointer to the operand in a base area and then modifies the instruction to reference through the pointer.

The pointer default base area is from memory location '200 to '777. For many programs, this area is sufficient. However, for larger programs this area might be inadequate. The loader has a number of commands to enlarge the default base area to create local base areas (SETBASE and AUTOMATIC).

The base area below location '1000 can be used to desectorize any instruction, no matter what its location. Local base areas (above location '1000) can be used only to desectorize instructions in a window around the local base area. The window extends approximately '400 locations above and below the base area. (See Figure 6-1.)

The loader uses local base areas when possible in preference to a base area below location '1000. The location in base areas used by the loader is not available for any other use during program loading or execution.

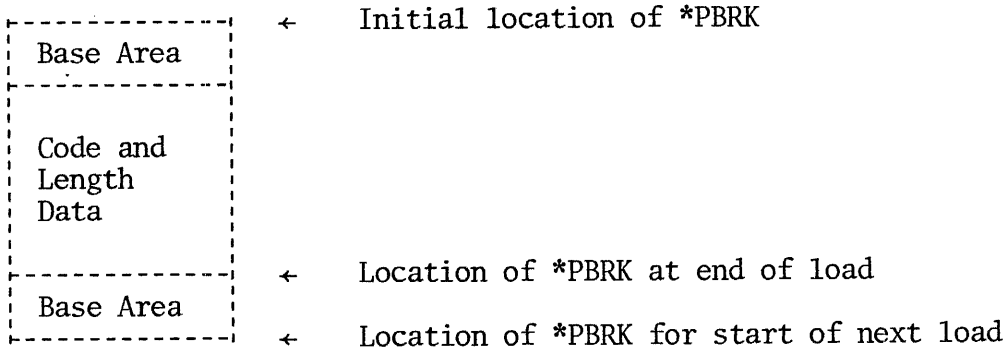


Figure 6-1. Base Area Orientation

Clearing the User Address Space

The PRIMOS level command FILMEM clears the user address space (for non-segmented programs). It is suggested that this command be invoked prior to the first use of the Linking Loader and after unsuccessful loading attempts. FILMEM will clear the user address space and assure the user of a clean start.

The command format is:

```
FILMEM      (RMODE (Prime 300 or higher) 32K))

  or

FILMEM ALL  (RMODE (Prime 300 or higher) 64K))
```

and has the result below:

Command	Operating System	
	PRIMOS II	PRIMOS III, IV, V
FILMEM	Clears locations '100 to '47777	Clears locations '100 to '77777
FILMEM ALL	Clear all user space	Clears locations '100 to '177777

When FILMEM is employed prior to loading, all memory locations will initially be set to zero. If no other characters are ever moved to the data area, the zeroes will remain, possibly as unwanted characters.

INVOKING THE LOADER

When the COBOL program is using sequential files (non-MIDAS), the Loader is invoked by the PRIMOS command:

```
LOAD
```

This loads the Linking Loader into locations '60000 to '63777 in the user's address space. When the COBOL program uses indexed and relative files employing MIDAS, the Loader should be invoked by the PRIMOS command:

```
HILOAD
```

This loads the Linking Loader into locations '174000 to '177777. Except for the relocation, HILOAD is identical to LOAD as far as the user is concerned.

USING THE LOADER UNDER PRIMOS

All loader functions are available through user terminal keyboard commands. When the LOAD (or HILOAD) command is typed, the Linking Loader is in command; the loader prints the $ prompt character on the user terminal and awaits a command line.

EXAMPLE:

```
LOAD
$
```

The $ prompt character means that the loader is in command mode until a QUIT command is received. (The QUIT command returns control to PRIMOS level). Each prompt character may be followed by a loader command, according to the command definitions. After each successful execution of a command, the loader types the $ prompt character. If the load is complete (i.e., there are no missing routines or modules) the loader will return the message LC, indicating that all external references have been satisfied.

EXAMPLE:

```
OK, LOAD                invoke loader
GO
$ LO B←TEST             load object program
$ LI COBLIB             load COBOL library
$ LI                    load FORTRAN library
LC                      load is complete will be returned by the Loader
$ QUIT                  ready for next command
OK,
```

If an error occurs in the loader itself during an operation, a two-letter error code will be printed followed by the $ prompt character. Loader error messages and suggested handling techniques are discussed immediately following the discussion on most frequently used loader commands.

When a system error (FILE NOT FOUND, etc.) is encountered, the loader prints this system error and returns prompt symbol ($).

NOTE: The loader also accepts commands from a command file. Comments may be used in this file; an asterisk (*) is the first character of a comment line.

Example of a Command File:

```
* COMMAND.FILE.TO.LOAD.THE.LOADER
FILMEM
* INVOKE LOADER.
LOAD B←PRGM
LI COBLIB
LI
SAVE *PRGM
QUIT
* INSTRUCT.COMPUTER.TO.READ.NEXT.FROM.TERMINAL
CO TTY
```

COMMAND FORMATS

Each loader command consists of a command name followed by a series of arguments, in the same format as the PRIMOS command line:

```
COMMAND name-1  name-2  arg-1  arg-2 . . . arg-n
```

where COMMAND is the command name, each name is a text string, and each Arg is an octal number of up to six octal digits.

Long filenames (up to 32 characters) are supported; treenames may not be used. Command names may be abbreviated to two characters. Arguments are separated by spaces. Up to three alphanumeric fields (non-numeric first) and nine (numeric only) arguments are allowed. In many cases, it is possible to omit arguments. (If any argument is included all arguments to the left of it in the command line must also be included).

A Complete list of the LOAD commands is given below. (Underlining indicates minimum required abbreviation).

Command	Function
ATTACH	Attach to different UFD
AUTOMATIC	Automatic generation of base areas
COMMON	Relocate common address
EXECUTE	Direct program execution

Command	Function
FORCELOAD	Unconditionally loads object files
HARDWARE	Hardware definition
INITIALIZE	Reinitialization (default load state)
LIBRARY	Loads library files (i.e., object files in UFD=LIB)
LOAD	Loads object files
MAP	Generates Load state map
MODE	Select addressing mode
QUIT	Returns command to PRIMOS
SAVE	Saves loaded memory image
SETBASE	Defines a new linkage area
VIRTUALBASE	Relocates base sector
XPUNGE	Deletes symbol

Loader Commands

It is convenient to discuss the loader commands under three categories:

1. Commands the programmer uses quite often:

   MODE (mostly with MIDAS files)
   COMMON (mostly with MIDAS files)
   LOAD
   LIBRARY
   SAVE
   QUIT
   MAP
   EXECUTE

2. Commands the programmer uses less often, usually in response to specific program requirements (as overflowing memory, etc.)

   AUTOMATIC
   INITIALIZE
   ATTACH

3. Commands designed for the use of the systems programmer. These are normally of very little use to the applications programmer. They are described in the PMA Programmer's Guide, PDR 3059.

   FORCELOAD
   VIRTUALBASE
   XPUNGE
   SETBASE
   HARDWARE

Most Frequently Used Loader Commands

MODE — MODE parameter

Specifies which of the CPU addressing modes the Loader is to use.

File Type	Parameter
MIDAS	D64R
non-MIDAS (small program)	D32R (Default)
non-MIDAS (large program)	D64R

NOTES:

1. D32R is the default parameter. It is not necessary to use a MODE command, as the Loader is in the mode when it is invoked.

2. If a program loaded with the default mode parameter causes a memory overflow (MO) error, it is too large for 32K and should be reloaded with the mode set using a MO D64R command. The MODE command, when used, should precede any other command.

COMMON — COMMON Address

Moves the top or starting location of FORTRAN-compatible COMMON to the address specified. This is done before loading any object modules. COMMON is a FORTRAN concept and is usually of no concern to the COBOL applications programmer. However, the COBOL libraries use the FORTRAN library, which, in turn, requires a COMMON area. When MIDAS files are used, the COBOL library COBKID location would interfere with the normal location of the FORTRAN library COMMON. The programmer therefore moves COMMON with the command:

```
CO 50000
```

LOAD

LOAD Filename

Loads an object file into memory. Filename is the name of the binary object file. The file name usually is of the form B₋Program-name.

LIBRARY

LIBRARY [Filename]

Temporarily attaches to the LIBRARY UFD, loads the specified file, and returns to the original UFD.

Filename is the name of the library file to be loaded; if omitted, the FORTRAN library FTNLIB is loaded.

The normal library loading order is:

MIDAS	non-MIDAS
LI COBKID	LI COBLIB
LI	LI

NOTE: LI is equivalent to LI FTNLIB.

SAVE

SAVE Filename

Saves the loaded memory image under the name Filename in the current UFD. Filename is the name under which the memory image is to be stored.

NOTE: Prime's convention is to use * as the first character of the Filename for the stored memory image. The user is not restricted to this convention.

QUIT

QUIT

Returns to the operating system command level with the user attached to the home UFD or the last UFD specified in an ATTACH command (see ATTACH). If the loader has opened a MAP file, it is closed at this time (see MAP).

Examples of load (user input is underlined):

MIDAS files

```
OK, HILOAD                  invoke Loader
$ MO D64R                   set mode
$ CO 50000                  move COMMON out of the way
$ LO B←SAM                  load COBOL object file
$ AU 20                     (see AUTOMATIC command)
$ LI COBKID                 load COBOL MIDAS library
$ LI                        load FORTRAN library
LC                          load is complete
$ SAVE *SAM                 save memory image
$ QUIT                      return to PRIMOS
OK,
```

non-MIDAS files

```
OK, LOAD                    invoke Loader
[$ MO D64R]                 [set mode if program is large]
 $ LO B←SAM                 load COBOL object file
[$ AU 20]                   [if program is large - see AUTOMATIC]
 $ LI COBLIB                load COBOL library
 $ LI                       load FORTRAN library
 LC                         load is complete
 $ SAVE *SAM                save memory image
 $ QUIT                     return to PRIMOS
 OK,
```

Less Frequently Used Loader Commands

Such commands are generally used for one or more of the following reasons:

1. Solving a specific problem in loading a program (see Loader error messages);
2. Optimizing the loading of a program;
3. Portability between different levels of Prime computers;
4. Added convenience to the programmer.

Load state parameters and their starting values are given in Table 6-1 below:

Parameter	Definition	Value at Start of load
*LOW	The lowest location in memory loaded	177777
*HIGH	The highest location in memory loaded	0
*START	The location at which execution will begin	0
*PBRK	The next location in memory to be loaded	1000
*CMLOW	The lowest location in COMMON	XX777
*CMHIGH	The highest location in COMMON	XX777
*SYM	The lowest location used by the symbol table	YY000
*UII	The net hardware/UII package requirement (see HARDWARE command for meaning)	0

Table 6-1. Load State Definition

NOTE: XX = Last Sector in loader occupied by loader

YY = First Sector occupied by loader

AUTOMATIC

AUTOMATIC XXXXXX

Causes the loader to insert a base area of length XXXXXX whenever the loader detects the end of a routine and more than 300 (octal) locations have been loaded since the last base area was inserted.

The value of XXXXXX may be changed between load files. This automatic feature is turned off with an AU 0 command.

AUtomatic helps to reduce the number of memory references through sector 0 by supplying base areas between load modules.

INITIALIZE

INITIALIZE [Filename]

Initializes the loader and then optionally performs the same actions as a LOAD command. In the loader's initialized state, the load state parameters (Table 6-1) return to their default values. If no Filename is provided, the loader repeats its prompt character ($). This allows the programmer to restart a LOAD session without the necessity of returning to the PRIMOS level and re-invoking the loader.

ATTACH

ATTACH [UFD] [Password] [Ldisk] [Key]

Attaches to different UFD's.

UFD: Any User File Directory. However, the user is attached to the home UFD when no UFD name is specified.

Password: The user gets owner status if he gives the owner password and non-owner status if he gives a non-owner password. The password parameter is necessary only when the UFD is password-protected.

Ldisk: If the Ldisk parameter is omitted, the loader searches only device 0 for the specified UFD. If an Ldisk value of '100000 is specified, the file system searches all initiated devices in logical unit order.

Key: The values for Key most likely to be useful during loading are:

0 Do not change home UFD (Default).

1 Adopt named UFD as home UFD.

2 Attach to sub-UFD in the current UFD; do not set as home.

3 Attach to subUFD in the current UFD; set as home.

If key was specified as 0 or 2, the user may return to the home UFD by entering AT.

The ATTACH command allows the programmer to load program modules stored in different UFDs without the need of explicitly copying these program modules into the UFD invoking LOAD.

NOTE: The LIBRARY command automatically attaches to the library UFD in order to load the library module and then re-attaches to the UFD in which LOAD was invoked.

MAP

MAP [Filename] [option]

Lists a load map. Filename is the name of the map to be opened, and option is an octal value which selects one of four map options. The loader will close the map file(s), if any, at the end of the load session.

Option Number	Load Map Information
None	Load state, base area, and symbol storage map; symbols sorted by address (full map).
1	Load state only
2	Load state and base area
3	Unsatisfied references only

Map Option 1 - Load State Map

The load state map identifies:

1. The lowest and the highest storage memory locations;

2. The location at which the program execution begins;

3. The next location available for loading;

4. The high and low common area;

5. The lowest location used by the symbol table;

6. The net hardware UII package requirement.

These parameters are printed in the load state map with a corresponding storage address (See Table 6-1).

Load State Map 1

```
OK, LOAD
GO
$ LO B←SIMP
$ LI COBLIB
$ LI
LC
$ MA 1
*START  001000  *LOW     000200  *HIGH  006512  *PBRK  006513
*CMLOW  063777  *CMHIGH  063777  *SYM   057401  *UII   000001
```

Map Option 2 - Load State Map and Base Area Map

The base area map includes the lowest, highest, and next available locations. Each line contains four addresses as follows:

```
*BASE   XXXXXX  YYYYYY   ZZZZZZ  WWWWWW

XXXXXX = Lowest location defined for this area
YYYYYY = Next available location if starting up
           from XXXXXX
ZZZZZZ = Next available location if starting down
           from WWWWWW
WWWWWW = Highest location defined for this area
```

The base area map includes a load state map:

Load State and Linkage Area Map 2

```
$ MA 2
*START  001000  *LOW     000200  *HIGH  006512  *PBRK  006513
*CMLOW  063777  *CMHIGH  063777  *SYM   057401  *UII   000001

*BASE   000200  000220   000777  000777
*BASE   001527  001571   001570  001570
*BASE   002515  002557   002556  002556
*BASE   003404  003427   003434  003435
```

MAP Option 3 - Unsatisfied References Only

Lists the labels and external reference names which have been referenced but not loaded.

Unsatisfied References Only MAP 3

$MA 3 (No unsatisfied references, therefore no printout)

MAP Option Number Omitted - Full Map

A full map contains all components of a load map including a full symbol storage listing.

The symbol storage listing consists of every defined label or external reference name, printed four per line in the following format:

namexx NNNNNN

or

Namexx NNNNNN**

NNNNNN is a six-digit octal address. The ** flag means the reference is unsatisfied (i.e., has not been loaded). Every map begins with a reference to a special FORTRAN COMMON block LIST, defined as starting at location 1.

Load State, Linkage Area and Instruction Storage Map

```
$MA
*START  001000  *LOW     000200  *HIGH    006512  *PBRK   0065
*CMLOW  063777  *CMHIGH  063777  *SYM     057401  *UII    00000

*BASE   000200  000220   000777  000777
*BASE   001527  001571   001570  001570
*BASE   002515  002557   002556  002556
*BASE   003404  003427   003434  003435

LIST    000001  F$WA     001020  F$WX    001026  F$IO    001102
F$A1    001501  F$A3     001501  F$A2    001505  F$A5    001505
F$A6    001512  F$CB     002034  F$IOBF  004660  F$ER    004762
F$HT    004767  AC1      005047  AC2     005050  AC3     005051
AC4     005052  AC5      005053  WRASC   005054  IOCS$   005061
IOCS$T  005160  F$AT     005172  F$AT1   005174  WATBL   005237
LUTBL   005256  PUTBL    005313  RSTBL   005350  0$AD07  005405
```

Load maps may be sent to a file instead of the user's terminal.

EXAMPLE:

This example illustrates how the loaded memory image can be SAVEd as a file (RUNFIL) in the UFD, and a Load Map stored in a file MAP1.

```
OK, LOAD                    invoke loader
GO
$ LO B←SIMP                 load object file
$ LI COBLIB                 load COBOL library
$ LI                        load FORTRAN library
LC
$ MA MAP1                   send map to file MAP1
$ SA RUNFIL                 save loaded memory image
$ EX                        execute program

TEST MESSAGE                output of program
```

Filename RUNFIL is now stored in the current UFD, and Filename MAP1 contains the map.

```
OK, SLIST MAP1
GO
*START  001000  *LOW     000200  *HIGH  006603  *PBRK  006604
*CMLOW  063777  *CMHIGH  063777  *SYM   057374  *UII   000001
```

EXECUTE

EXECUTE

Enables the user to start execution of the loaded program. Execution starts at the location shown by the *START entry of the load map.

LOADER ERROR MESSAGES

Message	Meaning
CM	Command error. Illegal command format.
* GT	Group Type error. The loader has encountered an unrecognizable piece of object text. Loading is discontinued. If object module is COBOL, make sure that it was compiled without errors. The source module is not an object file (output of FTN, PMA, etc.) or is a segmented-address object file (64V).

* MI xxxxxx — Multiple Indirect. While linking in 64R mode, the loader attempted to add indirection to an already indirect instruction at location xxxxxx. The contents of xxxxxx are the proper flag, tag, and object.

* MO — Memory Overflow Errors.

As users' programs become larger, MO (memory overflow) errors become more frequent. This section contains a description of the several typical causes of these errors and suggested solutions to these causes.

When MO error occurs, the user should do a 'MA 2' and examine the map for any of the following possible situations (see MAP):

a. The address of the bottom of the symbol table (*SYM) is at or close to *PBRK. This indicates that there is not enough room below the loader for the whole program. HILOAD will probably solve the problem - assuming the user is not already using HILOAD.

b. (For P400/500 only) The program and data are too large to fit into 64K of memory. The program modules should be recompiled in 64V mode and loaded using SEG (see Section 7).

N6 — Never 64R mode. Code is being loaded in 64R mode, which will not execute properly. Loading is discontinued.

Recompile or reassemble the source files in 64R mode, or remove a D64R command from the load session, or look for a PMA module which has set the load mode to 64R (see MODE).

* NOTE: These are hard errors; the load process cannot be renewed. Correct errors and begin the load process anew.

SECTION 7

LOADING SEGMENTED PROGRAMS

INTRODUCTION

This section describes the use of SEG, which is Prime's utility module for loading, modifying, and running segmented programs. A segment can be up to 64K word block of user's virtual address space. Segment '4000 is that segment which SEG and other external commands occupy when invoked. SEG creates a runfile of up to 15 or 31 segments. (Check with the systems manager to determine which version has been implemented.)

PRIMOS assigns memory segments to a user as they are accessed. These are not released until logout. Since only a fixed number of segments are available for all users, additional segments should not be invoked unless the user is actually executing or examining a segmented program. Most of the functions of SEG use only one segment; only those options which restore a runfile use extra segments, i.e., RESTORE, RESUME, and EXECUTE.

SEG must perform many of the operations on segmented runfiles which are performed on relative-addressed runfiles at the command level or by the Linking Loader. Since the nature of SEG runfiles differs from that of the relative-addressed runfiles, separate SEG commands are required.

Segmented Runfiles

A segmented runfile consists of segment subfiles in a segment directory. For this reason, the reader cannot delete a SEG runfile with a PRIMOS-level DELETE command; instead, use the DELETE command in SEG. (The TREDEL command in FUTIL can also be used to delete a SEG runfile, but it operates much more slowly than SEG's DELETE.) Each segment of the runfile consists of 32 ('40) subfiles of '4000 words each. Subfile 0 of the runfile is used for startup information, the load map, and the memory image subfile map. Memory image subfiles begin in segment subfile 1. Only the subfiles actually required for the runfile are stored on the disk.

SEG's Loader

SEG has a virtual loader (i.e., it loads to a file rather than memory) which requires the name of the runfile before anything is loaded. The runfile may be new, or it may be a previously used SEG runfile; it can be in any UFD. An old unsegmented SAVE file cannot be used.

As the symbol table is always available, SEG's loader may be used to add modules to an existing runfile. Similarly, a partial load may be saved with the SEG SAVE command and the load completed later. In addition, selected modules may be replaced in a SEG runfile.

Functional Structure of SEG's Loader

SEG's Loader has three types of commands:

1. Commands which load object files;

2. Commands which override the Loader's defaults ("how", "where", "what", "how much", "from where");

3. Commands which perform operations with the current state of the load and/or with SEG itself (e.g., getting a load map, executing the program.)

Type 1: Commands which load object file (LO, LI, RL, PL, IL)

These commands all have the possibility of having modifiers included in their command line. These modifiers are never used in the basic SEG load sessions. For the most part, only LO and LI are needed.

Modifiers are:

A. Prefixes - P/, S/, D/, F/

B. Three numeric field suffixes

The form of these modifiers is exactly the same for all loading commands.

Type 2: Commands which override Loader Defaults (AT, A/SY, R/SY, SY, SP, ST, XP, OP, CO)

Each of these commands requires an argument list unique to itself. These commands are never required in the basic SEG Load session.

Type 3: Commands operating with the current state of LOAD or SEG (MA, SA, EX, IN, QU, RE)

One or more of these commands is necessary to complete the load and leave the Loader in an orderly manner. The most useful commands are EX, SA, MA, and QU. Some of the type 3 commands have optional arguments; no arguments are required in the basic SEG Load session.

Object File As Input

The object file of the program modules must have been created using 64V mode of the COBOL compiler. Modules written in other languages may also be loaded, if they have been compiled or assembled properly.

Data consists of all COMMON blocks and link frames. Code and data are loaded in separate segments to support re-entrant procedures. The Loader assigns code and data segments. The first segment ('4001) is used for code. Usually segment '4002 will be used for data. The Loader loads data and code into appropriate segments and opens new segments as required. (It is possible to put both data and procedure in the same segment to save space. Care is required not to create an incorrect load.)

The Stack

The Loader assigns a stack (which is a dynamic work area) when SAVE is invoked. The stack is usually assigned as the next free location in the first procedure segment with '6000 free words. If no such segment exists, a new data segment will be assigned with the first location in the stack set to 4. The user may force the location of the stack and/or may change its size. (See the Loader's STACK command and the Modification sub-processor's SK command.)

SEG Commands

When invoking one of SEG's functions, the form of the command is:

```
COMMAND Fname-1 Fname-2 Par-1 Par-2 Par-3 ....
```

Fname-1 is the filename or the treename of the file to be accessed. Tree-name enables files outside the current UFD to be accessed. SEG remembers the name, and if the name is not changed, it becomes the default. If no current file name has been established, SEG will request a treename. In order to reference a new runfile, any SEG command may be invoked with a new Filename-1. The nature of the other parameters depend on the function.

A complete list of SEG commands is given below. Those commands discussed in this section are preceded by the greater than character (>). Those commands discussed in the shared code section are preceded by the plus sign (+). Permissable abbreviations are underlined. Commands not flagged require a knowledge of PMA and/or are specifically designed for in-house use.

	Command	Function
>	DELETE	delete a SEG runfile
>	HELP	print a list of SEG commands at user's terminal
>	MAP	generate a load map
>	MODIFY (SAVE)	invoke modification sub-processor
>	NEW	write new copy of SEG runfile to disk
	PATCH	modify save range of existing segment
>	RETURN	return to SEG command level
>	SK	alter stack size and/or location
>	START	change program execution start address
	WRITE	rewrite all segments to disk (to preserve patches)

	Command	Function
>	PSD	invoke VPSD debugging utility
>	QUIT	return to PRIMOS command level
>	RESTORE	bring SEG runfile into user memory
>	RESUME or RESUME	restore SEG runfile and begin execution
+	SHARE	write shared code and data into separate files
+	SINGLE	create RMODE file image of single segment
>	TIME	print time and date of last runfile modification
>	VLOAD (LOAD)	define runfile and invokes loader for creation
>	VLOAD * (LOAD*)	define runfile and invokes loader for appending
>	ATTACH	attach to another UFD
+	A/SYMBOL	define a symbol in memory and reserve space for it using absolute segment numbers
+	COMMON ABS	relocate COMMON using absolute segment numbers
>	COMMON REL	relocate COMMON using relative segment assignment
>	D/**	perform load using previous parameters
>	EXECUTE	save load to disk and execute program
+	F/**	forceload all routines in object file
+	IL	load the impure FORTRAN library
>	INITIALIZE	initialize and restart SEG's loader
>	LIBRARY	load library file (UFD=LIB)
>	LOAD	load object file (user UFD)
>	MAP	generate loadmap
	OPERATOR	relax/impose high level restrictions
+	PL	load the pure FORTRAN library
	P/**	load on a page boundary
>	QUIT	return to PRIMOS command level
+	RETURN	return to SEG command level
>	RL	reload a routine
>	R/SYMBOL	define a symbol in memory and reserve space for it using relative segment assignment
>	SAVE	save load to disk
+	SPLIT	break segment in data and procedure portions
>	STACK	change stack size
>	SYMBOL	define a symbol at a specific location in memory
+	S/**	expunge symbols from symbol table; delete base information
+	XP	

For clarity, the user may prefer to use command names in full rather than in abbreviated form. This will not adversely affect SEG's operation.

Vestigial Commands

A number of commands exist whose functionality have been superceded, either by improvements in SEG, improvements in PRIMOS itself, or for increased clarity. For compatibility with previous revisions, these commands are still supported and will perform exactly as before. However, they will no longer be documented.

Typing these letter combinations will not generate error messages, but users cannot be certain of the result. Do not use them.

Commands at SEG level:	LO, LO *, PA, SA
Commands in the loader:	AS, FO, SH
Commands in the Modification subprocessor:	A, B, EN, KE, X

SEG Messages

When a load is complete, i.e., all references have been satisfied, SEG's Loader prints the message LC at the user's terminal.

The message COMMAND ERROR and a new prompt character will be printed at the user's terminal in response to an unrecognized command or a command format error. The SEG Loader also has a series of error messages which will be printed at the terminal. These are listed in Appendix H, along with probable causes of the errors and suggestions for correcting or eliminating them.

USING SEG

SEG is a command under CMDNCO; the COBOL programmer will invoke SEG in one of two ways:

1. SEG Filename - where Filename is the filename (or treename) of a SEG runfile. This command loads the runfile into segmented memory and starts execution. This is analogous to the R Filename command for programs loaded with Prime's linking loader (see Section 8 - Execution).

2. SEG - accesses the SEG commands allowing the user to load, modify, and/or execute a SEG runfile. These are discussed in this section.

SEG displays a # on the terminal as a prompt character; the Loader and Modification subprocessors display a $ as a prompt character to solicit subcommands.

Command Files

SEG accepts commands from a command file.

NOTE: Command file comments, i.e., commands of the form:

```
* THIS.IS.A.COMMENT
```

are supported only in SEG's loader. Use of comments in any other portion of SEG will give a non-fatal COMMAND ERROR and a prompt character.

Filenames

SEG supports both long filenames and treenames. Treenames conform to the PRIMOS standard with one exception. If a password is required to obtain access, the entire treename must be preceded and followed by single quotes.

EXAMPLE:

An object file SECRET in UFD CYPHER is protected by the password CRYPTO. To load such a file, the command would be structured:

```
$LOAD 'CYPHER CRYPTO > SECRET'
```

(where user input is underlined)

If a command is given and a SEG runfile name is required, the request

```
SAVE FILE TREENAME:
```

will be printed out. The user should enter a SEG runfile filename (or treename).

The first time a SEG runfile is entered, it is remembered by SEG and becomes the established runfile name. In most commands, it is then unnecessary to reference any SEG runfile if the established one is meant. This remains the established runfile name unless a new SEG runfile name is established by the user. (This is discussed under each specific command.)

Frequently Used and Essential Commands - Applications Functions

The commands herein outlined are presented in the order in which they would normally be used.

HELP

HELP

Prints a list of the SEG commands at the user's terminal.

VLOAD

VLOAD [filename]

This command accesses the SEG loader. Filename is the filename (or treename) of a SEG runfile. If filename is omitted, the established runfile will be used. If file name as specified is the name of an existing SEG runfile, that runfile will be reinitialized before control is passed to the loader.

NOTE: Prime's convention is to use # as the first character of a SEG runfile name (e.g., #TEST). Although the system does not require this, the user should follow this convention unless there are compelling reasons not to do so.

The VLOAD (or VLOAD *) command performs three functions:

1. Defines (explicitly or implicitly) the name of the SEG runfile.

2. Specifies whether a new file is to be written or an existing file is to be added to.

3. Transfers operations to the SEG Loader. The SEG Loader prints the prompt character $ to differentiate itself from SEG-level commands.

The Loader has a large number of subfunctions. Most of these subfunctions, specifically designed for use in creating very large applications packages, shared procedures, and Prime in-house systems, will probably be of little consequence to most users. Frequently-used Loader commands are discussed below in their most common form.

LOAD

LOAD filename

Where filename is the filename (or treename) of the file to be loaded. Usually filename will be of the form B←Prgname. The file should be an object file created by the COBOL compiler with the 64V option. If filename is not given, or is an incorrect type (not an object file), an error will be generated.

The Loader will process the object file, making it part of the runfile being created, and linking it to other modules already loaded. All questions of memory management are handled by the Loader.

NOTE: If a treename is used, the Loader remains attached to the UFD (or sub-UFD) in which that file resides. The user must explicitly re-attach to the original UFD if desired, by typing AT in response to the $ prompt.

LIBRARY

LIBRARY [filename]

Where filename is the name of the file in UFD=LIB which is to be loaded into the runfile. The file filename must be one containing object text compiled (or assembled) in 64V mode; if not, an error will be generated. If filename is not supplied, the FORTRAN library files PFTNLB and IFTNLB will be used. The Loader will then process the library file in the same manner as LOAD processed object files. In most cases, any libraries needed are loaded after other object files.

NOTE: LOAD and LIBRARY are part of the Loader's family of load commands. Both may be modified by optional numeric parameters and/or command modifiers S/, F/, D/, to give the user greater control over placement of modules in the runfile. These options are described later in Sections 11 and 12.

MAP

MAP 3

This command prints a list of the unsatisfied references (i.e., procedures called which have not been loaded) at the user's terminal. This command is especially useful if the user does not get the LC (Load Complete) message from the Loader. Loadmaps are discussed in detail in Section 11.

SAVE

SAVE

This command saves the result of the load by writing all buffers out to the runfile on the disk. A location for the stack is assigned at this time. (A MAP command prior to SAVE will show no stack assigned; a MAP command afterwards will give the assigned location of the stack.

EXECUTE

EXECUTE

First SAVEs the program, if necessary, then executes it. After execution, control returns directly to PRIMOS. An EXECUTE command may follow a SAVE command.

QUIT

QUIT

Returns the user to PRIMOS command level. QUIT does not SAVE the runfile. To keep the established runfile, perform a Loader SAVE prior to QUITting.

EXAMPLE:

The user has compiled a main program, MAIN; a subroutine in a separate source file SUBR has also been compiled. Both have been compiled in 64V mode using the default object filenames. They could be loaded as follows (user input is underlined):

```
OK, SEG                    bring SEG into memory
GO
# VLOAD #MAIN              invoke the Loader and establish a runfile
$ LO B←MAIN                load the main program
$ LO B←SUBR                load any separately compiled subroutine
$ LI VCOBLB                load the COBOL library
( $ LI VKDALB              load this system library if MIDAS files
                              are used )
$ LI                       load the FORTRAN library
LC                         Loader indicates all references are satisfied
$ SAVE                     user saves runfile
$ QUIT                     return to PRIMOS level
OK,
```

DELETE

DELETE filename (1)

or

DELETE (2)

Where filename is the name (or treename) of a SAVE SEG runfile. This command deletes the SEG runfile filename (1) or the currently established runfile (2).

NOTE: Do not attempt to delete a SEG runfile with the PRIMOS level DELETE command. It will delete the segment directory, but not the subsidiary files in the directory, which you then cannot delete. If necessary to delete a runfile outside the SEG utility, use FUTIL'S TREDEL command.

SECTION 8

EXECUTING THE LOADED PROGRAM

INTRODUCTION

This section treats the following topics:

- Execution of program memory images saved by the Linking Loader (64R).
- Execution of segmented runfiles saved by SEG's Loader (64V).
- CM$L (64R)/C$IN (64V) utility programs.
- Run-time error messages.

EXECUTION OF PROGRAM MEMORY IMAGES SAVED BY THE LINKING LOADER (64R)

Execution of a COBOL program in 64R mode is performed at the PRIMOS level using the RESUME command:

OK, RESUME *filename

where *filename is the name of the file containing the saved memory image from the loading process and is in the current UFD to be executed.

RESUME brings the memory-image program *filename from the disk into the user's memory, and begins execution of the program after a dialogue with CM$L (see below).

The START command allows programs to be executed which have been made resident in the user's memory by a previous RESUME command. This is usually occasioned by a STOP literal statement in the COBOL program.

The START command is given as:

OK, START

The program resumes at the address value at which execution was interrupted.

EXAMPLE:

```
OK, R *PRGRM                Begin execution
GO
    .
    .
    .
QUIT                        User hit CTRL/P to stop.
OK, S                       Restart program from last point of
GO                          execution.
    .
    .
    .                       Execution restarted
```

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

For a complete discussion of these commands, see the PRIMOS Interactive User Guide, MAN 2602.

EXECUTION OF SEGMENTED RUNFILES SAVED BY SEG'S LOADER (64V)

Execution of a COBOL program in 64V mode is performed at the PRIMOS level using the SEG command:

```
OK, SEG #filename
```

where #filename is the filename (or treename) of a SEG runfile. SEG loads the runfile into segmented memory and begins execution of the program after a dialogue with C$IN (see below). SEG should be used for runfiles created by SEG's loader; it should not be used for program memory images created by the Linking Loader.

EXAMPLE:

```
OK, SEG #PRGRM              Begin execution
GO
    .
    .
    .
OK,                         Program complete; PRIMOS requests next
                            command.
```

CM$L (64R)/C$IN (64V) UTILITY PROGRAMS

Immediately following the execute commands of RESUME for 64R mode and SEG for 64V mode, a series of questions will be asked concerning run-time file assignments. These questions are prompted by the utility programs CM$L for 64R mode, and C$IN for 64V mode. To the user, there will be no noticeable difference between the two.

The utility programs will ask on the terminal:

```
ENTER FILENAME AND UNIT
```

All succeeding lines will begin with the prompt character >. 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 treename desired. 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 dialogue with CM$L or C$IN would be:

```
ENTER FILENAME AND UNIT
>FILE1 = PETERS>T1
        or
>FILE1 = $MT1, S, T1, ØØØØØ1
```

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

The second statement requires MAG TAPE unit one to be assigned, with the tape mounted to contain a TAPE-ID of T1 and a volume serial of ØØØØØ1.

The utility programs CM$L and C$IN will do all pre-screening of the files and display the prompt character > while waiting for user input. There should be one entry for each FD in the program. When no files remain to be entered, the single slash character (/) will conclude the session. Execution of the program will begin, using the file assignments which were just entered.

Disk Formats (Filenames and Treenames)

A treename in a disk format entry is an extended form of the filename, which describes the location of the file in the directory structure. Filenames and treenames may be of the following forms:

1. FILE-ID=UFDNAME [password] [logical disk number (octal)]

2. FILD-ID=* > filename

3. FILE-ID=filename

4. FILE-ID=<volumename>UFDNAME [password] > filename

Everything to the right of the equal sign follows the rules for TREENAME formation (see PE-T-341 for detailed treatment of TREENAMEs).

In 1 above, the volume with the specified logical disk number is searched for the specified UFDNAME.

In 2 above, the current UFD is the starting UFD.

In 3 above, the current UFD is searched for the specified filename.

In 4 above, the volume with the specified name is searched for the specified UFD name. If the volume name is a single asterisk (*), the MFD in the current volume is searched.

Tape Format

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

MAGTAPE: $MT(X) 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 tape is being read. Label must have been specified as S.

TAPE NUMBER: is a 6 character field which is checked at open-time when reading a tape, but is not needed when creating a tape.

CM$L/C$IN Error Messages

The following are error messages which may be output by the CM$L or C$IN utility programs:

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)
INVALID STANDARD/NON LABEL (non S or N)
TAPE NAME GREATER THAN 17
TAPE NUMBER GREATER THAN 6

RUN-TIME ERROR MESSAGES

Alphabetic lists of both RMODE and VMODE run-time error messages are available in Appendix G.

SECTION 9

SORT PROCEDURES

EXTERNAL/INTERNAL SORT ROUTINES

Various utilities are available to effect COBOL sort procedures. These include external and internal methods as outlined below:

NOTE: The ANSI Sort-Merge Module is not supported by Prime COBOL.

- External operating system COBOL sort procedures
- Internal application sort subroutines
- Sort considerations

External Operating System COBOL Sort Procedures

The External Sort utility of the Prime Operating System (PRIMOS) is easily accessed by a COBOL program. First, the user must specify the point in a program at which a sort is to be done. This is accomplished in the Procedure Division by employing a STOP statement at the desired location, followed by any valid literal.

EXAMPLE:

```
PROCEDURE DIVISION.
BEGIN-PROGRAM
   PERFORM CREATE-FILE THRU FILE-CREATED.
   STOP 'READY FOR EXTERNAL SORT'.
STATE-TWO
   PERFORM ADDRESS-CHANGE
       .
       .
       .
```

At this point, control will shift from COBOL execution to the operating system command level. The user will then enter an interactive session on the terminal.

In the following dialogue example, all underlined items must be typed by the user on the terminal:

SAVE *TEMP 1/777777	This will save memory image locations of all necessary address registers in a file named *TEMP.
SORT	This command invokes the SORT utility program.
GO SORT program parameters are: Input File Name - Output File Name followed by pairs of starting and ending columns.	The operating system responds with this documentation on the user terminal.
INFILE OUTFILE 2	This entry specifies the input file name to be sorted, and the output file; both must be resident within your UFD. The 2 indicates the number of columns to sort on.
Input pairs of starting and ending columns one per line. For reverse sorting enter "R" after ending columns.	
1 5 15 25 R	This specifies that the columns to be sorted are columns 1 through 5, with a reverse SORT on columns 15 through 25.
Beginning SORT Passes 3 items 2010	The computer responds, indicating the SORT has begin and providing PASS and Item data.
OK,	OK, indicates the SORT is complete. Control is returned to PRIMOS.
RESTOR *TEMP	This command will restore the memory image address register locations of the previously saved file.
START	This command will return control to the next source line of the application program, which immediately follows STOP literal.

NOTE: The interactive dialogue above may be established as a COMMAND file.

Internal Application Sort Subroutines

SUBSRT is a sort subroutine available to a COBOL program through a CALL STATEMENT. It is particularly effective and efficient when sorting 3000 or fewer records. For larger applications, its simple calling sequence and Data Division Entries may outweigh time considerations.

1. Calling Sequence:

   The calling sequence for SUBSRT contains eight required parameters. Use of this call may appear as follows:

   ```
   Call 'SUBSRT'  using SORT-INPUT-FILE, SORT-OUTPUT-
                  FILE, SORT-PAIRS, SORT-START-COLUMN,
                  SORT-END-COLUMN, SORT-PASSES, SORT-
                  ITEMS.
   ```

   Any valid COBOL data-names may be used.

2. Data Division Structure

   Using the above data-names the following DATA-DIVISION entries would be used.

   ```
   02 SORT INPUT FILE PIC X(6) VALUE 'SORTIN'.
   02 SORT-OUTPUT-FILE PIC X(6) VALUE 'SORTOT'.
   02 SORT-PAIRS            COMP VALUE 1.
   02 SORT-START-COLUMN     COMP VALUE 1.
   02 SORT-END-COLUMN       COMP VALUE 35.
   02 SORT-PASSES           COMP.
   02 SORT-ITEMS            COMP.
   ```

Using the CALL sequence outlined above in 1, and the related Data Division entries described in 2, the following would occur:

An input file by the name of 'SORTIN' would be stored in Columns 1 through 35, with the sorted file being designated as output "SORTOT". The number of passes and items sorted would be returned to the user from the "SUBSRT" utility.

Data-name parameters of the calling sequence above are defined as follows:

- SORT-INPUT-FILE

  The actual file system name of the block of records to be sorted must be placed within this six character field.

- SORT-OUTPUT-FILE

  The actual file system name for the sorted output file must be placed within this six character field. It may be the same name as the input file and may also be a file previously used by the COBOL application.

- SORT-PAIRS

  This field must be specified as computational and must contain the value of the total number of pairs of columns on which the subroutine will sort.

- SORT-START-COLUMN

  This field must be specified as computational and must contain the value of the column on which to begin the sort.

- SORT-END-COLUMN

  This field must be specified as computational and must contain the value of the column on which to end the sort.

- SORT-PASSES

  This field must be specified as computational, with no VALUE clause. This is a returned argument, stating how many passes the utility took to complete the sort.

- SORT-ITEMS

  This field must be specified as computational but cannot contain a VALUE clause. This is a returned argument stating how many lines (or records) were actually sorted.

Sort Considerations

Job analysis for SORT utility selection should take into account a variety of factors. These include file size, processing mode, data type specifications, command file specifications, loading factors, etc.

As previously mentioned, internal sort subroutine, SUBSRT, is particularly efficient when sorting 3000 or fewer records. Time efficiency decreases as record number increases. This should be considered when determining the most efficacious sort for the user's application.

SUBSRT is not available for 64V mode.

The PRIMOS External Sort utility allows specification of data type. That is, ASCII, Binary, Single Precision Integer, Single Precision REAL, Double Precision REAL may be specified. SUBSRT permits no such specification.

Command files used in conjunction with the External sort should be started on a unit greater than 6. The system default for running a command file is unit 6, however, SORT may also open unit 6. This conflict can result in the error message 'PRWFIL, UNIT NOT OPEN! This problem is avoided if a command file relating to a sort application is started on a unit greater than 6.

When using the internal SUBSRT subroutine, the SORT library must be loaded with the object program. The Linking Loader commands to accomplish this are as follows (underlined entries indicate user requirements, $ indicates loader prompt):

OK, HILOAD	call loader
GO	
$ COMMON 1ØØØØØ	set common
$ MODE D64R	
$ LOAD B←filename	load object program file
$ LIB COBLIB	load COBOL library
$ LIB SRTLIB	load SORT library
$ LIB FTNLIB	load system subroutines
LC	the system will respond with LOAD COMPLETE
$ SAVE *filename	save the loaded program
$ QUIT	quit the loader; return to operating system

PART III

ADVANCED CONCEPTS

SECTION 10

COBOL PROGRAM ENVIRONMENTS, EXPANDED

INTRODUCTION

INTERACTIVE

COMMAND FILES

PHANTOM USERS

CX MODE

The portions of SECTION 10 outlined above were incomplete at this printing.

SHARED PROCEDURES

The following steps should be taken to create and load programs as shared procedures: (Each step will later be considered in detail.)

- Determine whether shared procedure is applicable and desirable
- Write source code. Program must be identified with PROGRAM-ID where the program name must be MAIN.
- Load to the runfile using the SEG Loader's. Debug the program.

  With this information, initialize and load to the runfile, splitting procedure and data portions of programs.
- Load for shared procedure and return to SEG command level.
- Separate out segments below '4001 into separate RMODE runfiles using SEG's SHARE command.
- Incorporate runfiles below '4000 into segments for sharing using PRIMOS' SHARE command.

APPLICABILITY

In general, programs which are small, or which will normally only be run by one user at a time, are not candidates for shared procedure. Programs which are expected to be run by many operators simultaneously, especially large procedures which use relatively small amounts of data, are excellent candidates for shared procedures. Examples of the latter type include Prime's Shared Editor or a user-written order entry system.

The advantages of shared procedures are:

- Only one copy of code is necessary for all users
- Decreases restore time
- Program is more likely to be in cache memory; operation is much faster for multiple users
- Decreased memory usage, reducing paging

Once it is determined that a program will be loaded as shared procedure, the programmer must obtain from the system manager the segment numbers which are to be used for the particular program being loaded. Currently, segments '2000 to '2037 are available as public shared segments. Some of these segments may be occupied by Prime-supplied programs. For example, if the Shared Editor is installed, it will reside in segment '2000.

System Considerations for the Manager

Public shared segments are a large but finite resource; their allocation should be made carefully and only for those programs which will benefit by being loaded as shared procedure. It is possible to incorporate more than one program in the same segment; the manager is responsible that no conflict will exist from overwriting, etc.

CAUTION

> The public shared segments are re-initialized in a cold start of PRIMOS. The systems manager should include in the cold start command file the PRIMOS SHARE commands necessary to reload these segments. This also means the system manager must maintain copies of the SEG runfiles for each program.

SOURCE CODE

The main program which is loaded first must be identified with a PROGRAM-ID clause as MAIN.

COMPILING

The source program is compiled with the 64V mode option; this produces code to be loaded with SEG.

LOADING

Loading for shared procedure is a multi-phase process. The aim is to obtain an optimized load with the program operating properly as designed. It will be instructive to follow an example illustrating some general principles.

Consider a program BENCH, with 3 large COMMON blocks AA, BB, and AABB. The FORTRAN library is required. The simplest load, using SEG's defaults would be: (user input underlined)

```
   OK, SEG              invoke SEG
   #VL #PGRM            establish runfile and access loader
   $LO B←PGRM           load main program
   $LI VCOBLB           load COBOL library
(  $LI VKDALB           for MIDAS files)
   $LI                  load FORTRAN library
   LC                   load is complete
   SA                   save result
   MA MAPFIL            generate a map in file MAPFIL to be examined
   $QU                  return to PRIMOS
   OK,
```

At this point the program will be executed and, if necessary, debugged. The number of segments used can be decreased by moving the location by moving the location of COMMON blocks and the stack. The load would be: (user input underlined)

```
OK, SEG                          invoke SEG
#VL #PGRM                        establish runfile and access loader
$SY AA 4000 60000                locate COMMON block in Segment '4000
                                 above SEG
$SY BB 4002 1000                 put BB in segment '4002
$SY AABB 4001 10000              put AABB in segment '4001
$LO B↩PGRM                       load user program
$LI                              load FORTRAN library
LC                               load complete
$SA                              save load
$RE                              return to SEG command level
#MO                              invoke Modification subprocessor
$SK 4001 170000                  place stack above AABB in segment
                                 '4001 and assign it '170000 locations
#RE                              return to SEG command level
#MA * MAPFIL                     get a loadmap
#QU                              return to PRIMOS command level
```

Since the user has taken over some of SEG's functions, he must check the loadmap to see if the load is reasonable. It would not be amiss at this point to be certain that the program executes properly.

CAUTION

Relative assignment numbers and absolute segment numbers must not both be used in the same load.

LOADING FOR SHARED CODE

Loading for shared code requires the capability of being able to separate the procedure frame from the linkage frames. This capability exists in the advance functionality of the loader commands. Other commands in the loader allow placing of COMMON and other symbols using absolute segment numbers, expunging defined symbols from SEG's symbol table, and forceloading.

The loader also allows segments to be split into procedure and data portions to conserve segments and/or to load into segment '4000 the RMODE Interlude program RUNIT. RUNIT allows the segmented program to be invoked as an RMODE program from the user's UFD or installed in UFD=CMDNCØ. These commands will be discussed later in this section.

SPLIT segno addr Note 1.

or

SPLIT addr Note 2.

Breaks a segment into procedure (lower) and (upper) portions. This operation conserves segments. It also allows the loading of RUNIT as an aid to creating shared programs.

Segno is the absolute octal segment number.

Addr is the location of the split in the segment. Addr must be a multiple of '4000.

NOTES:

1. Splits the segment into procedure and data portions as specified; used to decrease number of segments used.

EXAMPLE:

SP '4000 '10000 - splits segment '4000, with locations below '10000 for procedure and rest of the segment for data.

2. This is the form used for shared procedure. Segment '4000 is assumed. In addition to splitting the segment, the interlude program RUNIT is loaded (in 64V mode) beginning at location '1000.

No data or procedure may be assigned to locations above '172000 in segment '4000, as this is where RUNIT places its stack.

After splitting, RUNIT and RESUME will exist in SEG's symbol table. RUNIT is the normal starting address; RESUME may be used as a starting address if the existing stack is to be preserved.

NOTE: Once a segment has been split, it is addressable only specifically, i.e., with the S/xx or P/xx command (or with D/xx following an S/xx or P/xx command). Loading must use absolute segment numbers. See S/xx, D/xx, P/xx.

CAUTION

SEG's Loader does not keep track of split segments and may assign the stack to the top of the procedure portion of a split segment. This may cause problems if there is not enough space between the end of the procedure portion and the start of the data portion.

A/SYMBOL

A/SYMBOL sname [segtype] segno size

where:	sname	is the name of the symbol.
	segtype	is the type of segment, either DATA. or PROCEDURE; if omitted, a data segment is assumed.
	segno	is the absolute octal segment number.
	size	is the number of locations to be reserved for the symbol if omitted; 0 is assumed.

This command places a symbol and reserved 0 or more locations in memory for it. If the segment specified does not exist, it will be created.

CAUTION

The user must verify that the number of locations reserved for the symbol are adequate for its intended use, and that there is actually sufficient room in the segment for the size specified.

This command may not be used to satisfy unsatisfied references already existing in the load.

Example: (TOP+1 is the next available location in a given segment.)

A/SY KELVIN 4002 1000	place symbol KELVIN at the current TOP+1 in data segment '4002 reserving 1000 (octal) locations for it.
A/SY KELVIN PR 4001 1000	place symbol KELVIN at current TOP+L in procedure segment '4001 reserving 1000 (octal) locations for it.

The example above illustrates one way of placing a COMMON block in a procedure segment.

A/SY KELVIN DA 4001 1000	place symbol KELVIN at current TOP+1 in data segment '4001 reserving 1000 (octal) locations for it.

If the segment specified above did not exist, it would be created and the address of KELVIN in it would be 0. (A special case of TOP+1.)

COMMON ABS segno

Where segno is the absolute octal segment number into which COMMON will be loaded.

When loading into specific segments, this command should be used to specify the COMMON segment either as the one into which the link frames are loaded, or another if there is some reason to move COMMON away from the link frames.

CO ABS 4015

Will cause the Loader to load all COMMON into segment '4015 so long as it will fit, then into segment '4016, '4017, etc. This bypasses SEG's normal default segment assignments.

CAUTION

> Since SEG's normal defaults are bypassed by this command, it is the user's responsibility to be certain that segments being reserved for loading COMMON have not been reserved for other uses.

ADVANCED FUNCTIONALITY OF THE LOADER'S FAMILY OF LOADING COMMANDS

The complete family of loading commands are:

LOAD	load an object file (user UFD)
LIBRARY	load a library object file (UFD=LIB)
RL	reload an object module
PL	load the PFTNLB file (UFD=LIB)
IL	load the IFTNLB file (UFD=LIB)

PL and IL load pure and impure FORTRAN libraries, respectively. (Relative segment assignments may be used with PL and IL, but there would rarely be a need for this.) Relative and absolute loading must not be mixed in the same load.

Modules may be loaded into specific segments for procedure and link frames by use of the S/ prefix modifier.

The command format is:

S/xx [filename] addr psegno lsegno

where xx is LO, LI, RL, PL, or IL.

If LO or RL is used, filename is mandatory.

If LI is used, filename is optional. (Omission loads PFTNLB and IFTNLB.)

If PL or IL is used, filename should be omitted.

Addr is the starting load address in the procedure segment.

An addr of 0 is interpreted as start loading at the current pointer position in the procedure segment. This is the usual value.

Psegno is the procedure segment number.

Lsegno is the data linkage segment number.

Both psegno and lsegno are absolute (octal) segment numbers; both must be supplies. When loading shared code, procedure will be loaded in segments '2000 - '2037 as allocated by the system manager.

As with the load into relative segment commands, the segments required will be created if they do not already exist. If a required segment runs out of room, the next segment in sequence will be created and used to continue the load. For example, if the user has declared psegno to be '2000 and segment '2000 becomes too full for the next routine to be loaded, segment '2001 will be created as a procedure segment and the load will precede in segment '2001. Note that some smaller routines may subsequently be loaded in segment '2000. The S/xx modifier does not place COMMON areas; this should be done using the CO ABS command prior to the load.

EXAMPLE:

S/LO B←JUNK 0 2000 4002 - Load object file B←JUNK with its procedure beginning at the current load pointer location in segment '2000 and its data linkage areas beginning at the current load pointer in segment '4002. Previously COMMON was located with a CO ABS command.

S/IL 0 4000 4000 - Load the impure portion of the FORTRAN library into the split segment '4000.

As with the relative assignment numbers, the D/ modifier prefix may be used.

EXAMPLE:

```
S/LO B←BENCH 0 2000 4000
D/PL
```

is equivalent to

```
S/LO B←BENCH 0 2000 4000
S/PL 0 2000 4000
```

CAUTION

When using this modifier (S/) some of SEG's checking mechanisms are overridden. Therefore, the user must carefully examine the loadmap to make sure there is no inconsistency or confusion.

The S/ modifier may not be combined with the D/ modifier either as D/S/xx or S/D/xx.

Forceloading

When a file is loaded, normally only those routines referenced by previously loaded modules (or by routines in the library) are loaded. When building templates or creating partial loads, it is often desirable to force all routines in a file to be loaded. Forceloading in SEG's Loader is accomplished with the F/ modification prefix as:

F/xx (filename) [addr psegno lsegno] Note 1.

or

F/S/xx (filename) [addr psegno lsegno] Note 2.

where xx is one of the loading commands, LO, LI, RL, PL, or IL.

Filename is the filename (or treename) of the object file. It is mandatory for LO and RL, optional for LI, and should be omitted for PL and IL.

Addr is the start address for forceloading in the procedure segment.

Psegno is the procedure segment number.

Lsegno is the data segment number.

NOTES:

1. This is a simple forceload of the object file filename. Both psegno and lsegno are relative assignment numbers. The defaults resulting if parameters are omitted are the same as for the commands without the F/ prefix.

   EXAMPLE:

   F/LO B←THINGS - forceload all modules in B←THINGS in default segment.

   F/LI - forceload all the FORTRAN library in default segments.

2. Forceloads object file to specific segments. Both psegno and lsegno are absolute (octal) segment numbers (see S/xx for details). This format would be used for forceloading shared procedures.

   EXAMPLE:

   F/S/PL 4000 2000 4002 - Forceload all of the procedure of the FORTRAN library PFTNLB beginning at location '4000 in segment '2000 with linkage area in segment '4002.

NOTE:

S/F/xx is identical to F/S/xx.

The D/ prefix may be combined with F/.

```
S/LO B←BENCH 0 2001 4002
F/S/PL 0 2001 4002
```

is equivalent to

```
S/LO B←BENCH 0 2001 4002
F/D/PL
```

RETURN

Returns the user to the SEG command level. This command does not SAVE the runfile; the user should perform the SEG SAVE subcommand before the RETURN if the established runfile is to be kept. After loading for shared procedure has been completed, the load must be SAVED; control returned to the SEG level and SEG's SHARE command invoked.

SPLITTING OUT

After the load has been completed, the portions of the SEG runfile corresponding to segments below '4001 must be transformed into RMODE runfiles using SEG's SHARE command. These files are similar to the relative addressed mode save files having a conventional save file header. No files are created for segments above '4000. If segment '4000 exists and it includes RUNIT (see SPLIT), it may be executed at PRIMOS command level. The command format is:

SHARE [filename]

Filename is the filename (or treename) of the SEG runfile. If omitted, the established runfile name is split out.

The RUNIT interlude program sets the correct addressing mode; starting location and registers are set to the standard default values.

SEG responds to the SHARE command by asking for a two-character ID. SHARE will use this ID to build the save files with the name yyxxxx, where yy is the ID given to SHARE, and xxxx is the segment number.

EXAMPLE: (user input is underlined)

```
#SH #TEST                    (use default values)
TWO CHARACTER FILE ID: BE
CREATING BE2000
CREATING BE4000
#                            (ready for next SEG command)
```

SINGLE

SEG's SHARE command creates an RMODE runfile for all segments below '4001. The SINGLE command creates an RMODE runfile for any specified segment, even those above '4000. This command is:

SINGLE [filename] segno

where filename is the SEG runfile name; if omitted, the established runfile is used.

Segno is the segment number to be used to create the runfile.

As in the SHARE command, the user is asked for a two character ID.

EXAMPLE: (user input is underlined)

```
#SI 4001
TWO CHARACTER FILE ID: IX
CREATING IX 4001
#
```

The SINGLE command only works for segments loaded with the S/xx command, including the RMODE interlude in the SEG runfile.

This method is of particular use in three cases:

1. The user's program has a small procedure part requiring a large data area.

2. The user has a large program, most of which is loaded below segment '4000 as shared procedure.

3. The user's program is primarily a 'transaction processing' system. Most of the user's (large) program can be loaded at LOGIN time, or is loaded below segment '4000 as shared procedure.

In case 1 the user will force all of the loaded portion of the program to reside in segment 4000. Unitialized COMMON blocks will be declared in other segments and need not be 'loaded' into memory.

In case 2 the user will load only the impure parts of the procedure (such as IFTNLB) into segment '4000 and will place all link frames and initialized COMMON in segment '4000.

In case 3 the external LOGIN program will load most of the user's SEG runfile (the portions residing above '4000) into memory at LOGIN time. The user's specific applications, referencing the fixed portions above and below '4000, will be loaded into segment '4000. This case requires the user to create a 'template' of the fixed portion of the application on top of which specific applications are loaded.

When the user's procedure is loaded with SEG's loader, segment '4000 is declared as a split segment using the loader's SPLIT command, and specifying only the location at which the segment is to be split. This causes SEG's loader to create a procedure area below the designated location, and a data link frame area above it. Then the RMODE interlude RUNIT is automatically loaded into the procedure portion. At run-time, RUNIT will initialize the stack, and transfer control to the user's routine, MAIN. The user may load other procedure and link-data information into segment '4000 using the loader's S/xx command.

The user must determine via a previous load where to split segment '4000.

A slightly different load sequence from that given earlier in this section:

```
OK, *
OK, * THE FOLLOWING EXAMPLE ILLUSTRATES USING SEG TO
OK, *  LOAD A NON-SHARED PROCEDURE.
OK, *
OK, SEG
GO
#VLOAD #DISPL.NON_SHARED
$ LO B_DISPL
$ LI VCOBLB
$ LI
LC
$ SAVE
$ MAP 6


$ MAP 7
*START  004002  000006  *STACK  004001  002404  *SYM    171716

SEG. #    TYPE      LOW       HIGH      TOP
004001    PROC##    001000    002403    002403
004002    DATA      000000    000325    000325


ROUTINE      ECB          PROCEDURE     ST. SIZE  LINK FR.
C$NCLT  4002  000306    4001  002216    000020    177706
MAIN    4002  000006    4001  001000    000072    177406


DIRECT ENTRY LINKS
  EXIT    4001  002374    TNOU    4001  002400

COMMON BLOCKS


OTHER SYMBOLS

$ QUIT
```

would load the program as non-shared procedure. The resulting RMODE runfile BE4000 can be invoked with the PRIMOS command RESUME as R BE4000 or it may be placed in the command UFD.

Finally, when the load is complete and saved, the user returns to SEG via the RETURN command and enters SH on the terminal. When all appropriate segments have been turned into separate runfiles, the one with the appended segment number '4000 may be run (suitably renamed if desired) from PRIMOS command level either from CMDNCO or by a PRIMOS RESUME command.

EXAMPLE:

Programmer has been assigned segment '2000 by the systems manager.

```
OK, *
OK, * THE FOLLOWING EXAMPLE DEMONSTRATES USING SEG TO
OK, * GENERATE A SHAREABLE PROCEDURE.
OK, *
OK, SEG
GO
# VLOAD #DISPL.SHARED_VERSION
$ SP 4000
$ S/LO B_DISPL 0 2000 4000
$ D/LI VCOBLB
D/PL
LC
$ S/IL 0 4000 4000
$ SAVE
$ MAP 6


$ MAP 7
*START  004000  004006  *STACK  004000  001616  *SYM    171556

SEG. #    TYPE      LOW       HIGH      TOP
004000    DATA##    004000    004431    004431
004000    PROC##    001000    001615    001615
002000    PROC      001000    002451    002451


ROUTINE      ECB          PROCEDURE     ST. SIZE  LINK FR.
C$NCLT  4000  004306    2000  002216    000020    003706
MAIN    4000  004006    2000  001000    000072    003406
T1IB    4000  004372    2000  002436    000012    003772
T1OB    4000  004412    2000  002444    000012    003772
T1OU    4000  004350    2000  002402    000016    003750
TONL    4000  004326    2000  002374    000012    003726


DIRECT ENTRY LINKS
  EXIT    2000  002422    TNOU    2000  002426    TNOUA   2000  002432
```

```
COMMON BLOCKS


OTHER SYMBOLS
  F$FLEX  4000  001174    RESUME  4000  001042    RUNIT   4000  001000

$ RE
# SH
TWO CHARACTER FILE ID: DI
CREATING DI4000
CREATING DI2000
# QU


OK,
```

INCORPORATING FILES INTO SHARED SEGMENTS

Using SEG's SHARE command creates one RMODE runfile for each segment of the SEG runfile below segment '4001. The RMODE runfiles for segments below '4000 must actually be incorporated into those segments using the PRIMOS SHARE command. This operation can only be performed at the system operator's console. The command format is:

SHARE filename segno access-rights

where filename is the name of the RMODE runfile to be incorporated into the segment.

Segno is the segment number to be shared.

Access-rights are the access rights assigned to this segment.

Access Rights	Permitted Operations
0	none
200	read
600	read and execute
700	read, write, and execute

Segments '1 to '12 and '2000 to '2037 are the current range of sharable segments; specification of segments other than these will give unpredictable results.

CAUTION

Since PRIMOS IV resides in segments '1 to '12, users should not create files which need to be incorporated into these segments.

If no value is specified, the default is '600.

The PRIMOS command OPR 1 must precede SHARE commands; OPR 0 must follow the last SHARE command.

EXAMPLE: (user input is underlined)

```
OK, OPR 1
OK, SHARE BE2000 2000
OK, OPR 0                          default access
```

The program BENCH can now be executed from the user's UFD by the command R BE4000 (the name of the RMODE runfile BE4000 may be changed if desired using the CNAME command).

```
CNAME BE4000 BENCH
```

The RMODE image of segment '4000 may also be put into the command UFD and invoked as a command.

```
OK, FUTIL                  invoke FUTIL
>TO CMDNC0                 define TO UFD
>COPY BE4000 BENCH         copy BE4000 into UFD=CMDNC0
                           under the name BENCH
>QU                        return to PRIMOS
OK,
```

It was not necessary to specify the FROM UFD; the default is the current UFD.

SECTION 11

MANAGEMENT SYSTEMS AND LANGUAGE INTERFACE

INTRODUCTION

This section discusses interfaces of the COBOL language to the following Prime systems:

- MIDAS, Multiple Index Data Access System
- DBMS, Database Management System
- FORMS, Forms Management System
- Other Programming Languages

MIDAS

Multiple (Keyed) Index Data Access System, MIDAS, provides a series of programs and subroutines for the creation and maintenance of keyed-index and/or keyed-index direct access (KI/DA) files.

Keyed-index files are sometimes referred to as ISAM (Indexed Sequential Access Mode) files. Prime COBOL utilizes MIDAS for its ISAM files.

NOTE: KI/DA is the file access method used by MIDAS. At present, MIDAS and KI/DA are identical.

Requirements

MIDAS usage requires that the UFD LIB contain the COBKID library (for non-segmented addressing use, 64R mode), and the VKDALB library (for segmented-addressing use, 64V mode). At load-time, these libraries are loaded just prior to loading the FORTRAN library.

Using MIDAS

MIDAS usage falls into four areas (see Figure 11-1).

- Creating/modifying the template - the user defines the data file, indices, etc. (CREATK)
- Building the data file - data existing in a text or binary file are converted to a MIDAS file. (KBUILD)
- Maintaining the file - data entries are added, deleted, changed, or viewed.

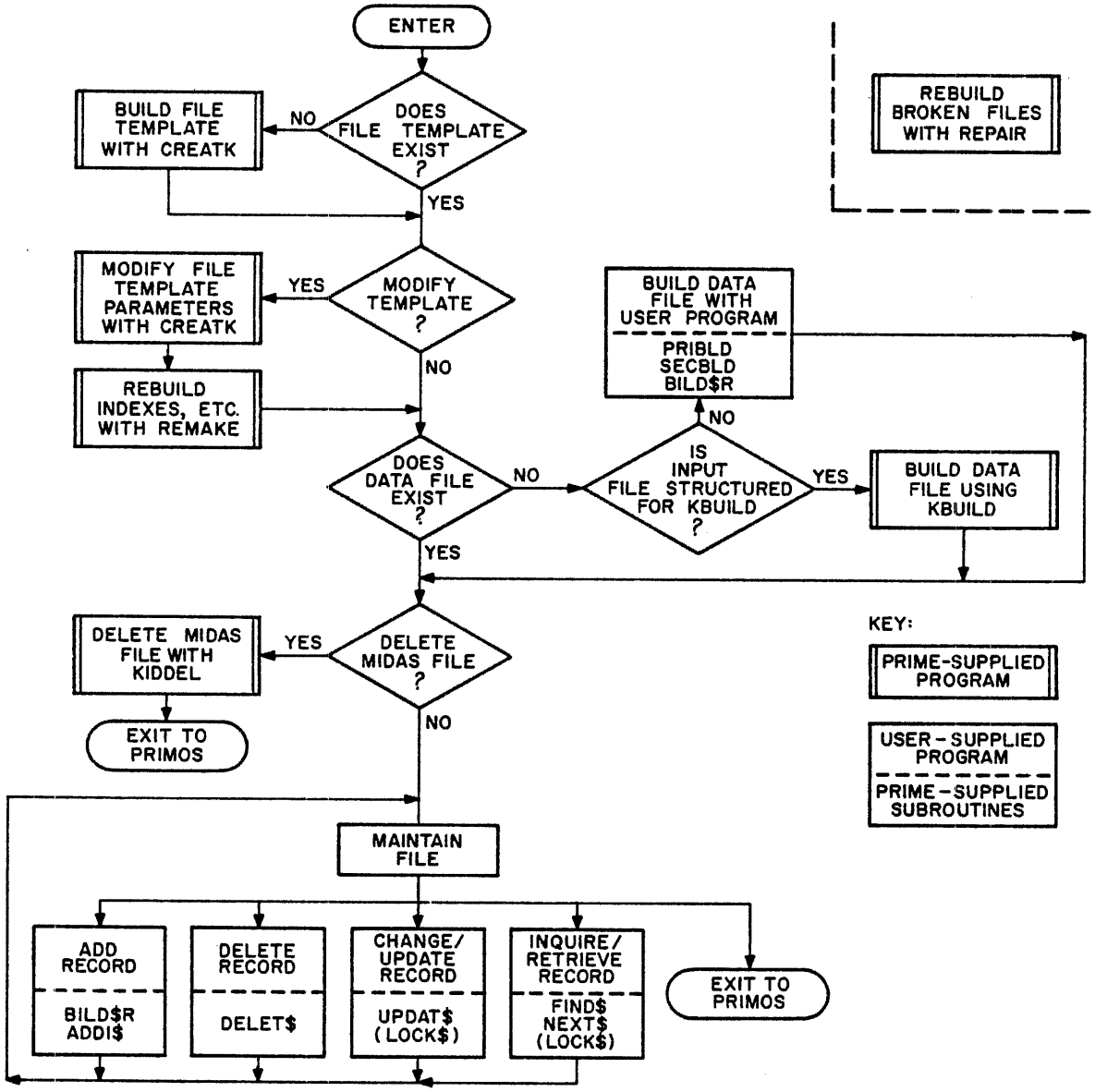


Figure 11-1. User's Functional Overview of the Midas File System

- Performing housekeeping - files are restructured after significant maintenance (REMAKE), deleted in part or full (KIDDEL), or rebuilt after crashes (REPAIR).

Maintenance of the file may be accomplished by more than one user simultaneously. A lockout subroutine protects data entries from attempts at simultaneous changes/deletions. All other operations require the user to have exclusive access to the MIDAS file.

The COBOL user will be most concerned with CREATK, REMAKE and KIDDEL; where after the initial dialogues later described, interaction with MIDAS is virtually transparent.

The Template

In order to initiate an Indexed or Relative file for Prime COBOL, the user must build a MIDAS Template File. This will minimally contain a segment directory, a file descriptor subfile, a one-level primary index subfile which contains the index descriptor block, and an empty last level index block. If the file is organized for direct access, data segments must be allocated and initialized. For each secondary index defined, there must be a corresponding index descriptor block and an empty last level index block.

Creating the Template (CREATK)

A template (file descriptor) for a keyed-index file can be created with the interactive utility program CREATK. The functions of CREATK are:

- Create a new file
- Modify index or data description for an old file
- Add new secondary indices to a file
- Display existing index or data descriptors.

When constructing a template, the user engages in an interactive dialogue.

Minimum Dialogue (user responses are underlined):

Prompt	Response	Remarks
OK,	CREATK	
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 Ldisk]>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 file name.
NEW FILE?	{ YES / NO }	If NO, see NOTE 2 at conclusion of dialogue.
DIRECT ACCESS?	{ YES / NO }	For a new relative file (goes to dialogue 2) For a new indexed file (goes to dialogue 1)

(Dialogue 1) Data Subfile Questions

(PRIME INDEX/RECORD KEY)

Prompt	Response	Remarks
KEY TYPE	B	
KEY SIZE=:	B number	Number is the number of bits in the primary key. It is equal to 8 times the number of characters in the key; e.g., 2 characters in a key = 16 bits. The maximum size for an indexed file is 32 characters, or 256 bits.
DATA SIZE=:	number	Number of words for a data record, where number equals the record length divided by 2. For COBOL programs, this includes the key size, and a remainder factor of 1 if it applies.

(SECONDARY INDEX/ALTERNATE RECORD KEYS - this section is repeated for each alternate record key.)

Prompt	Response	Remarks
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 DECLARATIVE section will be activated.

Prompt	Response	Remarks
KEY TYPE:	B	
KEY SIZE =:	B	Enter the number of bits in the key; use same formula as for primary index.
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 CREATK, or continue with alternate key specifications.

(Dialogue 2) Data Subfile Questions

KEY TYPE:	B	
KEY SIZE =:	B number	Number is the number of bits in the relative key; i.e., characters in the key X 8. The maximum size is 6 characters, or 48 bits. In sequential mode with no key, size must be specified at maximum: 48.
DATA SIZE =:	number	Number is the number of words in a data record: record length ÷ 2 plus the remainder factor of 1 if it applies.
NUMBER OF ENTRIES TO ALLOCATE? number		Number is the number of entries to allocate in the new KI/DA file. 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.

NOTES:

1. If an invalid response is entered by the user, the question (prompt) will be repeated.

2. If CREATK is not being run for a new file, and the response to the prompt NEW FILE? is NO, the succeeding prompt will be:

FUNCTION? { ADD / MODIFY DATA / PRINT / HELP / FILE / QUIT / USAGE / (CR) }

The response options to the FUNCTION prompt have the following significance:

ADD

Resulting dialogue is similar to the secondary index dialogue, except that an error message will be generated if the subfile already exists. The return at the end of the dialogue is to the prompt, INDEX NO.?

MODIFY DATA

This sequence allows the user to redefine the data. The length of a data entry may be changed (shorter or longer). It follows the Data Subfile Section above. At the end of the data dialogue, return is made to the prompt, INDEX NO.?

PRINT

Results in the prompt: INDEX NO.? { numeric, 0-17 / DATA }

The current configuration of the index subfile or data subfile given will be displayed on the user's terminal. The configuration displayed will be that in the file descriptor subfile. At the end of the display question, the prompt INDEX NO.? will be repeated.

HELP

The currently available options, and their functions, will be listed in the user's terminal.

FILE

This option will allow the user to specify a new working file without leaving and then re-entering CREATK. The program returns to the beginning of the dialogue with the prompt, FILE NAME?

QUIT

Exits

USAGE

This option will allow the user to display the number of entries currently available through any defined index. The number of entries are displayed as 'ENTRIES INDEXED', 'ENTRIES IN OVERFLOW', and 'ENTRIES DELETED'. These values are summed to provide 'TOTAL ENTRIES IN FILE'.

This option is of particular significance to the COBOL programmer. It indicates the state of overflow and helps determine the need for REMAKE.

REMAKE Program

This program can perform four levels of restructuring:

- Restructure selected secondary indices
- Restructure all indices
- Restructure all indices and data sub-file
- Rewrite file into new file with new template.

The programmer should run REMAKE after substantial numbers of data entries have been added to or deleted from the file. This restructuring clears out the index overflow areas (which when overloaded slow the searching process) and frees for use the space occupied by data entries flagged as deleted. See PDR 3061 Reference Guide, Multiple Index Data Access System (MIDAS).

KIDDEL Program

This program will delete all or part of a MIDAS file; the PRIMOS DELETE command should not be used for indexed files. KIDDEL allows deletion of:

- Selected secondary indices
- Unwanted segments at the end of the data sub-file
- The entire file

An example of an actual CREATK dialogue for sample program REF2 appears at the close of Section 16.

Complete information on programs outlined above, and KBUILD and REPAIR is presented in the Reference Guide, Multiple Index Data Access System (MIDAS) PDR 3061.

DBMS

For complete information relating to COBOL interface to Database Management System (DBMS), the user is referred to IDR 3046, COBOL Reference Guide to DBMS.

FORMS

The Prime Forms Management System (FORMS) provides a convenient and natural method of defining a form in a language specifically designed for such a purpose. These forms may then be implemented by any applications program which uses Prime's Input-Output Control System (IOCS), including programs written in COBOL. Applications programs communicate with FORMS through input/output statements native to the host language. Programs which currently run in an interactive mode can easily be converted to use FORMS. See PTU 45 and IDR 3040, Forms Management System (FORMS).

OTHER PROGRAMMING LANGUAGES

The reader is directed to Section 17, Inter-Program Communication.

PART IV

REFERENCE

COBOL CONCEPTS

REFERENCE

SECTION 12

FUNDAMENTAL CONCEPTS OF COBOL

DIVISIONS OF A COBOL PROGRAM: A SUMMARY

Every COBOL program consists of four divisions: Identification Division, Environment Division, Data Division, and Procedure Division.

- The Identification 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.

- The Environment Division specifies 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.

  The 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.

  The 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.

- 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.

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

  The Working-Storage Section describes records 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).

  The 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.

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

  This division contains two 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 procedures 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 procedures 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 completion, unless the rules indicate an exception.

  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 program component structure and order:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  program-name.
[AUTHOR.  comment-entry...]
[INSTALLATION. comment-entry...]
[DATE-WRITTEN. comment-entry...]
[DATE-COMPILED. comment-entry...]
[SECURITY. comment-entry...]
[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 ...]...]
[WORKING-STORAGE SECTION.
[data item description entry]...]
[LINKAGE SECTION.
[data item description entry]...]
PROCEDURE DIVISION [USING identifier-1...].
[DECLARATIVES.
{section-name SECTION. use-sentence.}
[paragraph-name. [sentence]...]...
END DECLARATIVES.]
[section-name SECTION.]
{paragraph-name. [sentence]...}...
```

The source program on the following pages, SAMPLE, illustrates program component structure and order. SAMPLE creates and reads a relative file sequentially.

A Listing File for SAMPLE is provided after the source program coding example. SAMPLE was compiled in 64R mode.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. SAMPLE.
       INSTALLATION. PRIME.
       REMARKS. THIS PROGRAM CREATES AND READS A RELATIVE FILE
                SEQUENTIALLY.
      *
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. PRIME.
       OBJECT-COMPUTER. PRIME.
       SPECIAL-NAMES. CONSOLE IS TTY.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT LIST-FILE ASSIGN TO PRINTER.
           SELECT CARD-FILE ASSIGN TO PFMS.
           SELECT DIRECTORY-FILE ASSIGN TO PFMS, ORGANIZATION
                  IS RELATIVE
                ACCESS MODE IS SEQUENTIAL
                FILE STATUS IS FILE-STATUS.
      *
      *
      *
       DATA DIVISION.
       FILE SECTION.
       FD  LIST-FILE, LABEL RECORDS ARE OMITTED.
           01  PRINT-LINE, PICTURE X(121).
           01  PRINT-REC.
               02  FILLER                  PIC X.
               02  PRINT-INPUT             PIC X(80).
               02  PRINT-ERROR             PIC X(40).
       FD  CARD-FILE, LABEL RECORDS ARE STANDARD
           VALUE OF FILE-ID IS 'INDATA'.
           01  CARD-IMAGE, PICTURE X(80).
       FD  DIRECTORY-FILE, LABEL RECORDS ARE STANDARD, VALUE OF FILE-ID
                  IS 'RELFILE'
                  OWNER IS 'LDAVIS'.
           01  DIRECTORY-RECORD.
               02  NAME.
                   03  LAST-NAME    PIC X(15).
                   03  FIRST-NAME   PIC X(15).
               02  FILLER  PICTURE X(1).
               02  ADDRESS PICTURE X(25).
               02  FILLER  PICTURE X(1).
               02  CITY    PICTURE X(4).
               02  FILLER  PICTURE X(3).
               02  PHONE-NO PICTURE 9(7).
               02  FILLER PICTURE X(9).
      *
```

```
WORKING-STORAGE SECTION.
77  FILE-STATUS, PICTURE X(2), VALUE IS SPACE.
01  HEADER.
    02  H1, PICTURE X(4), VALUE IS 'NAME'.
    02  FILLER, PICTURE X(27), VALUE IS SPACE.
    02  H2, PICTURE X(6), VALUE IS 'STREET'.
    02  FILLER, PICTURE X(20), VALUE IS SPACE.
    02  H3, PICTURE X(4), VALUE IS 'CITY'.
    02  FILLER, PICTURE X(3), VALUE IS SPACE.
    02  H4, PICTURE X(5), VALUE IS 'PHONE'.
*
*
PROCEDURE DIVISION.
BEGIN SECTION.
CREATE-FILE.
    OPEN OUTPUT LIST-FILE DIRECTORY-FILE.
    OPEN INPUT CARD-FILE.
        WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
READ-NEXT.
    READ CARD-FILE AT END GO TO
        LIST-DIRECTORY.
    MOVE CARD-IMAGE TO PRINT-LINE.
    MOVE CARD-IMAGE TO DIRECTORY-RECORD.
    WRITE PRINT-LINE.
    WRITE DIRECTORY-RECORD INVALID KEY
        DISPLAY 'INVALID KEY'.
    GO TO READ-NEXT.
LIST-DIRECTORY.
    CLOSE CARD-FILE, DIRECTORY-FILE.
    DISPLAY 'END TEST TO CREATE FILE'.
    OPEN INPUT DIRECTORY-FILE.
    PERMORM LIST THRU LIST-DONE.
LAST SECTION.
CLOSE-ALL.
    CLOSE DIRECTORY-FILE, LIST-FILE.
    DISPLAY 'END TEST SEQUENTIAL READ AFTER A START'.
    STOP RUN.
LIST.
    WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
READ-NEXT-DIRECTORY-RECORD.
    READ DIRECTORY-FILE NEXT RECORD, AT END
        GO TO LIST-DONE.
    MOVE DIRECTORY-RECORD TO PRINT-LINE.
    WRITE PRINT-LINE.
    GO TO READ-NEXT-DIRECTORY-RECORD.
LIST-DONE.
    EXIT.
*
*
```

```
REV 14  COBOL      SOURCE FILE: SAMPLE       08/18/77   10:23
 (ØØØ1)          IDENTIFICATION DIVISION.
 (ØØØ2)          PROGRAM-ID. SAMPLE.
 (ØØØ3)          INSTALLATION. PRIME.
 (ØØØ4)          REMARKS. THIS PROGRAM CREATES AND READS A RELATIVE FILE
 (ØØØ5)                                SEQUENTIALLY.
 (ØØØ6)         *
 (ØØØ7)         *
 (ØØØ8)          ENVIRONMENT DIVISION.
 (ØØØ9)          CONFIGURATION SECTION.
 (ØØ1Ø)          SOURCE-COMPUTER. PRIME.
 (ØØ11)          OBJECT-COMPUTER. PRIME.
 (ØØ12)          SPECIAL-NAMES. CONSOLE IS TTY.
 (ØØ13)          INPUT-OUTPUT SECTION.
 (ØØ14)          FILE-CONTROL.
 (ØØ15)              SELECT LIST-FILE ASSIGN TO PRINTER.
 (ØØ16)              SELECT CARD-FILE ASSIGN TO PFMS.
 (ØØ17)              SELECT DIRECTORY-FILE ASSIGN TO PFMS, ORGANIZATION
 (ØØ18)                                    RELATIVE
 (ØØ19)                  ACCESS MODE IS SEQUENTIAL
 (ØØ2Ø)                  FILE STATUS IS FILE-STATUS.
 (ØØ21)          DATA DIVISION.
 (ØØ22)          FILE SECTION.
 (ØØ23)          FD  LIST-FILE, LABEL RECORDS ARE OMITTED.
 (ØØ24)              Ø1   PRINT-LINE, PICTURE X(121).
 (ØØ25)              Ø1    PRINT-REC.
 (ØØ26)                  Ø2   FILLER           PIC X.
 (ØØ27)                  Ø2   PRINT-INPUT      PIC X(8Ø).
 (ØØ28)                  Ø2   PRINT-ERROR      PIC X(4Ø)
 (ØØ29)          FD   CARD-FILE, LABEL RECORDS ARE STANDARD
 (ØØ3Ø)              VALUE OF FILE-ID IS 'INDATA'.
 (ØØ31)              Ø1   CARD-IMAGE,   PICTURE X(8Ø).
 (ØØ32)          FD   DIRECTORY-FILE, LABEL RECORDS ARE STANDARD, VALUE OF FILE-ID
 (ØØ33)                  IS 'REFILE'
 (ØØ34)                  OWNER IS 'LDAVIS'.
 (ØØ35)              Ø1   DIRECTORY-RECORD.
 (ØØ36)                   Ø2   NAME.
 (ØØ37)                        Ø3   LAST-NAME     PIC X(15).
 (ØØ38)                        Ø3   FIRST-NAME    PIC X(15).
 (ØØ39)                   Ø2   FILLER  PICTURE X(1).
 (ØØ4Ø)                   Ø2   ADDRESS  PICTURE X(25).
 (ØØ41)                   Ø2   FILLER   PICTURE X(1).
 (ØØ42)                   Ø2   CITY     PICTURE X(4).
 (ØØ43)                   Ø2   FILLER   PICTURE X(3).
 (ØØ44)                   Ø2   PHONE-NO  PICTURE  9(7).
 (ØØ45)                   Ø2   FILLER   PICTURE X(9).
 (ØØ46)          *
 (ØØ47)           WORKING-STORAGE SECTION.
 (ØØ48)               77   FILE-STATUS, PICTURE X(2),  VALUE IS SPACE.
 (ØØ49)               Ø1   HEADER.
 (ØØ5Ø)                    Ø2   H1, PICTURE X(4), VALUE IS 'NAME'.
 (ØØ51)                    Ø2   FILLER, PICTURE X(27), VALUE IS SPACE.
 (ØØ52)                    Ø2   H2, PICTURE X(6), VALUE IS 'STREET'.
 (ØØ53)                    Ø2   FILLER, PICTURE X(2Ø), VALUE IS SPACE.
```

```
REV 14 COBOL     SOURCE FILE: SAMPLE      08/18/77   10:23
 (ØØ54)                 Ø2   H3, PICTURE X(4), VALUE IS 'CITY'.
 (ØØ55)                 Ø2   FILLER, PICTURE X(3), VALUE IS SPACE.
 (ØØ56)                 Ø2   H4  PICTURE X(5), VALUE IS 'PHONE'.
 (ØØ57)       *
 (ØØ58)       *
 (ØØ59)        PROCEDURE DIVISION.
 (ØØ6Ø)        BEGIN SECTION.
 (ØØ61)        CREATE-FILE.
 (ØØ62)            OPEN OUTPUT LIST-FILE DIRECTORY-FILE.
 (ØØ63)            OPEN INPUT CARD-FILE.
 (ØØ64)                WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
 (ØØ65)        READ-NEXT.
 (ØØ66)            READ CARD-FILE AT END GO TO
 (ØØ67)                LIST-DIRECTORY.
 (ØØ68)            MOVE CARD-IMAGE TO PRINT-LINE.
 (ØØ69)            MOVE CARD-IMAGE TO DIRECTORY-RECORD.
 (ØØ7Ø)            WRITE PRINT-LINE.
 (ØØ71)            WRITE DIRECTORY-RECORD INVALID KEY
 (ØØ72)                DISPLAY 'INVALID KEY'.
 (ØØ73)            GO TO READ-NEXT.
 (ØØ74)        LIST-DIRECTORY.
 (ØØ75)            CLOSE CARD-FILE, DIRECTORY-FILE.
 (ØØ76)            DISPLAY 'END TEST TO CREATE FILE'.
 (ØØ77)            OPEN INPUT DIRECTORY-FILE.
 (ØØ78)            PERFORM LIST THRU LIST-DONE.
 (ØØ79)        LAST-SECTION.
 (ØØ8Ø)        CLOSE-ALL.
 (ØØ81)            CLOSE DIRECTORY-FILE, LIST-FILE.
 (ØØ82)            DISPLAY 'END TEST SEQUENTIAL READ AFTER A START'.
 (ØØ83)            STOP RUN.
 (ØØ84)        LIST.
 (ØØ85)            WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
 (ØØ86)        READ-NEXT-DIRECTORY-RECORD.
 (ØØ87)            READ DIRECTORY-FILE NEXT RECORD, AT END
 (ØØ88)                GO TO LIST-DONE.
 (ØØ89)            MOVE DIRECTORY-RECORD TO PRINT-LINE.
 (ØØ9Ø)            WRITE PRINT-LINE.
 (ØØ91)            GO TO READ-NEXT-DIRECTORY-RECORD.
 (ØØ92)        LIST-DONE.
 (ØØ93)            EXIT.
 (ØØ94)       *
 (ØØ95)       *

ØØØØ ERRORS   ØØØØ WARNINGS   (COBOL  VER Ø4)
```

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 own statements. They are presented in a uniform system of notation:

1. 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.

2. All underlined reserved words are required unless the portion of the format containing them is itself optional. These are key words. If any key word is missing or is incorrectly spelled, it is considered an error in the program. Reserved Words not underlined may be included or omitted at the option of the programmer. These words are optional words; they are used solely for improving readability of the program.

3. The characters <, >, and = when appearing in formats, although not underlined, are required when such formats are used.

4. 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.

5. Words printed in lower-case letters in formats represent generic parts (e.g., data-names) of which a valid representation must appear.

6. 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.

7. 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.

8. 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.

9. 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.

10. Comments, restrictions, and clarifications on the use and meaning of every format are contained in the appropriate portions of the manual.

11. 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:

1. A period, semicolon, or comma, when used, can not be preceded by a space, but must be followed by a space.

2. A left parenthesis can not be followed immediately by a space; a right parenthesis can not be preceded immediately by a space.

3. 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.

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

5. When the period, comma, plus, or minus characters are used in the PICTURE clause, they are governed solely by rules for report items.

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

7. 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 12-1). The following rules are applicable:

1. 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.

2. 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.

3. All other program elements must be confined to positions 12-72, governed by the other rules of statement punctuation.

4. Positions 73-80 are ignored by the compiler. Frequently, these positions are used to contain the program identification.

5. 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 for coding rules.

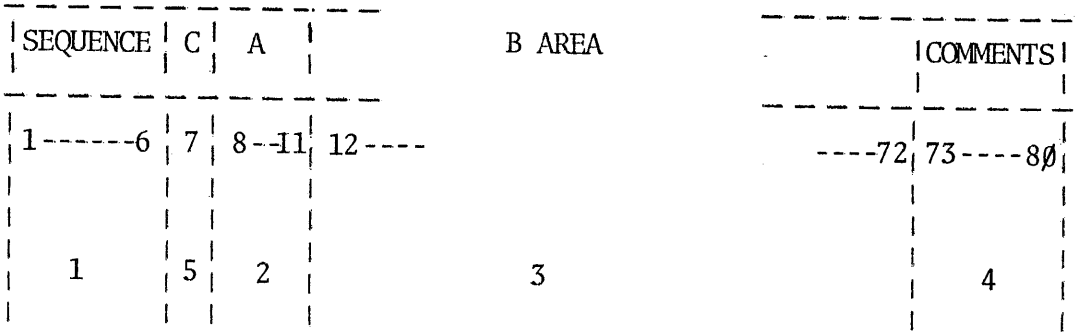


Figure 12-1. Standard COBOL Coding Sheets

Prime Character Set

The standard character set utilized 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 E.

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 America Standards Code for Information Interchange (ASCII). The characters in Appendix E, 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 12-2. An outline of Prime COBOL symbol usage is given in Appendix F.

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

Character Strings

A character-string is a character or a sequence 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 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 begin with a letter; it may not end with a hyphen. A word is ended by a space, or by proper punctuation. A word may contain more than one embedded hyphen; consecutive embedded hyphens are also permitted.

All words are either Reserved Words or programmer-defined words.

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

Paragraph-name and section-name are programmer-defined words which are not required to begin with an alphabetic character.

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

Figure 12-2. COBOL CHARACTERS

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.

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 programmer-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; i.e., 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 are 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 = the ASCII character represented by Octal 260
ZEROES

LOW-VALUE
LOW-VALUES = the character whose Octal representation is 000

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

NOTE: ALL literal is not currently available.

- 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

Table 12-1. Special-Character Words: Arithmetic Operators/Relation Characters

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
Condition-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 further subdivided, is an elementary item. Data items which contain subdivisions are known as group items.

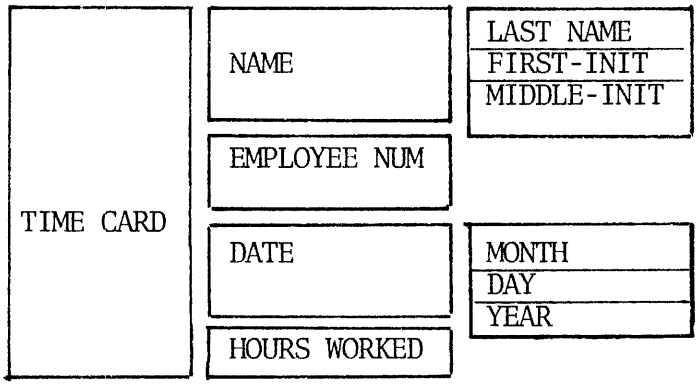
  Level numbers are one or two character, programmer-definded words. All level-numbers are numeric. They group items within the data hierarchy of the Record Description. Since records are the most inclusive data items, level-numbers for records begin at 01.

  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 30. 1 through 9 may be written as single numbers.

  Level numbers 77 and 88 are used in certain applications and are defined together with additional level-number information in Section 15, DATA DIVISION.

  A weekly timecard 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.



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

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

- Data-names

  In the preceding timecard 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 to 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:

  level data-name
         FILLER

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. 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 a unique programmer-supplied word called the file-name. Rules for composition of the file-name word are identical to those for data-names (see 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 ENVIRONMENT DIVISION under SPECIAL-NAMES for reference in ACCEPT or DISPLAY statements. A mnemonic-name is composed according to the rules for WORD FORMATION.

- Paragraph-names and Section-names

  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

Literals are not, strictly speaking, words; they are actual values.

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.

1. 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'
"DO'S & DON'T'S"
'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 rules apply to the next line of coding (continuation line):

A. A hyphen is placed in position 7 of the continuation line.

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

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

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

2. Numeric Literals

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

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

The following are examples of numeric literals:

```
72  +1011   3.14159   -6   -.333   0.5
```

By use of the Environment specification DECIMAL-POINT IS COMMA, the functions of 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 example following, the data-name, YEAR, will require qualification for procedural reference.

```
01   EMPLOYEE-RECORD
     02   NAME
     02   ADDRESS
     02   HIRE-DATE
          03   YEAR
          03   MONTH
          03   DAY
     02   TERMINATION-DATE
          03   YEAR
          03   MONTH
          03   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:

1. Each qualifier must be of a successively more inclusive level within the same hierarchy as the name it qualifies.

2. The same name must not appear at two levels in a hierarchy.

3. If a data-name or a condition-name is assigned to more than one data item in a source program, the data-name or condition-name must be qualified each time it is referred to in the Procedure Division (except in the REDEFINES clause where qualification must not be used).

4. A paragraph-name must not be duplicated within a section. When a paragraph-name is qualified by a section-name, the word SECTION must not appear. A paragraph-name need not be qualified when referred to from within the same section.

5. A data-name cannot be subscripted when it is being used as a qualifier.

6. A name can be qualified even though it does not need qualification. If more than one combination of qualifiers can make a name unique, only one 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.

7. A qualified name may only be written in the Procedure Division.

8. 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, alpha-numeric, and alphanumeric edited), as specified in the PICTURE clause, are grouped into three classes: Alphabetic, numeric, and alpha-numeric. For alphabetic and numeric data items, classes and categories are the same. The alphanumeric class includes the categories of alpha-numeric 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 ITEM	CLASS	CATEGORY
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric Edited Alphanumeric Edited Alphanumeric
Nonelementary (Group)	Alphanumeric	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

Figure 12-3. Classes of Data

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. A group cannot contain a PICTURE clause.

- 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.

The classes of data are: Alphabetic, numeric, alphanumeric. Within these, the categories of data are: Alphabetic, numeric, numeric edited, 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 or Report Item

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

- 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 in the Table 12-2.

- External Decimal Item

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

- Internal Decimal Item (Packed DECIMAL)

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

  A packed decimal item defined by n 9's in its PICTURE occupies $\frac{n}{2}+1$ 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 Ø 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.

- Index Item

  An index item has no picture; usage is INDEX. It is equivalent to COMPUTATIONAL.

USAGE IS	MACHINE DESCRIPTION
DISPLAY	EXTERNAL DECIMAL
COMPUTATIONAL	BINARY
INDEX	BINARY
COMPUTATIONAL-3	INTERNAL DECIMAL

Table 12-2. Data Representation and Usage

Standard Alignment Rules

1. If the receiving data item is described as numeric:

   A. 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.

   B. 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 Rule 1-A 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 positions 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: (␣=blank, (^)=implied decimal

DATA TO BE STORED	RECEIVING FIELD BEFORE TRANSFER	RECEIVING FIELD AFTER TRANSFER
ABC	PQRSTUVWXYZ	ABC␣␣␣␣␣␣␣␣
ABCDEF1234	PQRSTUVWXYZ	ABCDEF1234␣
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	34␣␣␣␣
'1234567890'	ABC234	123456
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. The compiler assumes a decimal point at the rightmost end of the field to be stored.

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.

Arithmetic Expressions

- Definition

  An arithmetic expression can be an identifier of 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. Permissible combinations of variables, numeric literals, arithmetic operator and parentheses are given in Table 12-3.

  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 Operators	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division

Unary Arithmetic Operators	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.

- Rules

1. 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 specified 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, providing intermediate result R3.

(4) The final result is computed by addition of A to R3.

Without parentheses, the expression

    A+B/C-D*E

is evaluated as:

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

final result = R2-R3

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

(1) A left parenthesis is preceded by one or more spaces.

(2) A right parenthesis is followed by one or more spaces.

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

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

2. Operators, variables, and parenthesis may be combined in arithmetic expressions as summarized below in Table 12-3.

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

Table 12-3. Symbol Combinations in Arithmetic Expressions

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

3. 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 eighteen (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 eighteen 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. There are two categories of conditions associated with conditional expressions: simple conditions and compound conditions.

- 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.

  1. Relation Condition

     A relation condition has the 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 (>).

Another form of relation which may be used involves the Reserved Word NOT, preceding any of the three relational symbols. Thus, the six relations in conditions are:

Relation	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 comprising 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.

The relational condition may take two forms; numeric comparisons and non-numeric comparisons.

A. 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 alignment of their decimal positions.

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

B. Non-Numeric Comparisons

For non-numeric Comparisons, non-equi-length comparisons are permitted, with spaces being assumed to extend the length of the shorter item, if necessary. Relationships are defined in the ASCII code; in particular, the letters A-Z are in an ascending sequence, and digits are less than letters. Refer to Appendix F for all ASCII character representations and the Prime collating sequence.

The data class (see Data Representation) of the two operands, where one is a literal, must be the same. For example, a numeric operand may not be compared to a non-numeric literal.

EXAMPLE:

```
Ø1 TEST-FIELD PIC 9
.
.
.
MOVE 1 TO TEST-FIELD
.
.
.
IF TEST-FIELD = '1'
.
.
.
```

The coding above will fail. The data class of the literal should be set up as numeric. Thus,

```
.
.
.
IF TEST-FIELD = 1
```

will execute properly.

2. Class Condition

The class condition determines whether the operand is numeric or alphabetic. If numeric, it consists entirely of the characters '0', '1', '2', ..., '9', with or without the operational sign. If alphabetic, it consists entirely of the characters 'A', 'B', 'C', ..., 'Z' and space. The general format for the class conditions is as follows:

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.

The class condition is equivalent to comparing the data contained in data-name to zero in order to determine the truth or falsity of the stated condition.

3. 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.

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.

4. 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 }

- Compound Conditions

A compound 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 parenthesis are permitted in compound conditions.

A compound condition has the format:

condition-1 $\frac{\text{AND}}{\text{OR}}$ [NOT] condition-2

The logical operators and their meanings 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.

The reserved words AND or OR permit the specification of a series of relational tests, as follows:

Individual relations connected by AND specify a compound condition which is met (true) only if all the individual relationships are met.

Individual relations connected by OR specify a compound condition which is met (true) if any of the individual relationships are met.

The compound 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:

X	Data Y	Value FLAG	SWITCH	EXECUTES PROCESSING?
1Ø	1Ø	'Z'	1	YES
1Ø	11	'Z'	1	NO
1Ø	11	'Z'	Ø	YES
1Ø	1Ø	'P'	1	NO
6	3	'P'	Ø	YES
6	6	'P'	1	NO

1. Evaluation

   A. Evaluation of individual simple conditions is done first.

   B. AND-connected simple conditions are next evaluated as a single result.

   C. OR and its adjacent conditions (or previously evaluated results) are then evaluated.

EXAMPLES:

(1) A < B OR C D OR E NOT > F

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

(2) WEEKLY AND HOURS NOT = Ø

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

(PAY-CODE = 'W') AND (HOURS = Ø)

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

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

OR P NOT EQUAL TO "SPAIN"

is evaluated as

[(A = 1) AND (B = 2) AND (G >-3)]

OR (P = "SPAIN")

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

```
A = 1
B = 2
G > -3
```

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

2. Other Considerations

A. Multiple Condition

Multiple Condition refers to compound conditions grouped in parenthesis. Where more than 5 levels of parenthesis are required, implicit grouping, condition-names, nested IF statements, or some combination should be substituted. For example, in the statement

```
IF A=B AND (C=D or E=F)
```

implicit grouping may be achieved by coding

```
IF A=B AND C=D OR A=B AND E=F.
```

B. Negating Conjunction

The use of NOT as a negating conjunction is not permitted in this compiler at this time. That is, IF A=B AND NOT C=D, is invalid. The reader should substitute suggested solutions for multiple conditions outlined above.

The use of NOT as a relation is permitted. Therefore, it is correct to code IF A=B AND C NOT=D, but incorrect to code IF A=B AND NOT C=D.

C. Implied Subjects (Abbreviated Combined Relation Conditions)

EXAMPLES:

```
IF A=B or C OR D (IMPLIED SUBJECT)

IF A=B OR A=C OR A=D (EXPLICIT SUBJECT)
```

Implied subjects or relations will be available at Revision 14.1. At this time, the statement IF A=B OR C, is invalid. It is suggested that the user employ condition-names, nested IF's, or full coding as alternatives.

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, DATA DIVISION).

The subscript can be represented either by a numeric literal which is an integer, or by a data-name. The data-name must be a numeric elementary item representing an integer. When the subscript is represented by a data-name, data-name 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 subscript is required, they are written in the order of successively less inclusive dimensions of the data organization.

The format is:

```
{data-name     }  (subscript-1 [, subscript-2  [, subscript-3]] )
{condition-name}
```

Indexing

References can be made to individual elements within a table of like elements by specifying indexing for that reference. An index is assigned 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, or a Format 4 PERFORM statement.

Direct and Relative Indexing

Direct and Relative indexing are supported by Prime COBOL as follows: 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 the operator + or -, followed by an unsigned 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 is executed which refers to an indexed table element, 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.

Restrictions on Qualification, Subscripting and Indexing Are:

- 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.

The general format for indexing is:

```
{data-name      } ( {index-name-1   [ {±} literal-2 ]}
{condition-name }   {literal-1                       }

[ , {index-name-2   [ {±} literal-4 ]} ] ...)
    {literal-3                       }

[ , {index-name-3   [ {±} literal-6 ]} ] ...)
    {literal-5                       }
```

NUCLEUS

REFERENCE

Sections 13, 14, 15, and 16, which follow, concern themselves with the four divisions of a COBOL program: The Identification Division, The Environment Division, The Data Division, The Procedure Division, respectively.

At the completion of each section, source coding for the corresponding division of a sample program, REF2, is presented as an example. At the close of Section 16, PROCEDURE DIVISION, the reader will find a print-out of the 64V mode Listing File for the entire REF2 program.

SECTION 13

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:

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 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, but the first 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 the order shown above.

GENERAL RULES:

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

2. 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.

EXAMPLE: REF 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REF2.
AUTHOR. PRIME COMPUTER.
INSTALLATION. FRAMINGHAM.
DATE-WRITTEN. SEPTEMBER, 1977.
DATE-COMPILED. SEPTEMBER, 1977.
REMARKS.  THIS AREA IS USED TO DESCRIBE THE PROGRAM.
```

SECTION 14

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 filename ASSIGN TO device
    [;RESERVE integer [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 filename-1, filename-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-dependant configurations and considerations do not apply, the entire ENVIRONMENT DIVISION may be omitted.

GENERAL RULES:

1. 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.

2. The sections and paragraphs in the Environment Division are discussed separately under their appropriate headings on the following pages.

[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 comments 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 comments 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.

A. [CONSOLE IS mnemonic-name]

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

EXAMPLE:

```
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.

B. [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.

C. [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 employment 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.

D. [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 F. 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 specific 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 1:

```
SELECT file-name

    ASSIGN TO device

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

     [; ORGANIZATION IS SEQUENTIAL]

     [; ACCESS MODE IS SEQUENTIAL]

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

FORMAT 2:

```
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 3:

```
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]
```

A. SELECT filename ASSIGN TO device ...

Filename 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 filename. The ASSIGN to devices clause associates the file with a storage medium or input/output hardware. Allowable devices appear in Table 14-1.

Device	Logical Device Maximum No. of Unit	Hardware Device
TERMINAL	-	CRT TERMINAL TTY TERMINAL
READER	-	CARD READER (for future designation)
PRINTER	-	SYSTEM PRINTER
PUNCH	-	CARD PUNCH (for future designation)
MT7	-	7 TRACK MAG. TAPE DRIVE (MT7 is currently interpreted as MT9
MT9		9 TRACK MAG. TAPE DRIVE
PFMS*	0-7	DISK STORAGE
OFFLINE-PRINT	0-7	FORMS PRINTER OUTPUT

*PFMS = PRIME FILE MANAGEMENT SYSTEM

Table 14-1. Device Specifications

EXAMPLES: SELECT filename ASSIGN TO TERMINAL.
SELECT filename ASSIGN TO PFMS.
SELECT FILENAME ASSIGN TO MT9.

B. [RESERVE integer [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.

C. [ORGANIZATION IS { SEQUENTIAL / RELATIVE / INDEXED }]

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

D. [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.

E. [FILE STATUS IS data-name].

The FILE STATUS clause permits the user to specify a two character, unsigned field (data-name) described in the Working Storage Section.

When the FILE STATUS clause is specified in the FILE-CONTROL paragraph, a value is moved by the operating system into data-name. This occurs 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 indicates to the COBOL program the status of execution of the statement.

The leftmost 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 key 2 further describes the results of the operation.

Status Key 1 settings:

'0' indicates Successful Completion.
'1' indicates At End
'3' indicates Permanent Error
'9' indicates Implementor Defined

NOTE: A setting of 9 indicates that the input-output statement was unsuccessfully executed as a result of a condition which is specified by the implementor. This value is used only to indicate a condition not otherwise specified by the values of status key 1, or by valid combinations of the values of status key 1 and status key 2. When status key 1 contains a value of '9', indicating an implementor-defined condition, the value of status key 2 is defined by Prime.

Valid combinations of key values for each type of file organization are shown in Appendix D, File Status Key Definitions.

[I-O-CONTROL.

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

SAME AREA FOR filename-2, filename-3... .]

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: REF 2

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. PRIME.
OBJECT-COMPUTER. PRIME.
SPECIAL-NAMES. CONSOLE IS TTY.
    ASCII IS NATIVE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT LIST-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-NO,
        ALTERNATE RECORD KEY LAST-NAME
        ALTERNATE RECORD KEY STATE
        ALTERNATE RECORD KEY BIRTHD
        ALTERNATE RECORD KEY FIRST-NAME
        FILE STATUS IS FILE-STATUS.
```

SECTION 15

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. That which is contained in files and enters or leaves the internal memory of the computer from a specified area or areas.

2. That which 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...]...]

[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. Sections and statements in the Data Division are discussed on the following pages in the same order in which they occur in the division. File and Record Description entries are presented in the File Section; the same Record Description entry format 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, and by one or more Record Description entries.

FORMAT:

```
[FILE SECTION.
    [[file-description-entry [record-description-entry]]...] ...
```

SYNTAX RULES:

1. The File Section is optional. If used, it must begin with the words FILE SECTION, followed by a period and a space.

2. The section consists of the header, followed by one or more File Description entries (FD). Each FD must be followed by Record Description entries for all records within the file described by the FD entry.

GENERAL RULE:

Each file associated with an input-output device must be represented by a File Description entry (see FILE-CONTROL.)

FILE DESCRIPTION

FUNCTION:

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

FORMAT:

FD file-name [UNCOMPRESSED]

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

[BLOCK CONTAINS integer-1 { CHARACTERS / RECORDS }]

[RECORD CONTAINS integer-2 [TO integer-3] CHARACTERS]

[VALUE OF FILE-ID IS literal-1]

[OWNER IS literal-2]
[LIFE-CYCLE IS integer-4]

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

[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.

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. Clauses in the File Description are presented on the following pages in the same order as they appear above.

8. The LIFE-CYCLE IS clause is not implemented at this revision. However, this clause is acceptable to the compiler.

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.

LABEL RECORDS

FUNCTION:

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

FORMAT:

$$\underline{\text{LABEL}} \left\{\begin{matrix}\underline{\text{RECORD}} \text{ IS} \\ \underline{\text{RECORDS}} \text{ ARE}\end{matrix}\right\} \left\{\begin{matrix}\underline{\text{STANDARD}} \\ \underline{\text{OMITTED}}\end{matrix}\right\}$$

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 15-1 below.

DEVICE	STANDARD	OMITTED
TERMINAL		X
READER		X
PRINTER		X
PUNCH		X
MT7 (TAPE)	X	
MT9 (TAPE)	X	
PFMS (DISK)	X	

Table 15-1. Label Options

BLOCK CONTAINS

FUNCTION:

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

FORMAT:

```
[BLOCK CONTAINS integer-1  { 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-1 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.

RECORD CONTAINS

FUNCTION:

The RECORD CONTAINS clause specifies the size of data records.

FORMAT:

[RECORD CONTAINS integer-2 [TO integer-3] 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-2 may not be used by itself unless all the data records in the file have the same size. In this case, integer-2 represents the exact number of characters in the data record. If integer-2 and integer-3 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.

VALUE OF FILE ID

FUNCTION:

The VALUE OF 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 is the name which is used by Prime at run-time to dynamically allocate files. It is a non-numeric value which may not exceed 8 characters.

2. If further definition does not occur at run-time, literal-1 will become the default value for internal filename designation.

OWNER IS

FUNCTION:

The OWNER IS clause points to the User File Directory (UFD) in a Prime system, in which literal-1 of VALUE OF FILE-ID 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 6 characters.

2. The clause is essentially ignored in this compiler in 64R mode; and it may be overridden in both 64R and 64V mode by explicit definition at run-time.

3. If the clause is omitted, a default of the current UFD will apply.

LIFE-CYCLE

FUNCTION:

LIFE-CYCLE allows for the development of an expiration date as "today plus integer" for output files.

FORMAT:

[LIFE-CYCLE IS integer-4]

SYNTAX RULE:

Integer-4 can contain a value of 0 to 32,767 inclusive.

GENERAL RULE:

If LIFE-CYCLE is omitted, integer-4 is assumed to be zero.

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 data records. They must be defined by 01 level-number Record Description entries and follow the general rules for WORD FORMATION.

GENERAL RULES:

1. If the file contains more than one type of data record, each type should be indicated by a data-name in this clause. These records may be different in format. 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, reflecting the fact that both internal and external data is represented in ASCII code.

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 1:

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

[OCCURS-Clause]

[{ PICTURE / PIC } IS picture-string]

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

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

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

[{ JUSTIFIED / JUST } RIGHT]

[BLANK WHEN ZERO]

[VALUE IS literal].

FORMAT 2:

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

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

FORMAT 3:

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

SYNTAX RULES:

1. The level-number in Format 1 may contain a value of 01 through 30, or 77.

2. In Format 1, 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 1, 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 an 01, 77, or 88 level-number.

5. Formats 2 and 3 are used only for condition-names which must have a level-number 88. Formats 2 and 3 may not be combined for a single, level 88 entry.

6. The words THRU and THROUGH are equivalent and interchangeable Reserved Words.

GENERAL RULES:

1. A detailed discussion of each clause in the Data Description entry appears under the appropriate clause heading on the following pages.

2. 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.

LEVEL-NUMBER

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 30, or 88.

3. Data Description entries in the Working-Storage and Linkage Sections must have level-numbers 01 through 30, 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 30, the least inclusive level. Any level-number except 30 can denote a group.

4. The level number 01 identifies the first entry in each Data Description. Reference to level-number 01 data-name in the Procedure Division causes the entire record to be accessible.

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:

A. Level-number 77 is assigned to identify noncontiguous working storage or linkage data items. They may be used only as described in Format 1 of the Data Description entry.

Level-number 77 data items are independent elementary items which cannot be subdivided.

B. Level-number 88 is assigned to entries which define condition-names associated with a conditional variable. They can be used only with Format 2 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:

```
01  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:

1. Another level 88 entry, where there are several consecutive condition-names pertaining to an elementary item;

2. 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 not pertain to an elementary item (a conditional variable) requiring subscripts. In this case, the condition-name, when written in the Procedure Division, cannot be subscripted according to the same requirements as the associated elementary item.

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

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

Using the above description, 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.

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 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 / FILLER } ;[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 77 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 must not contain a REDEFINES clause.

5. The Record 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 following example, redefinition of the data-name-2 area by data-name-1 ends when data-name-3 is encountered:

```
02 data-name-2 PICTURE A(3).

02 data-name-1; REDEFINES data-name-2.

     03 ITEM-A PICTURE A.

     03 ITEM-B PICTURE AA.

02 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.

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

Redefinition to a depth greater than one level is not permitted (see SYNTAX RULE 4, above). Thus, the nested REDEFINES outlined below is invalid:

```
02  A  PIC X(10).
02  B  REDEFINES A.
       03  C  PIC X(5).
       03  D REDEFINES C.
           04  E   PIC X(5).
       03  F  PIC X(5).
```

Identical results may be achieved with the following definition:

```
02  A  PIC  X(10).
02  B  REDEFINES A.
       03  C  PIC  X(5).
       03  F  PIC  X(5).
02  FILLER-1 REDEFINES A.
       03  D.
             04  E  PIC  X(5).
       03  FILLER  PIC  X(5).
```

Notice that the clauses B REDEFINES A, and FILLER-1 REDEFINES A are at the same level. Such definition is valid.

OCCURS

FUNCTION:

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 [INDEXED BY index-name-1 [, index-name-2] ...]

SYNTAX RULES:

1. The OCCURS clause must not be used in any Data Description entry having a level number 01, 77, or 88.

2. The maximum OCCURS specification (integer-1) is 1024.

3. When the OCCURS clause is used without the INDEXED BY option, the data-name which is the subject of the OCCURS clause is referred to by subscripting (see General Rule 4 below). If this data-name is the name of a group item, all data-names belonging to the group must be subscripted whenever used.

4. An INDEXED BY phrase is required if the subject of this entry, or an entry subordinate to this entry, is to be referred to by indexing. Neither index-name-1 nor index-name-2 are defined elsewhere, since their allocation and format are dependent on the system; not representing data, the index-names cannot be associated with any data hierarchy (see General Rule 5 below).

GENERAL RULES:

1. The OCCURS clause defines tables and other homogenous sets of repeated data items. Whenever the clause is used, the data-name that is its subject must be either subscripted or indexed whenever it is referenced.

2. Except for the OCCURS clause, all data description clauses associated with an item whose description includes an OCCURS clause apply to each occurrence of the item described.

3. Integer-1 represents the exact number of occurrences of the subject entry.

4. When the INDEXED BY option is omitted, subscripting is used to indicate an individual item within a list, or within a table of like items which do not have individual data-names.

The format for a subscript 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. The data-name may be qualified but not subscripted. The subscript must be delimited by a pair of parentheses 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.

The value of the subscript indicates the position of the item in a table. The lowest possible value of a subscript is 1, indicating the first position in the table. Subsequent positions are indicated by sequential values 2, 3, 4 ..., up to the highest permissible value, which is the maximum number of occurrences of the item specified in the OCCURS clause.

A data-name may not be subscripted if it is being used for any of the following functions:

A. When it is being used as a subscript.

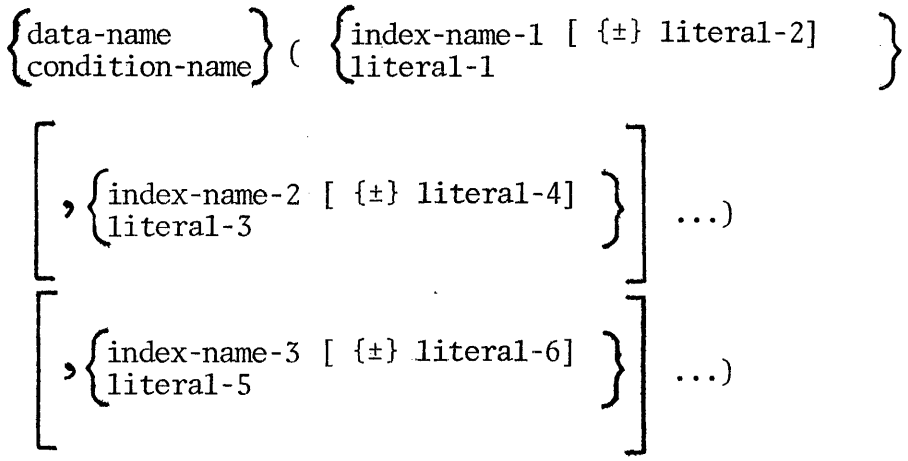
B. When it appears as the defining name of a Data Description entry.

C. When it appears as data-name-2 in a REDEFINES clause.

A subscript value is changed via the MOVE, ADD, or SUBTRACT verbs. The SET verb cannot be used on a subscript data-name. (See TABLE HANDLING.)

5. When INDEXED BY is used, an index is assigned to a table of like elements, with individual items in the table being identified by index-name.

The general format for indexing is:



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. Index-name is equivalent to an index-item. The compiler assigns a full word for each index-name defined.

An index-name must be uniquely named. An index item may only be referred to by a SET statement, a CALL statement's USING list, a Procedure header USING list, as the variation item in PERFORM VARYING and PERFORM UNITL, or in a relational condition. In all cases, the process is equivalent to dealing with a binary word integer subscript. (See TABLE-HANDLING.)

Direct indexing is specified by using an index-name in the form of a subscript. Relative indexing is specified by a parenthetic statement following data-name, in which index-name is followed by the operator + or - and an unsigned integer numeric literal.

When a statement referring to an indexed table element is being executed, the value in the index referred to by the index-name must be from 1 to the highest permissible occurrence number specified in the OCCURS clause. This restriction applies also to the value resulting from relative indexing. See TABLE HANDLING for more detailed discussion.

PICTURE

FUNCTION:

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

FORMAT:

$$\left[\left\{ \begin{matrix} \text{PICTURE} \\ \text{PIC} \end{matrix} \right\} \text{ IS picture-string } \right]$$

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 the picture-string is 30. As an example, PICTURE X(89) consists of five PICTURE characters.

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 be described with a PICTURE clause: Alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric edited.

A. 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.

B. 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.

C. 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.

D. Alphanumeric edited:

- Picture-string is restricted to certain combinations of the following symbols: A, X, 9, B, 0, /; and

- Item contents are any character from the computer's ASCII character set.

E. Numeric Edited:

- The picture-string is a certain combination of the editing symbols: Z . CR DB , $ + * B 0 = - / 9 V P; and

- The picture-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 Z * 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 (see also Appendix F, SYMBOLS).

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 rightmost PICTURE characters), the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.

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 99000 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.

Ø - Each Ø in the character-string represents a character position into which the numeral zero will be inserted. The 'Ø' 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 wherever 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.

A. 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:

CATEGORY OF DATA	TYPE OF EDITING
Alphabetic	Simple insertion 'B' only
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple insertion Ø, B and /
Numeric Edited	All, subject to rules in Rule 3 below

Table 15-2. Categories of Data and Editing

B. Insertion Editing includes the following types:

Simple insertion
Special insertion
Fixed insertion
Floating insertion

1) Simple insertion editing utilizes B Ø , / 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.

2) 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.

3) 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 the item, except that it can be preceded by either a + or a - 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:

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

Table 15-3. Results of Sign Control Symbols in Editing

4) 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, ß represents a blank in the developed items.

EXAMPLES:

Picture-string	Numeric Value	Developed Item
$$$999	14	ßß$Ø14
--,---,999	-456	ßßßßß-456
$$$$$$	14	ßß$14

A floating string need not constitute the entire PICTURE of a report 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 positions in the PICTURE must be represented by the same character.

The PICTURE character 9 can never appear to the left of a floating string, or replacement character. 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.

C. 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 suppression symbols in a picture-string. These symbols are mutually exclusive in a given 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;

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 *.

D. 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.ØØ
9(5)	ØØ123	$$$,$$9.99	$123.ØØ
9(5)	ØØØØØ	$$$,$$9.99	$Ø.ØØ
9(4)V9	12345	$$$,$$9.99	$1,234.5Ø
V9(5)	12345	$$$,$$9.99	$Ø.12
S9(5)	ØØ123	-------.99	123.ØØ
S9(5)	-ØØØØ1	-------.99	-1.ØØ
S9(5)	ØØ123	+++++++.99	+123.ØØ
S9(5)	ØØØØ1	-------.99	1.ØØ
9(5)	ØØ123	+++++++.99	+123.ØØ
9(5)	ØØ123	-------.99	123.ØØ
S9(5)	12345	*******.99CR	**12345.ØØ
S999V99	Ø2345	ZZZVZZ	2345
S999V99	ØØØØ4	ZZZVZZ	Ø4
S9(5)	-12345	*******.99CR	**12345.ØØCR

Figure 15-1. 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 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:

$$[\underline{\text{SIGN}} \text{ IS } \left\{\begin{matrix}\underline{\text{LEADING}} \\ \underline{\text{TRAILING}}\end{matrix}\right\} [\underline{\text{SEPARATE}} \text{ CHARACTER}]]$$

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:

   A. The operational sign is presumed associated with the leading (or, respectively, trailing) digit position of the elementary numeric data item.

   B. The character S in picture-string is not counted in determining item size.

3. If the SEPARATE CHARACTER phrase is present, then:

   A. 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.

   B. The letter S in a picture-string is counted in determining the size of the item (in terms of standard data format characters).

   C. 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 15-4 depicts sign representations for the various SIGN clause options.

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

Table 15-4. Sign Representation

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 than 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:

```
[ BLANK WHEN ZERO ]
```

SYNTAX RULE:

The BLANK WHEN ZERO clause can be used only for an elementary numeric or numeric edited (report) 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 suppressed by this clause.

2. If the clause is specified for a numeric item, the category of the item is interpreted as numeric edited.

3. The BLANK WHEN ZERO clause may be used in conjunction with editing characters. In such instances, editing occurs according to PICTURE specifications if data item values are not zero. For example, if a data item value is 0000.04, and the editing PICTURE is ****.99 BLANK WHEN ZERO, the result will be ****.04. Since leading zeros are not affected by the BLANK WHEN ZERO clause, the asterisk editing characters take precedence, and leading zeros are replaced by the character *.

EXAMPLES: (Ø=blank)

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
0000.00	****.99	****.00
0000.00	****.99 BLANK WHEN ZERO	ØØØØØØØ
0012.34	****.99 BLANK WHEN ZERO	**12.34
0000.04	$$$$$.99 BLANK WHEN ZERO	$.04
0000.00	$$$$$.99 BLANK WHEN ZERO	ØØØØØØØ
0000.00	ZZZZVZZ BLANK WHEN ZERO	ØØØØØØØ
0000.04	ZZZZVZZ BLANK WHEN ZERO	4
0000.00	ZZZZ.ZZ BLANK WHEN ZERO	ØØØØØØØ
0000.04	ZZZZ.ZZ BLANK WHEN ZERO	.04

Figure 15-2. 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 1:

```
[ VALUE IS literal ]
```

FORMAT 2:

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

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, or in any entry subordinate to one specifying an OCCURS or REDEFINES clause.

3. Numeric literals in a VALUE clause must have a value which is within the 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 128 characters; e.g., PIC X(129) VALUE SPACES is invalid.

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 a 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 1 and Format 2 instead of a literal.

5. Format 1 is required to define an initial value for a data item or a constant.

6. Format 2 is required 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 THRU 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:

   A. In the File and Linkage Sections, the clause can be used only in condition-name entries.

   B. 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. 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.

   B. 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.

C. A condition-name is used in the Procedure Division in place of a simple relational condition.

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

E. 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.)

EXAMPLE:

```
02  PAYROLL-PERIOD     PICTURE IS 9.

    88 WEEKLY          VALUE IS 1.

    88 SEMI-MONTHLY    VALUE IS 2.

    88 MONTHLY         VALUE IS 3.
```

Using the above description, 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.
```

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

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

SYNTAX RULES:

1. The Working-Storage Section is optional. If included, it must begin with the words WORKING-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 qualification.

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:

   A. Noncontiguous elementary items with the level-number of 77. These items and constants have no hierarchical relationship to one another and cannot be grouped into records because they cannot be further subdivided.

B. Data items in records not associated with an input-output device and not part of external data files, but developed and processed internally. These items employ level numbers 01 through 30.

2. VALUE clauses, prohibited in the FILE SECTION, are permitted throughout Working-Storage to specify the initial value of an item, except for an index data item.

LINKAGE SECTION

FUNCTION:

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

FORMAT:

```
[ LINKAGE SECTION.
  [level 77 data description entry     ] ...]
  [data item description entry         ]
```

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 clause or USAGE IS INDEX clause. Other optional data description clauses may be included as necessary.

4. Data items in the Linkage Section, which have a definate hierarchic relationship to one another, must be grouped into records according to the rules for formation of Record Descriptions.

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 object 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 object program is under the control of a CALL statement which specifies a USING phrase.

4. A Linkage Section example is presented in Section 17, INTER-PROGRAM COMMUNICATION.

EXAMPLE: COBOL Statement REF 2

```
       *
       *
       DATA DIVISION.
       FILE SECTION.
       FD  LIST-FILE, LABEL RECORDS ARE OMITTED.
           01 PRINT-LINE, PICTURE X(100).
           01 PRINT-LINE1.
              02 FILLER PIC X.
              02 PRINT-LIN PIC X(99).
       FD  CARD-FILE, LABEL RECORDS ARE STANDARD
           VALUE OF FILE-ID IS 'INDAT1'.
           01 CARD-IMAGE, PICTURE X(80).
           01 CARD-D1.
              02 DATA-D1      PIC X(64).
              02 PHONE-D1     PIC X(8).
              02 D2           PIC X(8).
       FD  DIRECTORY-FILE, LABEL RECORD ARE STADARD,
           VALUE OF FILE-ID IS 'INDXFILE'
           OWNER IS 'LDAVIS'.
           01 DIRECTORY-RECORD.
              02 PHONE-NO     PIC X(8).
              02 NAME.
                 03 LAST-NAME PIC X(14).
                 03 FILLER    PIC X.
                 03 FIRST-NAME PIC X(13).
                 03 FILLER    PIC XX.
              02 FILLER, PICTURE X.
              02 ADDRESS, PICTURE X(25).
              02 FILLER, PICTURE X.
              02 CITY, PICTURE X(4).
              02 FILLER, PICTURE X(3).
              02 STATE, PICTURE XX.
              02 BIRTHD, PICTURE 9(6).
              02 FILLER, PICTURE X(20).
           01 DIR-1.
              02 DISPLAY-DIR  PIC X(72).
              02 FILLER       PIC X(28).
           01 SOME-D1.
              02 D1           PIC X(8).
              02 D3           PIC X(64).
              02 D4           PIC X(8).
              02 FILLER       PIC X(20).
       WORKING-STORAGE SECTION.
           77 GO-TO-READ       PICTURE 9 VALUE 0.
           77 CREATE-UPDATE    PICTURE X VALUE SPACE.
           77 GO-TO-NAME       PICTURE 9 VALUE 0.
           77 FILE-STATUS      PICTURE X(2) VALUE IS SPACE.
           77 CHAR-1           PICTURE X VALUE SPACE.
```

REF 2

```
       01  PERFORM-COUNT1.
           02  PERFORM-COUNT     PIC 999.
           02  PER-CO  REDEFINES PERFORM-COUNT
                   PICTURE X, OCCURS 3 TIMES.
       01  WS-RECORD.
           02  WS-LAST-NAME              PIC X(14).
           02  FILLER                    PIC X.
           02  WS-FIRST-NAME             PIC X(13).
           02  FILLER                    PIC XXX.
           02  WS-ADDRESS                PIC X(25).
           02  FILLER                    PIC X.
           02  WS-CITY                   PIC X(4).
           02  FILLER                    PIC XXX.
           02  WS-PHONE-NO               PIC X(8).
           02  WS-STATE                  PIC XX.
           02  WS-BIRTHD                 PIC X(6).
       01  HEADER.
           02  FILLER          PICTURE X       VALUE SPACE.
           02  H0              PIC X(8)        VALUE 'PHONE'.
           02  H1              PICTURE X(4)  VALUE IS 'NAME'.
           02  FILLER          PICTURE X(27) VALUE IS SPACE.
           02  H2              PICTURE X(6)  VALUE IS 'STREET'.
           02  FILLER          PICTURE X(20) VALUE IS SPACE.
           02  H3              PICTURE X(4)  VALUE IS 'CITY'.
           02  FILLER          PICTURE X(3)  VALUE IS SPACE.
```

SECTION 16

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:

   A. The program being written is a CALLable subprogram which is to function under the control of a CALL statement.

   B. 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 18, 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 entity 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:

A. 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 INVALID KEY clause.

B. 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.

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:

A. 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 permissable in these arithmetic statements.

B. Decimal point alignment is supplied automatically throughout the computations.

C. Intermediate result fields generated for the evaluation of arithmetic expressions assure the accuracy of the result field, except where high-order truncation is necessary.

D. The maximum size of each operand is eighteen (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 eighteen decimal digits.

E. When arithmetic is attempted with one or more non-numeric operands in VMODE, the program will execute, but results are invalid. In RMODE, the program will terminate with an error message "NON-NUMERIC DATA".

NOTE: With UII (Unimplemented Instruction Package) on Prime 400 and Prime 500 units, SPACES is interpreted as zeros when utilized in arithmetic statements.

6. The three statement components which may appear in all arithmetic statements are: The GIVING option, the ROUNDED option, the SIZE ERROR option.

A. 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 report item.

B. 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

Figure 16-1. Rounding Results

C. 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 occurs (in this case, it is apparent that RECORD-COUNT HAS Picture 99, and cannot hold a value of 100), both the MOVE and DISPLAY statements are executed. Otherwise, the MOVE and DISPLAY statements are not executed.

PROCEDURE STATEMENTS

COBOL statements (verbs) are described on the following pages alphabetically as presented in the index below. This index is designed as a quick reference to assist the user in locating format descriptions and in determining verb category and special applications.

PRIME COBOL VERBS

[a] IF is a verb in COBOL, although not a verb in the grammatical sense in English.

Table 16-1. Prime COBOL Verb Index

ACCEPT

FUNCTION:

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

FORMAT 1:

ACCEPT data-name [FROM mnemonic-name]

FORMAT 2:

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

SYNTAX RULE:

The mnemonic-name in Format 1 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:

A. Execution is suspended.

B. When the operator enters a response, the program stores the acquired data in the field designated by data-name, and normal execution proceeds.

C. The data size is controlled by the size specified for data-name.

D. 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 2 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 elasped 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.

A D D

FUNCTION:

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

FORMAT 1:

```
ADD {data-name-1} [,data-name-2] . . . TO data-name-n [ROUNDED]
    {literal-1  } [,literal-2  ]

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

FORMAT 2:

```
ADD {data-name-1} {,data-name-2} [,data-name-3] . . .
    {literal-1  } {,literal-2  } [,literal-3  ]

    GIVING data-name-m [ROUNDED] [; ON SIZE ERROR imperative-statement]
```

SYNTAX RULES:

1. In Formats 1 and 2, each data-name must refer to an elementary numeric item, except that in Format 2 each item following GIVING can be either an elementary numeric item or 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 operands, excluding those following the word GIVING, were to be superimposed upon each other, aligned by their implied decimal points, their composite can not exceed 18 decimal digits in length.

GENERAL RULES:

1. In Format 1, the values of the operands preceding the word TO are added, the sum is added to the current value of data-name-m and the result is stored immediately in data-name-m.

2. In Format 2, the values of the operands preceding the word GIVING are added, and the sum is stored as the new value of data-name-m following GIVING.

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 the results could occur.

4. 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 MF-OPEN.
M-F-OPEN.
    OPEN INPUT MASTER-FILE.
    ALTER GATE TO PROCEDE 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 this COBOL. Its use, however, is inconsistent with structured programming techniques. The reader should be aware that the ALTER statement presents difficulties in the debugging 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 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.

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 01 or 77. Data-name-1, data-name-2, ..., 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.

5. See Section 17, Interprogram Communication for additional information and examples.

CLOSE

FUNCTION:

The CLOSE statement terminates the processing of files, reels/units, with optional rewind and/or lock or removal where applicable.

FORMAT 1:

```
CLOSE file-name-1 [ { REEL } [with NO REWIND ] ... ]
                  { UNIT } [FOR REMOVAL      ]
```

FORMAT 2:

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

FORMAT 3:

```
CLOSE index-file-name
```

SYNTAX RULES:

1. The REEL or UNIT phrase must be used only for sequential files.

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

3. Except where specifically stated, the terms UNIT and REEL are synonomous and interchangeable.

GENERAL RULES:

1. Format 3 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. Files are divided into the following categories to show the effect of various types of CLOSE statements as applied to various storage media:

   A. Nonreel/unit - A file on an input or output medium (a printer, a disk).

   B. Sequential single-reel/unit - A file wholly contained on one reel/unit.

C. Sequential multireel/unit - A sequential file which is contained on more than one reel/unit.

D. Nonsequential single/multireel/unit - A relative or indexed file, residing on a disk device, which may be a single or multiunit file.

4. For this compiler, CLOSE statement options are treated as comments.

COMPUTE

FUNCTION:

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

FORMAT:

$$\underline{\text{COMPUTE}}\ \text{data-name-1}\ [\underline{\text{ROUNDED}}]\ =\ \left\{\begin{matrix}\text{data-name-2}\\ \text{numeric-literal}\\ \text{arithmetic-expression}\end{matrix}\right\}\ [\underline{\text{SIZE}}\text{-}\underline{\text{ERROR}}\text{-clause}]$$

SYNTAX RULE:

In general, data-names 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 and LANGUAGE SPECIFICATIONS.

COPY

FUNCTION:

The COPY statement provides a means of including pre-written COBOL source coding in the programs at compile time.

FORMAT:

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

SYNTAX RULES:

1. OF and IN are interchangeable and mutually 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:

A. FILE-CONTROL. COPY text-name.

B. FD MASTER-FILE COPY text-name OF SUB.

C. Ø1 MASTER-RECORD. COPY text-name IN SUB.

D. SECTION-NAME SECTION. COPY text-name.

E. PARAGRAPH-NAME. COPY text-name IN SUB.

Of the examples above, A and D have copy members contained on the same UFD as the source program. B, C, and D 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 Ø1 MASTER-DESCRIPTION entry; it might have the format:

```
    Ø2  BADGE-NO  PIC 9(5).
    Ø2  NAME.
        Ø3  LAST-NAME  PIC X(15).
        Ø3  FIRST-NAME  PIC X(15).
```

After compilation, examination of the listing file would reveal:

```
Ø1  MASTER-DESCRIPTION.  (COPY MASDES.) (where the copy member is
    Ø2  BADGE-NO  PIC 9(5).                          comment only.)
    Ø2  NAME.
        Ø3  LAST-NAME  PIC X(15).
        Ø3  FIRST-NAME  PIC X(15).
```

DELETE

FUNCTION:

The DELETE statement logically removes a record from a disk file.

FORMAT:

DELETE file-name [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.

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 19 and 20.

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. 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	522Ø7
data-name literal	DISPLAY 'BADGE-NO = 'BADGE-NO	BADGE-N = 522Ø7
literal	DISPLAY 'END-JOB'	ENDJOB
literal figurative-constant	DISPLAY 'SELECT' ZERO	SELECTØ

DIVIDE

FUNCTION:

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

FORMAT 1:

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

FORMAT 2:

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

FORMAT 3:

```
DIVIDE {data-name-1} INTO {data-name-2} GIVING data-name-3 [ROUNDED]
       {literal-1   }     {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 1, data-name-1 or literal-1 is divided into data-name-2; the quotient then replaces the divident (data-name-2).

2. In FORMAT 2, data-name-1 or literal-1 is divided by data-name-2 or literal-2. The quotient replaces the first operand, data-name-1.

3. In FORMAT 3, division occurs as in A or B below, and the quotient is stored in the data items following the word GIVING.

   A. 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.

   B. 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.

4. The REMAINDER clause of 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 MINUS
(data-name-2 TIMES data-name-3).
```

ENTER

FUNCTION:

The ENTER statement is classified as a compiler-directing statement; it acts as a modifier 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. 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 Assembler Language subroutine must be preceded by its own ENTER ASSEMBLER statement.

GENERAL RULES:

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

2. The ENTER statement is optional in this compiler.

EXHIBIT

FUNCTION:

The EXHIBIT statement provides a means for receiving 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 check-pointing information. Specified data is EXHIBITED on the terminal.

2. The EXHIBIT statement differs from DISPLAY in that data-name is printed as well as its value and an = character.

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. The EXIT sentence must be the only sentence in the paragraph.

GENERAL RULES:

1. An EXIT statement serves only to enable the user 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. The EXIT PROGRAM sentence must be the only sentence in the paragraph.

GENERAL RULES:

1. An 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.

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 1:

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

FORMAT 2:

```
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 1 GO TO statement.

2. In Format 2, 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. When a Format 1 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.

3. When a GO TO statement represented by Format 2 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.

4. In a Format 2 GO TO statement, there is no limitation to the total number of characters permitted in procedure-names. The aggregate number of acceptable characters is unlimited.

I F

FUNCTION:

The IF statement causes the evaluation of a condition, permitting the execution of specified procedural statements if the condition is true.

FORMAT:

IF condition $\left\{\begin{matrix}\underline{\text{NEXT}}\ \underline{\text{SENTENCE}} \\ \text{statement(s)-1}\end{matrix}\right\}$ [$\underline{\text{ELSE}}$ $\left\{\begin{matrix}\text{statement(s)-2} \\ \underline{\text{NEXT}}\ \underline{\text{SENTENCE}}\end{matrix}\right\}$]

SYNTAX RULE:

The conditions in the IF statement must conform to the rules and outlining of conditions specified in Conditional Expressions, Section 12.

GENERAL RULES:

1. If the condition is true, any ELSE phrase is bypassed and either statement-1 or the NEXT SENTENCE (whichever was specified in the statement) is executed, as follows:

   A. 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.

   B. 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:

   A. 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.

   B. 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:

IF operand relation operand

The six relations in conditions are:

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.

EXAMPLES:

```
IF BALANCE = Ø GO TO NOT-FOUND.

IF X = 1.743 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 { 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 16-2.

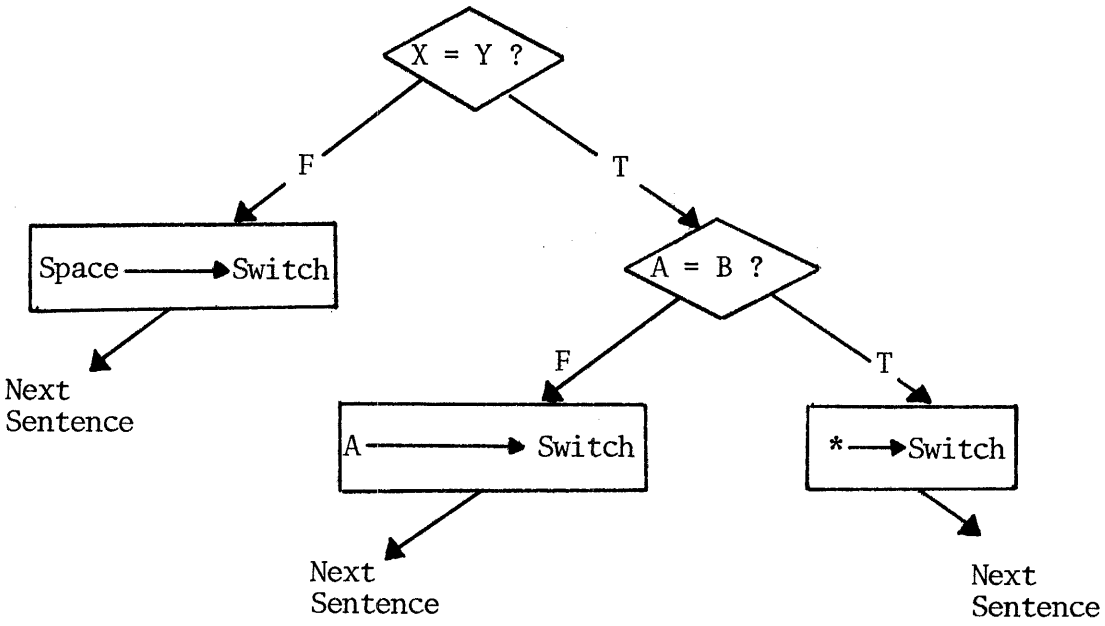


Figure 16-2. Nested IF Tree Structure

Another useful way of viewing nested IF structures 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 "A" 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.

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 } | CHARACTERS } operand-2

[ { BEFORE | AFTER } INITIAL operand-3]

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

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

SYNTAX RULE:

Data-name operands must be described (implicitly or explicitly) as USAGE IS DISPLAY.

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 containing only a REPLACING clause.

2. The INSPECT statement enables examination of a character-string item, permitting various combinations of the following actions:

   A. Counting appearances of a specified character;

   B. Mapping a specified character into an alternative.

   C. 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. When an AFTER INITIAL operand-3 subclause 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. Data-name-2 is not initialized prior to the operation.

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.

MOVE

FUNCTION:

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

FORMAT:

$$\underline{\text{MOVE}} \left\{\begin{matrix}\text{data-name-1} \\ \text{literal}\end{matrix}\right\} \underline{\text{TO}} \text{ data-name-2 [data-name-n...].}$$

SYNTAX RULE:

Data-name-1 and literal represent the sending area; data-name-2, data-name-n 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 report:

   A. The items are aligned by decimal points, with generation of zeros or truncation on either end, as required.

   B. When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place.

   C. The items may have special editing performed on them with suppression of zeros, insertion of a dollar sign, etc., and decimal point alignment, as specified by the receiving area.

4. For non-numeric source and targets:

   A. The characters are placed in the receiving area from left to right (unless JUSTIFIED RIGHT applies).

   B. If the receiving field is not completely filled by data being moved, the remaining positions are filled with spaces.

C. If the source field is longer than the receiving field, the move is terminated as soon as the receiving field is filled.

5. When overlapping fields are involved, results are not predictable.

6. Table 16-2 summarizes the various types of moves permitted with the MOVE statement.

SENDING \ RECEIVING	ALPHABETIC	BINARY	ALPHANUMERIC EDITED	NUMERIC	NUMERIC EDITED	ALPHANUMERIC
ALPHABETIC	X		X			X
BINARY		X		X	X	X (A)
ALPHANUMERIC EDITED	X		X (C)			X
NUMERIC		X		X	X	X (B)
NUMERIC EDITED			X (C)			X (C)
ALPHANUMERIC	X			X	X (D)	X

NOTES:

(A) If receiving operand length L is less than or equal to 18, target Picture 9(L) is assumed. Otherwise, the MOVE is disallowed.

(B) 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.

(C) The source is considered as a character string.

(D) If source length L is less than or equal to 18, source Picture 9(L) is assumed. Otherwise, the MOVE is disallowed.

Table 16-2. Permissible Moves

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.

EXAMPLE:

If the field BALANCE is to be multiplied by 1.03, it must be written as:

```
MULTIPLY 1.03 BY BALANCE
```

Where the result will be stored in the data item named BALANCE.

2. When the GIVING option is taken, the product is stored in data-name-3.

3. The rules for signs are those presented in FUNDAMENTAL CONCEPTS OF COBOL, Algebraic Signs.

OPEN

The OPEN statement initiates the processing of files, and enables other input/output operations, such as label checking and writing.

FORMAT 1:

```
OPEN { INPUT  }
     { I-O    }  filename ...
     { OUTPUT }
```

FORMAT 2:

```
OPEN { { INPUT  }                          }
     { { I-O    }  index-file-name-1 ... }....
     { { OUTPUT }                          }
```

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.

GENERAL RULES:

1. Format 1 is used for Sequential I-O (SAM files).

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 statements.

6. 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.

7. See Sections 19 and 20 for additional information on Indexed I-O and Relative I-O, respectively.

8. OPEN statements vs. Access Mode for Indexed and Relative files are presented in Table 16-3 below.

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

Table 16-3. OPEN Statements and Access Modes

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 1:

```
PERFORM procedure-name-1 [ {THROUGH} procedure-name-2 ]
                           {THRU   }

[ {integer    } TIMES]
  {data-name-1}
```

FORMAT 2:

```
PERFORM procedure-name-1 [ {THROUGH} procedure-name-2 ]
                           {THRU   }

[ VARYING {data-name-2} FROM {data-name-3} BY
          {index-name-1}     {index-name-2}
                             {literal-1   }

{data-name-4}] [ UNTIL condition-1 ]
{literal-2  }
```

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. In Format 2, if an index-name is specified in the VARYING or AFTER phrase, then:

   A. Data-name in the associated FROM and BY phrases must be an integer data item.

B. The literal in the associated FROM phrase must be a positive integer.

C. The literal in the associated BY phrase must be a non-zero integer.

5. In Format 2, if an index-name is specified in the FROM phrase, then:

A. Data-name in the associated VARYING or AFTER phrase must be an integer data item.

B. The data-name in the associated BY phrase must be an integer data item.

C. The literal in the associated BY phrase must be an integer.

6. In Format 2, literal in the BY phrase must not be zero.

7. In Format 2, condition-1... condition-n may be any conditional expression as described in FUNDAMENTAL CONCEPTS OF COBOL, Conditional Expressions.

GENERAL RULES:

1. If procedure-name-n is a paragraph-name, control is returned to the next sequential instruction after the last sentence of that paragraph.

2. If procedure-name-n is a section-name, control is returned to the next sequential instruction after the last sentence of the last paragraph of that section.

3. If the PERFORM statement is written with no options, control is transferred to procedure-name-1. At the completion of procedure-name-1, control is implicitly returned to the next executable statement following the PERFORM statement.

4. If the THROUGH option in Format 1 is taken, multiple paragraphs or sections can be executed before control is returned to the next sequential statement.

5. In Format 1, 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.

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 is not executed; control passes to the statement following PERFORM.

6. If the UNTIL option in Format 2 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 or OTHERWISE 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 statement, PERFORM is not executed and control passes to the next sequential instruction.

7. Format 2 with all options is used to vary the values referred to by data-name-2 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-2 is set to the current value of data-name-3 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-2 is incremented or decremented by the value in data-name-4 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.

8. At the termination of a Format 2 PERFORM statement, data-name-2 or index-name-1 have a value which exceeds the last used setting by the value of data-name-4 or literal-2. If the condition was true before initial execution of PERFORM, data-name-2 or index-name-1 contain the current value of data-name-3 or index-name-2.

READ

FUNCTION:

The READ statement makes available a record from a file.

FORMAT 1:

READ file-name [NEXT] RECORD [INTO data-name-1]

[AT END imperative statement].

FORMAT 2:

READ file-name [INTO data-name-1] [KEY IS data-name-2]

[INVALID KEY imperative-statement].

SYNTAX RULES:

1. Format 1 is used for all sequentially read files.

2. The NEXT phrase option in Format 1 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 2 is used only for Indexed I-O and Relative I-O files.

4. The KEY IS option of Format 2 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 1, 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 Declarative procedure is performed. If neither AT END nor Declarative exists, an execution I-O error occurs.

4. Format 1, 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 1, 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 2 statement. Thereafter, a Format 1 statement may be used.

11. NOTE: 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 19 and 20, 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 RULES:

1. Each time a paragraph or section in the Procedure Division is entered, that paragraph or section name is output to the terminal.

2. In 64R mode the format printed is:

Program name/subprogram name section-name/paragraph-name

3. In 64V mode, the format printed is:

ENTER: section-name/paragraph-name.

4. At Rev. 14, the output from the READY TRACE statement can be directed to a separate file in addition to the user terminal output. The system command COMOUT is used for this purpose. The command is given just prior to program execution; its format is:

COMO file-name

where file-name is a programmer supplied word.

At program completion, the system command, COMO -E will close the file. close the file.

All data resulting from READY TRACE will be output to file-name and can be SPOOLed or SLISTed at program termination.

5. It is a good technique to TRACE only a limited number of records, such that the output will not be too large to handle, thereby diminishing its value for debugging purposes.

RESET TRACE

FUNCTION:

This statement turns off the Prime tracing function.

FORMAT:

RESET TRACE

GENERAL RULE:

The RESET TRACE statement may be coded anywhere in the Procedure Division when a READY TRACE statement has been previously coded.

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 DECLARATIVE. One or the other of these options must be taken for indexed files. Refer to Appendix E for status codes.

6. 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 DECLARATIVE. One or the other of these options must be taken.

7. A sequential file using REWRITE must be a COBOL-created file other than a printer file, or any uncompressed file.

8. See Sections 19 and 20 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 satifies the specified condition, and to adjust the associated index-name to indicate that table element.

FORMAT:

```
SEARCH data-name-1 [ VARYING { data-name-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         }
```

SYNTAX RULES:

1. Data-name-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause.

2. Data-name-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. Condition-name-1, condition-name-2 may be any condition as described under Conditional Expressions in Section 12.

GENERAL RULES:

1. A SEARCH statement enables a serial type of search operation, starting with the current index setting.

   A. If, at the start of execution of the SEARCH statement, the index-name associated with data-name-1 contains a value which corresponds to an occurrence number greater than the highest permissible occurrence number for data-name-1, the SEARCH is terminated immediately. If the AT END phrase is specified, imperative-statement-1 is executed; if the AT END phrase is not specified, control passes to the next executable sentence.

   B. If, at the start of execution of the SEARCH statement, the index-name associated with data-name-1 contains a value corresponding to an occurrence number not greater than the highest permissible occurrence number for data-name-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 data-name-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 data-name-1 corresponds to a table element outside the permissible range of occurrence values, the search terminates as indicated in 1A 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. If imperative-statement-1, imperative-statement-2, or imperative-statement-3, that does not terminate with a GO TO statement, control passes to the next executable sentence.

3. 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 data-name-1. Any other index-names for data-name-1 remain unchanged.

4. If the VARYING index-name-1 phrase is specified, and if index-name-1 appears in the INDEXED BY phrase of data-name-1, that index-name is used for this search. If this is not the case, or if the VARYING data-name-2 phrase is specified, the first (or only) index-name given in the INDEXED BY phrase of data-name-1 is used for the search. In addition, the following operations will occur:

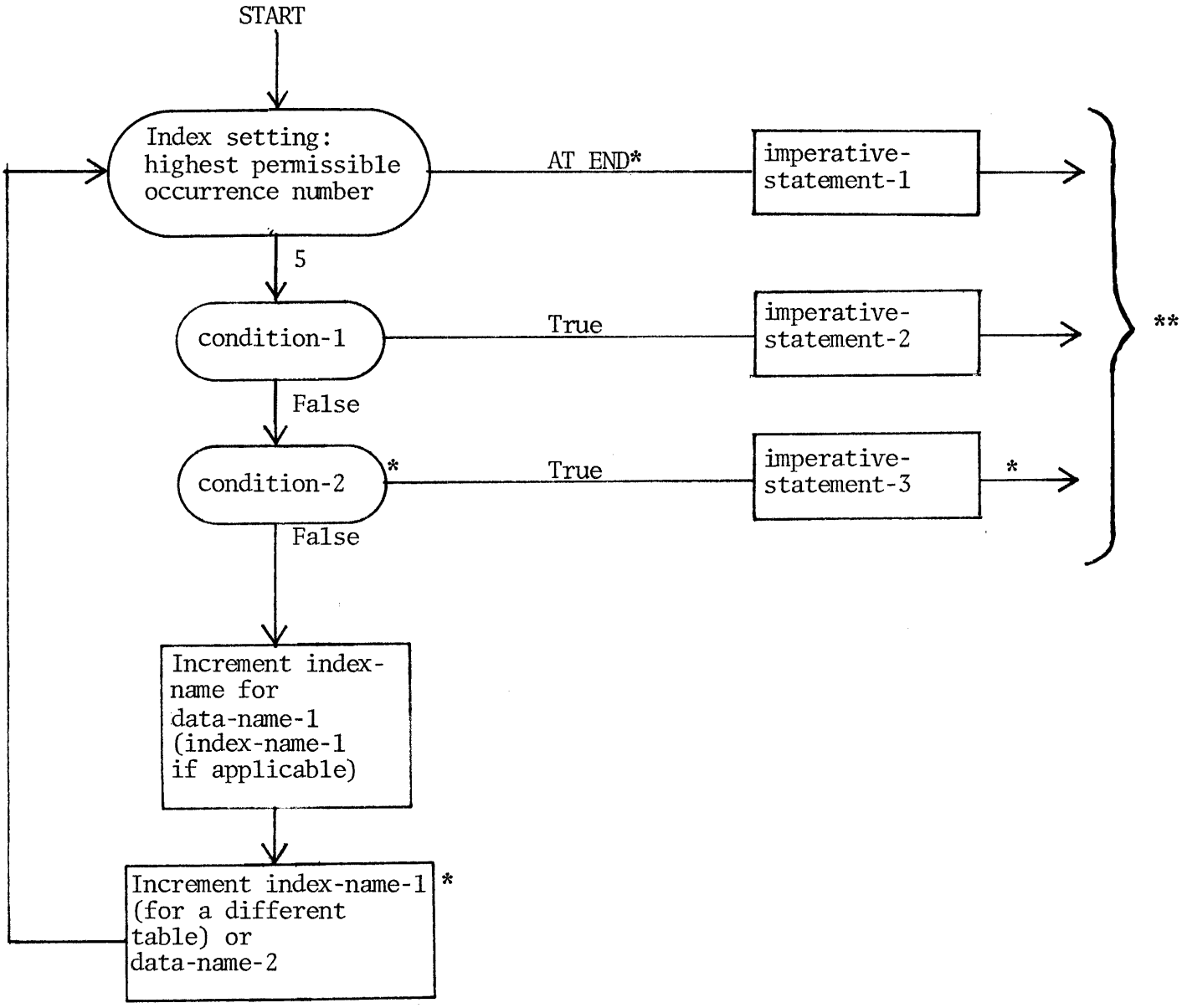
A. If the VARYING index-name-1 phrase is used, and if index-name-1 appears in the INDEXED BY phrase of another table entry, the occurrence number represented by index-name-1 is incremented by the same amount as, and at the same time as, the occurrence number represented by the index-name associated with data-name-1.

B. If the VARYING data-name-2 phrase is specified, and data-name-2 is an index data item, then the data item referenced by data-name-2 is incremented by the same amount as, and at the same time as, the index associated with data-name-1. If data-name-2 is not an index data item, the data item referenced by data-name-2 is incremented by the value one (1) at the same time as the index referenced by the index-name associated with data-name-1.

5. If data-name-1 is a data item subordinate to another 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 data-name-1 (and data-name-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.

6. A flow chart of the SEARCH operation containing two WHEN phrases is presented in Figure 16-3.

7. Additional information may be found in Section 18, Table Handling.



* These operations are options included only when specified in the SEARCH statement.

** Each of these control transfers is to the next executable sentence unless the imperative-statement ends with a GO TO statement.

Figure 16-3. SEARCH Operation Flowchart

SET

FUNCTION:

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements.

FORMAT 1:

```
SET  { index-name-1 }  ...TO  { index-name-3 }
     { data-name-1  }         { data-name-3 }
                              { integer-1   }
```

FORMAT 2:

```
SET  { index-name-4 }  ...  { UP BY   }  { index-name-6 }
     { data-name-4  }       { DOWN BY }  { data-name-6 }
                                         { integer-2   }
```

SYNTAX RULES:

1. All references to index-name-1, data-name-1, index-name-4 and data-name-4 apply equally to index-name-2, data-name-2, index-name-5, and data-name-5, respectively.

2. Data-name-6 must be described as an elementary numeric integer.

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 index-name-3 is specified, the value of the index before the execution of the SET statement must not exceed the occurrence number of an element in the associated table.

6. In Format 1, the following action occurs:

   A. 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.

   B. 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.

   C. 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.

   D. 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-2 is used as it was at the beginning of the execution of the statement.

7. In Format 2, 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-6; thereafter, the process is repeated for index-name-5, etc. Each time the value of data-name-6 is used as it was at the beginning of the execution of the statement.

8. See Section 18, TABLE HANDLING for additional information.

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.

   A. In an Indexed file, this option positions the file to the value contained in the RECORD KEY data-name.

   B. In a Relative file, this option positions the file to a value contained in the RELATIVE KEY data-name.

   C. In either file structure, if the indicated record is not present on the file, control is passed to the DECLARATIVES section if present; otherwise, the program terminates.

2. Option 2: START file-name KEY IS data-name.

   A. 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).

   B. In a Relative file, this option will position the file to the file to the value contained in data-name as defined in RELATIVE KEY.

   C. In either file structure, if the indicated record is not present on 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 / NOT LESS THAN / EQUAL TO }] data-name]

[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. This option allows the keys to contain partial values.

4. The INVALID clause or DECLARATIVES is taken if there is no data satisfying data-name and the STATUS code returned is a 23 on a full key.

STOP

FUNCTION:

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

FORMAT:

$$\underline{\text{STOP}} \left\{ \begin{matrix} \underline{\text{RUN}} \\ \text{literal} \end{matrix} \right\}$$

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 program, 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:

```
STRING {data-name-1} [, data-name-2] ... DELIMITED BY {data-name-3}
       {literal-1  }  [, literal-2  ]                  {literal-3  }
                                                       {SIZE       }

     [, {data-name-4} [, data-name-5] ... DELIMITED BY {data-name-6}
        {literal-4  }  [, literal-5  ]                 {literal-6  }
                                                       {SIZE       }] ...

     INTO data-name-7 [WITH POINTER data-name-8]

     [; ON OVERFLOW imperative-statement]
```

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:

A. 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.

B. 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.

C. 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 transferr 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. He 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 following occurs. 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.

SUBTRACT

FUNCTION:

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

FORMAT 1:

```
SUBTRACT {data-name-1} [, data-name-2] ... FROM data-name-m [ROUNDED]
         {literal-1  } [, literal-2  ]

[ON SIZE ERROR imperative-statement]
```

FORMAT 2:

```
SUBTRACT {data-name-1} [, data-name-2] ...
         {literal-1  } [, literal-2  ]

FROM {data-name-m} GIVING data-name-n
     {literal-m  }

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

SYNTAX RULES:

1. Each data-name must refer to a numeric elementary item, except that data-name-n (following GIVING) may be an elementary numeric edited item (report 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.

GENERAL RULES:

1. In Format 1, 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-m.

2. In Format 2, all literals and data-names preceding FROM are added together, the sum is subtracted from data-name-m or literal-m, and the result is stored in data-name-n.

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.

UNSTRING

FUNCTION:

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

FORMAT:

```
UNSTRING data-name-1
    [DELIMITED BY [ALL] {data-name-2 / literal-1} [, OR [ALL] {data-name-3 / literal-2}] ...]
    INTO data-name-4 [, DELIMITER IN data-name-5] [, COUNT IN data-name-6]
        [, data-name-7 [, DELIMITER IN data-name-8] [, COUNT IN data-name-9]] ...
    [WITH POINTER data-name-10] [TALLYING IN data-name-11]
    [; ON OVERFLOW imperative-statement]
```

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, 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-string), 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, liateral-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.

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 in General Rule 13D below.

9. When any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled according to the description of the receiving area.

10. Literal-1, or the contents of the data item referenced by data-name-2, can contain any character in the computer's character set.

11. 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 recognized as a delimiter.

12. 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 a part of more than one delimiter.

Each delimiter is applied to the sending field in the sequence specified in the UNSTRING statement.

13. 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:

A. 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 leftmost character position.

B. 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 11.) 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.

C. 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.

D. 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.

E. 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.

F. 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.

G. After data is transferred to data-name-4, the current receiving area is data-name-7. The behavior described in paragraphs 13C through 13F 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.

14. The initialization of the contents of the data items associated with the POINTER phrase or the TALLYING phrase is the responsibility of the user.

15. 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.

16. 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.

17. Either of the following situations causes an overflow condition:

A. An UNSTRING is initiated, and the value in the data item referenced by data-name-10 is less than 1 or greater than the size of the data item referenced by data-name-1.

B. 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.

18. When an overflow condition exists, the UNSTRING operation 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.

19. The evaluation of subscripting and indexing for the identifiers is as follows:

A. 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.

B. 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.

20. Up to five delimiters may be specified.

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:

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE ON {file-name }
                   {ERROR    }               {INPUT    }
                                             {OUTPUT   }
                                             {I-O      }
```

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 referended 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-O 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 1:

```
WRITE record-name [FROM data-name-1]

[ {AFTER  } ADVANCING {integer  LINE(s)} ]
  {BEFORE }           {PAGE             }
```

FORMAT 2:

```
WRITE record-name [FROM data-name-1]

    [INVALID KEY imperative-statement]
```

SYNTAX RULES:

1. Format 1 can only be used for sequential files.

2. Format 2 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 1, 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 1, 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.

   A. If the BEFORE option is taken, a line is written before advancing.

   B. If the AFTER option is taken, spacing occurs, and then the line is written.

   C. Integer LINE(s) is the number of spacing lines required between data lines. Integer may be Ø to 62.

   D. 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 1, the value of integer is as described in Table 16-4.

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

Table 16-4. Carriage Control Integer Values

5. In Format 2 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 DECLARATIVE section will be executed. Refer to Table 19-1 for Error Conditions.

6. In Format 2 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 (refer to Table 20-1.)

A. 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.

B. For Dynamic and Random Access:

The value of the record number must be placed in the RELATIVE KEY data-name-1.

REF 2

```
      *
      *
       PROCEDURE DIVISION.
       START-PROGRAM.
                 DISPLAY 'ENTER 1 TO CREATE NEW FILE'.
                 DISPLAY 'ENTER 2 TO UPDATE OLD FILE'.
                 ACCEPT CREATE-UPDATE.
                 IF CREATE-UPDATE = '2'
                           OPEN OUTPUT LIST-FILE
                           GO TO UPDATE-ONLY.
       CREATE-FILE.
                 MOVE SPACES TO WS-RECORD.
                 OPEN INPUT CARD-FILE, OPEN OUTPUT LIST-FILE,
                           DIRECTORY-FILE.
                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
       READ-NEXT.
                 READ CARD-FILE AT END GO TO LIST-DIRECTORY.
                 MOVE CARD-IMAGE TO PRINT-LINE.
                 WRITE PRINT-LINE.
                 MOVE SPACES TO DIR-1.
                 MOVE DATA-D1 TO D3.
                 MOVE PHONE-D1 TO D1.
                 MOVE D2 TO D4.
                 WRITE DIRECTORY-RECORD
                           INVALID KEY DISPLAY FILE-STATUS.
                 GO TO READ-NEXT.
       LIST-DIRECTORY.
                 CLOSE CARD-FILE, DIRECTORY-FILE.
                 MOVE 'END OF CREATE 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.
       LIST-DIR.
                 MOVE LOW-VALUE TO PHONE-NO.
                 PERFORM LIST THRU LIST-DONE.
                 MOVE LOW-VALUE TO LAST-NAME.
                 PERFORM LIST1 THRU LIST-DONE.
                 MOVE LOW-VALUE TO STATE.
                 PERFORM LIST2 THRU LIST-DONE.
                 MOVE ZEROS TO BIRTHD.
                 PERFORM LIST3 THRU LIST-DONE.
                 MOVE LOW-VALUE TO FIRST-NAME.
                 PERFORM LIST4 THRU LIST-DONE.
```

REF2

```
       LIST-DIR-EXIT.
            EXIT.
       START-PAR.
                   MOVE 'END OF TEST FOR START VERB' TO PRINT-LINE.
                   WRITE PRINT-LINE AFTER ADVANCING 3 LINES.
                   MOVE SPACES TO PRINT-LINE.
                   DISPLAY 'END OF TEST TWO'.
                   GO TO GET-NEXT-INQUIRY.
       LIST.       START DIRECTORY-FILE KEY IS NOT LESS THAN PHONE-NO.
                   WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
                   GO TO READ-NEXT-DIRECTORY-RECORD.
       LIST1.      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.      START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
                   WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
                   GO TO READ-NEXT-DIRECTORY-RECORD.
       LIST3.      START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTHD.
                   WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
                   GO TO READ-NEXT-DIRECTORY-RECORD.
       LIST4.      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 GO TO
                                LIST-DONE.
                   MOVE DIRECTORY-RECORD TO PRINT-LIN.
                   WRITE PRINT-LINE.
                   GO TO READ-NEXT-DIRECTORY-RECORD.
       LIST-DONE.
                   EXIT.
      *
      *
       GET-NEXT-INQUIRY.
                   DISPLAY 'ENTER TRAN TYPE'.
                   DISPLAY ' #   =  READ FILE SEQ'.
                   DISPLAY ' +   =  ADD'.
                   DISPLAY ' -   =  DELETE'.
                   DISPLAY ' /   =  CHANGE'.
                   DISPLAY ' *   =  QUIT'.
                   ACCEPT CHAR-1 FROM TTY.
                   IF CHAR-1 = '+', GO TO ADDITION.
                   IF CHAR-1 = '-', GO TO DELETION.
                   IF CHAR-1 = '/', GO TO CHANGE.
                   IF CHAR-1 = '*', GO TO WRAPUP.
                   IF CHAR-1 = '#', GO TO READ-FILE.
                   DISPLAY 'INVALID TRANS TYPE = ' CHAR-1.
                   DISPLAY 'TRY AGAIN'.
                   GO TO GET-NEXT-INQUIRY.
```

REF2

```
                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 INVALID KEY
                                DISPLAY FILE-STATUS
                                DISPLAY DISPLAY-DIR.
                    GO TO GET-NEXT-INQUIRY.
*
*
            DELETION.
                    DISPLAY 'ENTER PHONE NUMBER TO BE DELETED'.
                    ACCEPT PHONE-NO FROM TTY.
                    READ DIRECTORY-FILE 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 'BIRTHD     = 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 GET-NEXT-INQUIRY.
*
            READ-ALT1.
                    DISPLAY 'ENTER LAST NAME'.
                    ACCEPT WS-LAST-NAME.
                    MOVE SPACES TO DIRECTORY-RECORD.
                    MOVE WS-LAST-NAME TO LAST-NAME.
                    READ DIRECTORY-FILE KEY IS LAST-NAME
                                INVALID KEY DISPLAY 'LAST-NAME = ' LAST-NAME
                                DISPLAY 'STATUS = ' FILE-STATUS
                                DISPLAY DISPLAY-DIR
                                GO TO GET-NEXT-INQUIRY.
                    GO TO CHANGE-RECORD.
```

REF2

```
       READ-ALT2.
           DISPLAY 'ENTER STATE '.
           ACCEPT WS-STATE.
           MOVE SPACES TO DIRECTORY-RECORD.
           MOVE WS-STATE TO STATE.
           READ DIRECTORY-FILE KEY IS STATE
                INVALID KEY DISPLAY 'STATE = ' STATE
                DISPLAY 'STATUS = ' FILE-STATUS
                DISPLAY DISPLAY-DIR
                GO TO GET-NEXT-INQUIRY.
           GO TO CHANGE-RECORD.
      *
       READ-ALT3.
           DISPLAY 'ENTER BIRTHDAY'.
           ACCEPT WS-BIRTHD.
           MOVE SPACES TO DIRECTORY-RECORD.
           MOVE WS-BIRTHD TO BIRTHD.
           READ DIRECTORY-FILE KEY IS BIRTHD
                INVALID KEY DISPLAY 'BIRTHD = ' BIRTHD
                DISPLAY '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.
           MOVE WS-FIRST-NAME TO FIRST-NAME.
           READ DIRECTORY-FILE KEY IS FIRST-NAME
                INVALID KEY DISPLAY 'FIRST-NAME = ' FIRST-NAME
                DISPLAY '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.
```

REF2

```
                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-NO NOT = SPACES
                    MOVE WS-PHONE-NO TO PHONE-NO.
                IF WS-STATE NOT = SPACES
                    MOVE WS-STATE TO STATE.
                IF WS-BIRTHD NOT = SPACES
                    MOVE WS-BIRTHD TO BIRTHD.
            MOVE-EXIT.
                EXIT.
            *
            REWRITE-RECORD.
                REWRITE DIRECTORY-RECORD 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
                    NEXT SENTENCE
                        ELSE
                    GO TO READ-TYPE.
                IF PER-Co (1) NOT NUMERIC AND
                    PER-Co (2) NOT NUMERIC AND
                    PER-Co (3) NOT NUMERIC
                    MOVE ϕϕ2 TO PERFORM-COUNT
                    GO TO READ-TYPE.
                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 ZEROS 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
```

REF 2

```
                MOVE PER-CO (2) TO PER-CO (3)
                MOVE PER-CO (1) TO PER-CO (2)
                MOVE ZERO TO PER-CO (1).
*
*
*
*
       READ-TYPE.
                DISPLAY 'ENTER KEY TO BE READ'.
                DISPLAY 'PHONE-NO    = 1'.
                DISPLAY 'LAST-NAME   = 2'.
                DISPLAY 'STATE       = 3'.
                DISPLAY 'BIRTHD      = 4'.
                DISPLAY 'FIRST-NAME  = 5'.
                ACCEPT GO-TO-READ.
                IF GO-TO-READ NOT NUMERIC
                    DISPLAY 'INVALID KEY TRY AGAIN'
                    GO TO READ-TYPE.
                GO TO READ-1 READ-2 READ-3 READ-4 READ-5
                    DEPENDING ON GO-TO-READ.
*
*
       READ-1.
                MOVE LOW-VALUES TO PHONE-NO.
                START DIRECTORY-FILE KEY IS NOT LESS THAN PHONE-NO.
                GO TO READ-FILE-GO.
       READ-2.
                MOVE LOW-VALUES TO LAST-NAME.
                START DIRECTORY-FILE KEY IS NOT LESS THAN LAST-NAME.
                GO TO READ-FILE-GO.
       READ-3.
                MOVE LOW-VALUES TO STATE.
                START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
                GO TO READ-FILE-GO.
       READ-4.
                MOVE ZEROS TO BIRTHD.
                START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTHD.
                GO TO READ-FILE-GO.
       READ-5.
                MOVE LOW-VALUES TO FIRST-NAME.
                START DIRECTORY-FILE KEY IS NOT LESS THAN FIRST-NAME.
       READ-FILE-GO.
                READ DIRECTORY-FILE NEXT RECORD
                    AT END MOVE ZEROS TO PERFORM-COUNT
                        GO TO READ-FILE-EXIT.
                DISPLAY DISPLAY-DTR.
```

REF 2

```
000100 READ-FILE-EXIT.
000200     EXIT.
000300*
000400*
000500*
000600*
000700 WRAPUP.
000800     PERFORM LIST-DIR.
000900     MOVE 'END OF INDEXED TEST TO CHANGE FILE' TO PRINT-LINE.
001000     DISPLAY 'END OF INDEXED TEST'.
001100     CLOSE LIST-FILE, DIRECTORY-FILE.
001200     STOP RUN.
001300*
001400*
001500*
001600 FORMAT-INPUT
001700     MOVE SPACES TO WS-RECORD.
001800     DISPLAY 'ENTER LAST-NAME'.
001900     ACCEPT WS-LAST-NAME.
002000     DISPLAY 'ENTER FIRST NAME'.
002100     ACCEPT WS-FIRST-NAME.
002200     DISPLAY 'ENTER ADDRESS '.
002300     ACCEPT WS-ADDRESS.
002400     DISPLAY 'ENTER CITY '.
002500     ACCEPT WS-CITY.
002600     DISPLAY 'ENTER PHONE NUMBER '.
002700     ACCEPT WS-PHONE-NO.
002800     DISPLAY 'ENTER STATE XX'.
002900     ACCEPT WS-STATE.
003000     DISPLAY 'ENTER BIRTHDAY MMDDYY'.
003100     ACCEPT WS-BIRTHD.
003200
003300
```

COMPILE SEQUENCE FOR REF2

64R

OK, COBOL REF2 -64R

GO,

.

.

.

ØØØØ ERRORS ØØØØ WARNINGS (COBOL VER Ø4)

64V

OK, COBOL REF2 -64V

GO,

.

.

.

ØØØØ ERRORS ØØØØ WARNINGS, P400/500 COBOL REV 14.0 <REF2>

Print Listing File

OK, SPOOL L←REF2

LISTING File For Sample Program REF2 Compiled In 64V Mode

```
REV 14  COBOL     SOURCE FILE: REF2     09/20/77  11:51
 (0001)        *
 (0002)        *
 (0003)        *
 (0004)         IDENTIFICATION DIVISION.
 (0005)         PROGRAM-ID. REF2.
 (0006)         AUTHOR. PRIME COMPUTER.
 (0007)         INSTALLATION. FRAMINGHAM.
 (0008)         DATA-WRITTEN. SEPTEMBER, 1977.
 (0009)         DATE-COMPILED. SEPTEMBER, 1977.
 (0010)         REMARKS.  THIS AREA IS USED TO DESCRIBE THE PROGRAM.
 (0011)        *
 (0012)        *
 (0013)        *
 (0014)         ENVIRONMENT DIVISION.
 (0015)         CONFIGURATION SECTION.
 (0016)         SOURCE-COMPUTER. PRIME.
 (0017)         OBJECT-COMPUTER. PRIME.
 (0018)         SPECIAL-NAMES. CONSOLE IS TTY.
 (0019)             ASCII IS NATIVE.
 (0020)         INPUT-OUTPUT SECTION.
 (0021)         FILE-CONTROL.
 (0022)             SELECT LIST-FILE ASSIGN TO PRINTER.
 (0023)             SELECT CARD-FILE ASSIGN TO PFMS.
 (0024)             SELECT DIRECTORY-FILE ASSIGN TO PFMS,
 (0025)                ORGANIZATION IS INDEXED
 (0026)                ACCESS MODE IS DYNAMIC, RECORD KEY IS PHONE-NO,
 (0027)                ALTERNATE RECORD KEY LAST-NAME
 (0028)                ALTERNATE RECORD KEY STATE
 (0029)                ALTERNATE RECORD KEY BIRTHD
 (0030)                ALTERNATE RECORD KEY FIRST-NAME
 (0031)                FILE STATUS IS FILE-STATUS.
 (0032)        *
 (0033)        *
 (0034)        *
 (0035)        *
 (0036)        *
 (0037)         DATA DIVISION.
 (0038)         FILE SECTION.
 (0039)         FD  LIST-FILE, LABEL RECORDS ARE OMITTED.
 (0040)             01 PRINT-LINE, PICTURE X(100).
 (0041)             01 PRINT-LINE1.
 (0042)                 02  FILLER  PIC X.
 (0043)                 02  PRINT-LIN  PIC  X(99).
 (0044)         FD  CARD-FILE, LABEL RECORDS ARE STANDARD
 (0045)             VALUE OF FILE-ID IS 'INDAT1'.
 (0046)             01  CARD-IMAGE, PICTURE X(80).
 (0047)             01  CARD-D1.
 (0048)                 02  DATA-D1       PIC X(64).
 (0049)                 02  PHONE-D1      PIC X(8).
 (0050)                 02  D2            PIC X(8).
 (0051)         FD  DIRECTORY-FILE, LABEL RECORDS ARE STANDARD, VALUE OF FILE-ID
 (0052)                 IS  'INDXFILE'
 (0053)                 OWNER IS 'LDAVIS'.
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0054)             01  DIRECTORY-RECORD.
  (0055)                 02  PHONE-NO    PIC X(8).
  (0056)                 02  NAME.
  (0057)                     03  LAST-NAME   PIC X(14).
  (0058)                     03  FILLER      PIC X.
  (0059)                     03  FIRST-NAME  PIC X(13).
  (0060)                     03  FILLER      PIC XX.
  (0061)                 02  FILLER, PICTURE X.
  (0062)                 02  ADDRESS, PICTURE X(25).
  (0063)                 02  FILLER, PICTURE X.
  (0064)                 02  CITY,   PICTURE X(4).
  (0065)                 02  FILLER, PICTURE X(3).
  (0066)                 02  STATE,  PICTURE XX.
  (0067)                 02  BIRTHD, PICTURE 9(6).
  (0068)                 02  FILLER, PICTURE X(20).
  (0069)             01  DIR-1.
  (0070)                 02  DISPLAY-DIR     PIC X(72).
  (0071)                 02  FILLER          PIC X(28).
  (0072)             01  SOME-D1.
  (0073)                 02  D1          PIC  X(8).
  (0074)                 02  D3          PIC  X(64).
  (0075)                 02  D4          PIC  X(8).
  (0076)                 02  FILLER      PIC  X(20).
  (0077)         WORKING-STORAGE SECTION.
  (0078)             77  GO-TO-READ      PICTURE 9 VALUE 0.
  (0079)             77  CREATE-UPDATE   PICTURE X VALUE SPACE.
  (0080)             77  GO-TO-NAME      PICTURE 9 VALUE 0.
  (0081)             77  FILE-STATUS     PICTURE X(2)  VALUE IS SPACE.
  (0082)             77  CHAR-1          PICTURE X VALUE SPACE.
  (0083)             01  PERFORM-COUNT1.
  (0084)                 02  PERFORM-COUNT   PIC 999.
  (0085)                 02  PER-CO  REDEFINES PERFORM-COUNT
  (0086)                         PICTURE X, OCCURS 3 TIMES.
  (0087)             01  WS-RECORD.
  (0088)                 02  WS-LAST-NAME        PIC X(14).
  (0089)                 02  FILLER              PIC X.
  (0090)                 02  WS-FIRST-NAME       PIC X(13).
  (0091)                 02  FILLER              PIC XXX.
  (0092)                 02  WS-ADDRESS          PIC X(25).
  (0093)                 02  FILLER              PIC X.
  (0094)                 02  WS-CITY             PIC X(4).
  (0095)                 02  FILLER              PIC XXX.
  (0096)                 02  WS-PHONE-NO         PIC X(8).
  (0097)                 02  WS-STATE            PIC XX.
  (0098)                 02  WS-BIRTHD           PIC X(6).
  (0099)             01  HEADER.
  (0100)                 02  FILLER      PICTURE X   VALUE SPACE.
  (0101)                 02  H0          PIC X(8)    VALUE 'PHONE'.
  (0102)                 02  H1          PICTURE X(4) VALUE IS 'NAME'.
  (0103)                 02  FILLER      PICTURE X(27) VALUE IS SPACE.
  (0104)                 02  H2          PICTURE X(6)  VALUE IS 'STREET'.
  (0105)                 02  FILLER      PICTURE X(20) VALUE IS  SPACE.
  (0106)                 02  H3          PICTURE X(4)  VALUE IS  'CITY'.
  (0107)                 02  FILLER      PICTURE X(3)  VALUE IS  SPACE.
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0108)        *
  (0109)        *
  (0110)         PROCEDURE DIVISION.
  (0111)         START-PROGRAM.
  (0112)                 DISPLAY 'ENTER 1 TO CREATE NEW FILE'.
  (0113)                 DISPLAY 'ENTER 2 TO UPDATE OLD FILE'.
  (0114)                 ACCEPT CREATE-UPDATE.
  (0115)                 IF CREATE-UPDATE = '2'
  (0116)                         OPEN OUTPUT LIST-FILE
  (0117)                         GO TO UPDATE-ONLY.
  (0118)         CREATE-FILE.
  (0119)                 MOVE SPACES TO WS-RECORD.
  (0120)                 OPEN INPUT CARD-FILE, OPEN OUTPUT LIST-FILE,
  (0121)                         DIRECTORY-FILE.
  (0122)                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
  (0123)         READ-NEXT.
  (0124)                 READ CARD-FILE AT END GO TO LIST-DIRECTORY.
  (0125)                 MOVE CARD-IMAGE TO PRINT-LINE.
  (0126)                 WRITE PRINT-LINE.
  (0127)                 MOVE SPACES TO DIR-1.
  (0128)                 MOVE DATA-D1 TO D3.
  (0129)                 MOVE PHONE-D1 TO D1.
  (0130)                 MOVE D2 TO D4.
  (0131)                 WRITE DIRECTORY-RECORD
  (0132)                     INVALID KEY DISPLAY FILE-STATUS.
  (0133)                 GO TO READ-NEXT.
  (0134)         LIST-DIRECTORY.
  (0135)                 CLOSE CARD-FILE, DIRECTORY-FILE.
  (0136)                 MOVE 'END OF CREATE FILE' TO PRINT-LINE.
  (0137)                 WRITE PRINT-LINE AFTER ADVANCING 3 LINES.
  (0138)         UPDATE-ONLY.
  (0139)                 MOVE SPACES TO PRINT-LINE.
  (0140)                 DISPLAY 'END TEST ONE'
  (0141)                 OPEN I-O DIRECTORY-FILE.
  (0142)                 IF CREATE-UPDATE = '2'
  (0143)                     GO TO GET-NEXT-INQUIRY.
  (0144)         LIST-DIR.
  (0145)                 MOVE LOW-VALUE TO PHONE-NO.
  (0146)                 PERFORM LIST THRU LIST-DONE.
  (0147)                 MOVE LOW-VALUE TO LAST-NAME.
  (0148)                 PERFORM LIST1 THRU LIST-DONE.
  (0149)                 MOVE LOW-VALUE TO STATE.
  (0150)                 PERFORM LIST2 THRU LIST-DONE.
  (0151)                 MOVE ZEROS TO BIRTHD.
  (0152)                 PERFORM LIST3 THRU LIST-DONE.
  (0153)                 MOVE LOW-VALUE TO FIRST-NAME.
  (0154)                 PERFORM LIST4 THRU LIST-DONE.
  (0155)         LIST-DIR-EXIT.
  (0156)             EXIT.
  (0157)         START-PAR.
  (0158)                 MOVE 'END OF TEST FOR START VERB' TO PRINT-LINE.
  (0159)                 WRITE PRINT-LINE AFTER ADVANCING 3 LINES.
  (0160)                 MOVE SPACES TO PRINT-LINE.
  (0161)                 DISPLAY 'END OF TEST TWO'.
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0162)                 GO TO GET-NEXT-INQUIRY.
  (0163)          LIST.  START DIRECTORY-FILE KEY IS NOT LESS THAN PHONE-NO.
  (0164)                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
  (0165)                 GO TO READ-NEXT-DIRECTORY-RECORD.
  (0166)          LIST1. START DIRECTORY-FILE KEY IS NOT LESS THAN LAST-NAME.
  (0167)                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
  (0168)                 GO TO READ-NEXT-DIRECTORY-RECORD.
  (0169)          LIST2. START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
  (0170)                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
  (0171)                 GO TO READ-NEXT-DIRECTORY-RECORD.
  (0172)          LIST3. START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTHD.
  (0173)                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
  (0174)                 GO TO READ-NEXT-DIRECTORY-RECORD.
  (0175)          LIST4. START DIRECTORY-FILE KEY IS NOT LESS THAN FIRST-NAME.
  (0176)                 WRITE PRINT-LINE FROM HEADER AFTER ADVANCING PAGE.
  (0177)          READ-NEXT-DIRECTORY-RECORD.
  (0178)                 READ DIRECTORY-FILE NEXT RECORD AT END GO TO
  (0179)                         LIST-DONE.
  (0180)                 MOVE DIRECTORY-RECORD TO PRINT-LIN.
  (0181)                 WRITE PRINT-LINE.
  (0182)                 GO TO READ-NEXT-DIRECTORY-RECORD.
  (0183)          LIST-DONE.
  (0184)                 EXIT.
  (0185)         *
  (0186)         *
  (0187)          GET-NEXT-INQUIRY.
  (0188)                 DISPLAY 'ENTER TRAN TYPE'.
  (0189)                 DISPLAY ' #  =  READ FILE SEQ'.
  (0190)                 DISPLAY ' +  =  ADD'.
  (0191)                 DISPLAY ' -  =  DELETE'.
  (0192)                 DISPLAY ' /  =  CHANGE'.
  (0193)                 DISPLAY ' *  =  QUIT'.
  (0194)                 ACCEPT CHAR-1 FROM TTY.
  (0195)                 IF CHAR-1 = '+', GO TO ADDITION.
  (0196)                 IF CHAR-1 = '-', GO TO DELETION.
  (0197)                 IF CHAR-1 = '/', GO TO CHANGE.
  (0198)                 IF CHAR-1 = '*', GO TO WRAPUP.
  (0199)                 IF CHAR-1 = '#', GO TO READ-FILE.
  (0200)                 DISPLAY 'INVALID TRANS TYPE =  ' CHAR-1.
  (0201)                 DISPLAY 'TRY AGAIN'.
  (0202)                 GO TO GET-NEXT-INQUIRY.
  (0203)          NO-SUCH-NAME.
  (0204)                 DISPLAY ' NO SUCH RECORD =  ' DISPLAY-DIR.
  (0205)                 GO TO GET-NEXT-INQUIRY.
  (0206)         *
  (0207)         *
  (0208)         *
  (0209)         *
  (0210)          ADDITION.
  (0211)                 DISPLAY 'ENTER DATA RECORD FOR ADD'.
  (0212)                 PERFORM FORMAT-INPUT.
  (0213)                 PERFORM MOVE-REC.
  (0214)                 WRITE DIRECTORY-RECORD INVALID KEY
  (0215)                         DISPLAY FILE-STATUS
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0216)                      DISPLAY DISPLAY-DIR.
  (0217)                GO TO GET-NEXT-INQUIRY.
  (0218)         *
  (0219)         *
  (0220)          DELETION.
  (0221)                DISPLAY 'ENTER PHONE NUMBER TO BE DELETED'.
  (0222)                ACCEPT PHONE-NO FROM TTY.
  (0223)                READ DIRECTORY-FILE  INVALID KEY GO TO
  (0224)                       NO-SUCH-NAME.
  (0225)                DELETE DIRECTORY-FILE RECORD INVALID KEY GO TO
  (0226)                       NO-SUCH-NAME.
  (0227)                GO TO GET-NEXT-INQUIRY.
  (0228)         *
  (0229)         *
  (0230)          CHANGE.
  (0231)                DISPLAY 'ENTER KEY TO BE CHANGED'.
  (0232)                DISPLAY 'LAST-NAME  = 1'.
  (0233)                DISPLAY 'STATE      = 2'.
  (0234)                DISPLAY 'BIRTHD     = 3'.
  (0235)                DISPLAY 'FIRST-NAME = 4'.
  (0236)                ACCEPT GO-TO-NAME.
  (0237)                GO TO READ-ALT1 READ-ALT2 READ-ALT3 READ-ALT4
  (0238)                       DEPENDING ON GO-TO-NAME.
  (0239)                DISPLAY 'WRONG TYPE ENTERED TRY AGAIN'.
  (0240)                GO TO GET-NEXT-INQUIRY.
  (0241)         *
  (0242)          READ-ALT1.
  (0243)                DISPLAY 'ENTER LAST NAME'.
  (0244)                ACCEPT WS-LAST-NAME.
  (0245)                MOVE SPACES TO DIRECTORY-RECORD.
  (0246)                MOVE WS-LAST-NAME TO LAST-NAME.
  (0247)                READ DIRECTORY-FILE KEY IS LAST-NAME
  (0248)                       INVALID KEY DISPLAY 'LAST-NAME = ' LAST-NAME
  (0249)                       DISPLAY 'STATUS = ' FILE-STATUS
  (0250)                       DISPLAY DISPLAY-DIR
  (0251)                       GO TO GET-NEXT-INQUIRY.
  (0252)                GO TO CHANGE-RECORD.
  (0253)         *
  (0254)          READ-ALT2.
  (0255)                DISPLAY 'ENTER STATE '.
  (0256)                ACCEPT WS-STATE.
  (0257)                MOVE SPACES TO DIRECTORY-RECORD.
  (0258)                MOVE WS-STATE TO STATE.
  (0259)                READ DIRECTORY-FILE KEY IS STATE
  (0260)                    INVALID KEY DISPLAY, 'STATE = ' STATE
  (0261)                    DISPLAY 'STATUS = ' FILE-STATUS
  (0262)                    DISPLAY DISPLAY-DIR
  (0263)                    GO TO GET-NEXT-INQUIRY.
  (0264)                GO TO CHANGE-RECORD.
  (0265)         *
  (0266)          READ-ALT3.
  (0267)                DISPLAY 'ENTER BIRTHDAY'.
  (0268)                ACCEPT WS-BIRTHD.
  (0269)                MOVE SPACES TO DIRECTORY-RECORD.
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0270)                 MOVE WS-BIRTHD TO BIRTHD.
  (0271)                 READ DIRECTORY-FILE KEY IS BIRTHD
  (0272)                     INVALID KEY DISPLAY 'BIRTHD = ' BIRTHD
  (0273)                     DISPLAY 'STATUS = ' FILE-STATUS
  (0274)                     DISPLAY DISPLAY-DIR
  (0275)                     GO TO GET-NEXT-INQUIRY.
  (0276)                 GO TO CHANGE-RECORD.
  (0277)           *
  (0278)            READ-ALT4.
  (0279)                 DISPLAY 'ENTER FIRST-NAME'.
  (0280)                 ACCEPT WS-FIRST-NAME.
  (0281)                 MOVE SPACES TO DIRECTORY-RECORD.
  (0282)                 MOVE WS-FIRST-NAME TO FIRST-NAME.
  (0283)                 READ DIRECTORY-FILE KEY IS FIRST-NAME
  (0284)                     INVALID KEY DISPLAY 'FIRST-NAME = ' FIRST-NAME
  (0285)                     DISPLAY 'STATUS = ' FILE-STATUS
  (0286)                     DISPLAY DISPLAY-DIR
  (0287)                     GO TO GET-NEXT-INQUIRY.
  (0288)           *
  (0289)           *
  (0290)            CHANGE-RECORD.
  (0291)                 DISPLAY DISPLAY-DIR.
  (0292)                 PERFORM FORMAT-INPUT.
  (0293)           *
  (0294)           *
  (0295)            MOVE-REC.
  (0296)                 IF WS-RECORD = SPACES
  (0297)                     DISPLAY 'NO DATA ENTERED TRY AGAIN'
  (0298)                     GO TO GET-NEXT-INQUIRY.
  (0299)                 IF WS-LAST-NAME NOT = SPACES
  (0300)                     MOVE WS-LAST-NAME TO LAST-NAME.
  (0301)                 IF WS-FIRST-NAME NOT = SPACES
  (0302)                     MOVE WS-FIRST-NAME TO FIRST-NAME.
  (0303)                 IF WS-ADDRESS NOT = SPACES
  (0304)                     MOVE WS-ADDRESS TO ADDRESS.
  (0305)                 IF WS-CITY NOT = SPACES
  (0306)                     MOVE WS-CITY TO CITY.
  (0307)                 IF WS-PHONE-NO NOT = SPACES
  (0308)                     MOVE WS-PHONE-NO TO PHONE-NO.
  (0309)                 IF WS-STATE NOT = SPACES
  (0310)                     MOVE WS-STATE TO STATE.
  (0311)                 IF WS-BIRTHD NOT = SPACES
  (0312)                     MOVE WS-BIRTHD TO BIRTHD.
  (0313)            MOVE-EXIT.
  (0314)                 EXIT.
  (0315)           *
  (0316)            REWRITE-RECORD.
  (0317)                 REWRITE DIRECTORY-RECORD INVALID KEY
  (0318)                     GO TO NO-SUCH-NAME.
  (0319)                 GO TO GET-NEXT-INQUIRY.
  (0320)           *
  (0321)           *
  (0322)           *
  (0323)            READ-FILE.
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0324)                 MOVE ZEROS TO PERFORM-COUNT.
  (0325)                 DISPLAY 'ENTER NUMBER OF RECORDS TO BE READ'.
  (0326)                 ACCEPT PERFORM-COUNT.
  (0327)                 IF  PERFORM-COUNT = ZEROS
  (0328)                     DISPLAY 'NO RECORD COUNT ENTERED'
  (0329)                     GO TO GET-NEXT-INQUIRY.
  (0330)                 IF PERFORM-COUNT1 NOT NUMERIC
  (0331)                     NEXT SENTENCE
  (0332)                         ELSE
  (0333)                     GO TO READ-TYPE.
  (0334)                 IF PER-CO (1) NOT NUMERIC  AND
  (0335)                     PER-CO (2) NOT NUMERIC AND
  (0336)                     PER-CO (3) NOT NUMERIC
  (0337)                     MOVE 002 TO PERFORM-COUNT
  (0338)                     GO TO READ-TYPE.
  (0339)                 IF PER-CO (1) NUMERIC   AND
  (0340)                     PER-CO (2) NOT NUMERIC AND
  (0341)                     PER-CO (3) NOT NUMERIC
  (0342)                     MOVE PER-CO (1) TO PER-CO (3)
  (0343)                     MOVE '0' TO PER-CO (1) PER-CO (2)
  (0344)                     GO TO READ-TYPE.
  (0345)                 IF PER-CO (1) NUMERIC   AND
  (0346)                     PER-CO (2) NUMERIC  AND
  (0347)                     PER-CO (3) NOT NUMERIC
  (0348)                     MOVE PER-CO (2) TO PER-CO (3)
  (0349)                     MOVE PER-CO (1) TO PER-CO (2)
  (0350)                     MOVE '0' TO PER-CO (1).
  (0351)           *
  (0352)           *
  (0353)           *
  (0354)           *
  (0355)            READ-TYPE.
  (0356)                 DISPLAY 'ENTER KEY TO BE READ'.
  (0357)                 DISPLAY 'PHONE-NO   = 1'.
  (0358)                 DISPLAY 'LAST-NAME  = 2'.
  (0359)                 DISPLAY 'STATE      = 3'.
  (0360)                 DISPLAY 'BIRTHD     = 4'.
  (0361)                 DISPLAY 'FIRST-NAME = 5'.
  (0362)                 ACCEPT GO-TO-READ.
  (0363)                 IF GO-TO-READ NOT NUMERIC
  (0364)                     DISPLAY 'INVALID KEY TRY AGAIN'
  (0365)                     GO TO READ-TYPE.
  (0366)                 GO TO READ-1 READ-2 READ-3 READ-4 READ-5
  (0367)                     DEPENDING ON GO-TO-READ.
  (0368)           *
  (0369)           *
  (0370)            READ-1.
  (0371)                 MOVE LOW-VALUES TO PHONE-NO.
  (0372)                 START DIRECTORY-FILE KEY IS NOT LESS THAN PHONE-NO.
  (0373)                 GO TO READ-FILE-GO.
  (0374)            READ-2.
  (0375)                 MOVE LOW-VALUES TO LAST-NAME.
  (0376)                 START DIRECTORY-FILE KEY IS NOT LESS THAN LAST-NAME.
  (0377)                 GO TO READ-FILE-GO.
```

```
REV 14 COBOL     SOURCE FILE: REF2          09/20/77  11:51
  (0378)         READ-3.
  (0379)                 MOVE LOW-VALUES TO STATE.
  (0380)                 START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
  (0381)                 GO TO READ-FILE-GO.
  (0382)         READ-4.
  (0383)                 MOVE ZEROS TO BIRTHD.
  (0384)                 START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTHD.
  (0385)                 GO TO READ-FILE-GO.
  (0386)         READ-5.
  (0387)                 MOVE LOW-VALUES TO FIRST-NAME.
  (0388)                 START DIRECTORY-FILE KEY IS NOT LESS THAN FIRST-NAME.
  (0389)         READ-FILE-GO.
  (0390)                 READ DIRECTORY-FILE NEXT RECORD
  (0391)                     AT END MOVE ZEROS TO PERFORM-COUNT
  (0392)                     GO TO READ-FILE-EXIT.
  (0393)                 DISPLAY DISPLAY-DIR.
  (0394)         READ-FILE-EXIT.
  (0395)                 EXIT.
  (0396)        *
  (0397)        *
  (0398)        *
  (0399)        *
  (0400)         WRAPUP.
  (0401)                 PERFORM LIST-DIR.
  (0402)                 MOVE 'END OF INDEXED TEST TO CHANGE FILE' TO PRINT-LINE.
  (0403)                 DISPLAY 'END OF INDEXED TEST'.
  (0404)                 CLOSE LIST-FILE, DIRECTORY-FILE.
  (0405)                 STOP RUN.
  (0406)        *
  (0407)        *
  (0408)        *
  (0409)         FORMAT-INPUT.
  (0410)                 MOVE SPACES TO WS-RECORD.
  (0411)                 DISPLAY 'ENTER LAST NAME'.
  (0412)                 ACCEPT WS-LAST-NAME.
  (0413)                 DISPLAY 'ENTER FIRST NAME'.
  (0414)                 ACCEPT WS-FIRST-NAME.
  (0415)                 DISPLAY 'ENTER ADDRESS '.
  (0416)                 ACCEPT WS-ADDRESS.
  (0417)                 DISPLAY 'ENTER CITY '.
  (0418)                 ACCEPT WS-CITY.
  (0419)                 DISPLAY 'ENTER PHONE NUMBER '.
  (0420)                 ACCEPT WS-PHONE-NO.
  (0421)                 DISPLAY 'ENTER STATE XX'.
  (0422)                 ACCEPT WS-STATE.
  (0423)                 DISPLAY 'ENTER BIRTHDAY MMDDYY'.
  (0424)                 ACCEPT WS-BIRTHD.

 0000 ERRORS   0000 WARNINGS, P400/500 COBOL REV 14.0 <REF2>
```

LOAD SEQUENCE FOR REF2

64R

```
OK, HILOAD
$ MO D64R                 set mode
$ CO 120000               move common
$ LO B<REF2
$ AU 20
$ LI COBKID               load COBOL MIDAS library
$ LI                      load FORTRAN library
LC                        load complete
$ SAVE *REF2              save memory image
$ QUIT                    return to PRIMOS
```

64V/SEG

```
OK, SEG
GO,
# VLOAD #REF2
$ LO B<REF2
$ LIB VCOBLB              load SEG COBOL library
$ LIB VKDALB              load SEG, COBOL MIDAS library
$ LI                      load the FORTRAN library
LC                        load complete prompt
$ SAVE                    the memory image is saved as #REF2
$ QUIT                    return to PRIMOS
```

CREATK SEQUENCE FOR REF2

The following represents the minimum dialogue to create the MIDAS template for sample program REF2 (underlining indicates user response):

OK, CREATK	
MINIMUM OPTIONS? YES	
FILENAME? DIRECTORY-FILE	The name of the file in the COBOL program which is to be indexed.
NEW FILE? YES	
DIRECT ACCESS? NO	
KEY TYPE: B	Key type is Binary
KEY SIZE=: B 64	Eight times the characters in the primary key, PHONE-NO.
DATA SIZE=: 50	
SECONDARY INDEX	
INDEX NO.? 1	
DUPLICATE KEYS PERMITTED? YES	
KEY TYPE: B	
KEY SIZE=: B 112	
USER DATA SIZE=: Ø	
INDEX NO.? 2	
DUPLICATE KEYS PERMITTED? YES	
KEY TYPE: B	
KEY SIZE=: B 16	
USER DATA SIZE=: Ø	
INDEX NO.? 3	
DUPLICATE KEYS PERMITTED? YES	
KEY TYPE: B	
KEY SIZE=: B 48	

```
USER DATA SIZE=: Ø
INDEX NO.? 4
DUPLICATE KEYS PERMITTED? YES
KEY TYPE: B
KEY SIZE=: 104
USER DATA SIZE=: Ø
INDEX NO? (CR)
```

EXECUTE SEQUENCE FOR REF2

64R

OK, R *REF2

ENTER FILENAME AND UNIT

>INDAT1=INDAT1

>INDXFILE=DIRECTORY-FILE

>/

64V

OK, SEG #REF2

ENTER FILENAME AND UNIT

>INDAT1=INDAT1

>INDXFILE=DIRECTORY-FILE

>/

FUNCTIONAL PROCESSING MODULES

REFERENCE

SECTION 17

INTER-PROGRAM COMMUNICATION

DEFINITION

Inter-Program Communication provides a facility by which a program can communicate with one or more 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 object 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 these 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 object 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 that:

1. The VALUE clause may not be specified for other than level 88 items;

2. Data-names used in the LINKAGE SECTION must be unique (may not be qualified);

3. Level 01 and 77 items must start on a computer word boundary. The programmer must ensure proper alignment between an argument (pointer to data) in a CALL statement and the corresponding data-name in a USING list on a subprogram Procedure header;

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 header so that data in the calling program may be manipulated in the subprogram.

CALL Statement

The CALL statement format is:

CALL 'literal' [USING data-name-1 data-name-n]

where literal is a subprogram name defined as the PROGRAM-ID of a separately compiled program and must be enclosed in quote marks. (The relationship of literal and PROGRAM-ID is illustrated in the example at the end of this section.)

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. At this time, data-name-n must not exceed 15.

NOTE: Correspondence between caller and callee lists are positional, not by identical spelling of names. For additional information, see CALL statement in the PROCEDURE DIVISION SECTION.

EXIT PROGRAM Statement

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. This statement must be a paragraph by itself.

ENTER Statement

An ENTER statement is classified as a compiler-directing statement; it acts as a modifier to a subsequent CALL statement.

A subprogram which is called may have been written in COBOL, FORTRAN, or ASSEMBLER language. The ENTER statement provides the means to identify the language in which a subprogram is written.

The general format is:

ENTER { COBOL / ASSEMBLER }

ENTER ASSEMBLER tells the compiler that the ensuing callee is not a COBOL subprogram.

ENTER COBOL tells the compiler that the ensuing callee is a COBOL subprogram.

ENTER COBOL may also be used following a CALL statement. This traditional usage is optional; after any CALL statement, ENTER COBOL is assumed.

EXAMPLE:

Filename = CALLER

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
Ø1  WS-ITEM   PICTURE  9(5).          Parameter being passed
PROCEDURE DIVISION.
FIRST-PARAGRAPH.
     CALL 'CALLED1' USING WS-ITEM.    The name in quotations must be
     STOP RUN.                        the Program-Id-name, not the
         .                            file-name.
         .
         .
         .
```

Filename = CALLED

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLED1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
Ø1  WS-TEST   PICTURE  9(5).
LINKAGE SECTION.
Ø1  WS-ITEM   PICTURE  9(5).
*WS-ITEM MUST BE DESCRIBED IN THE SAME MANNER
*IN BOTH THE CALLING AND THE CALLED PROGRAM.
*ONLY Ø1 AND 77 LEVEL ITEMS MAY BE CODED HERE.
 PROCEDURE DIVISION USING WS-ITEM.
 FIRST-PARAGRAPH.
      MOVE WS-TEST TO WS-ITEM.
      EXIT PROGRAM.
```

SECTION 18

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

OCCURS

The OCCURS clause eliminates the need for separate entries for repeated data items. Further, it supplies information required for the application of subscripts or indices. The OCCURS clause cannot be used on a level 77 or 88.

Data Description clauses associated with an item whose description includes an OCCURS clause apply to each repetition of the item being described. When OCCURS is used, the data-name which is the defining name of the entry must be subscripted (or if the INDEXED BY phrase is specified, must be indexed) whenever it appears in the Procedure Division. If the data-nama applies to a group item, all data-names belonging to the group must be subscripted (or indexed) whenever they are used.

The OCCURS clause format is:

```
[OCCURS integer TIMES [INDEXED BY index-name-1
    [index-name-2...]]]
```

INDEXED BY

The format of the INDEXED BY clause appears directly above. Index-name is not declared in the usual method of: INDEXED BY.

The format of the INDEXED BY phrase is:

```
[INDEXED BY index-name-1 [index-name-2...]]
```

when used, the INDEXED BY phrase is appended to the OCCURS clause. It is required if the subject of this entry, or one subordinate to this 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.

For this reason, index-name is not declared in the usual method of: level number, name, Data Description clauses. Rather, the declaration is implicit in the appearance of an "INDEXED BY index-name" appendage to an OCCURS clause.

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 CALL statement USING list, a Procedure header USING list, as the variation item in PERFORM VARYING, by a SEARCH statement, or in a relational condition. In all cases, the process is equivalent to dealing with a binary word integer subscript. A maximum of 3 indexes may be used on any given data-name.

Relative indexing may be specified wherever indexing can be specified. In this instance, index-name is followed by one of the operators + or -, followed by an unsigned, integer numeric literal, all delimited by the balanced pair of separators left parenthesis and right parenthesis.

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 a statement is executed which refers to an indexed table element, 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 applies equally to indexing and relative indexing.

The general format for indexing is:

```
{data-name      }  (  {index-name-1 [ {±} literal-2]}
{condition-name }     {literal-1                    }

[ , {index-name-2 [ {±} literal-4]} ]  ...)
    {literal-3                     }

[ , {index-name-3 [ {±} literal-6]} ]  ...)
    {literal-5                     }
```

Subscripting

When an OCCURS clause is specified for an item in the Data Division, that item must be subscripted or indexed whenever it is used.

Subscripting provides the facility for referring to those data items in a table or list which have not been assigned individual data-names.

The format is:

```
data-name (subscript-1 [,subscript-2 [,subscript-3]])
```

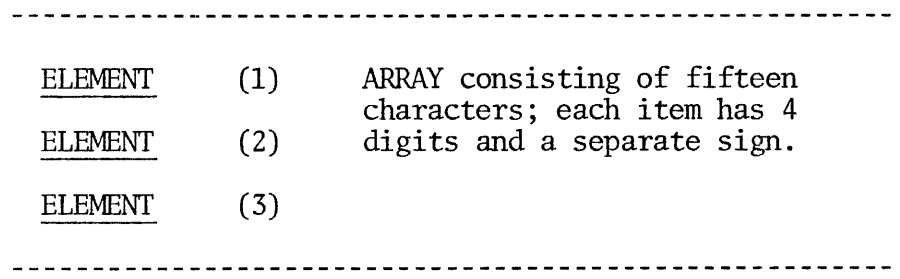
The subscript can be represented either by a positive numeric literal or by a data-name. Such a data-name must be a numeric, elementary item which represents an integer. The data-name as subscript may be qualified but not itself subscripted.

The subscript data-name may be signed, but the value must be positive. The lowest value which the subscript can contain is 1 (this would point to the first occurrence of the data within a table.) Thus, the subscript contains the numeric 'OCCURS' number within a table; its value must not exceed the 'OCCURS' integer for the table with which it is associated. The subscript can be used on any table.

EXAMPLE:

```
01 ARRAY
   03 ELEMENT, OCCURS 3, PICTURE S9(4), SIGN TRAILING SEPARATE.
```

The example above would be allocated storage as shown below:



A data-name may not be subscripted if it is being used for any of the following functions:

1. When it is being used as a subscript;

2. When it appears as the defining name of a Data Description entry;

3. When it appears as data-name-2 in a REDEFINES clause.

A maximum of three (3) subscripts can be used on any given data item. Multiple subscripts are separated by a comma.

A subscript value is changed in the Procedure Division via MOVE, ADD, or SUBTRACT verbs. The SET verb cannot be used on a subscript data-name.

PROCEDURE DIVISION

SET Statement

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

FORMAT 1:

```
SET { index-name-1 }  ... TO { index-name-3 }
    { data-name-1  }         { data-name-3  }
                             { integer-1    }
```

FORMAT 2:

```
SET { index-name-4 }  ... { UP      } { index-name-6 }
    { data-name-4  }      { DOWN BY }  { data-name-6 }
                                       { integer-2   }
```

Format 1 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.

Format 2 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.

An index-name should only apply to the OCCURS which define it.

SEARCH statement

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.

The format is:

```
SEARCH data-name-1 [VARYING { data-name-2  } ]
                            { index-name-1 }

[; WHEN condition-1 { imperative-statement-2 }
                    { NEXT SENTENCE          }

[; WHEN condition-2 { imperative-statement-3 }
                    { NEXT SENTENCE          }
```

A SEARCH statement enables a serial type of search operation, starting with the current index setting.

Data-name-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause. Data-name-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.

A complete discussion of the SEARCH verb is presented in Section 16, Procedure Division.

SECTION 19

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:

1. SELECT file-name

The SELECT clause specifies the name of the indexed sequential file. Refer to Environment Division for rules.

2. ORGANIZATION IS INDEXED

This clause specifies that the file named in the SELECT statement contains data organized by indices, and that it is to be processed by the Multiple Index Data Access System, MIDAS.

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

The ACCESS MODE clause specifies how an indexed file is written or retrieved.

A. 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 records sequentially.

B. RANDOM

When 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-n, prior to a READ, otherwise the record will not be found. Random mode precludes a sequential READ or WRITE.

C. DYNAMIC

When DYNAMIC access method is specified, a program can read or write randomly or sequentially.

4. RECORD KEY IS data name-1

The RECORD KEY clause specifies the data item within each record which is used for the primary index.

A. Data-name-1 must be defined in the Record Description FD entry.

B. 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.

C. Data-name-1 must not be specified with an OCCURS clause, or be contained within a group affected by an OCCURS clause.

D. Data-name-1 must not be specified with a P character in its PICTURE clause, with a separator sign (/).

E. Data-name-1 must have the same description and relative location as when the file was created.

F. Data-name-1 cannot exceed 32 characters.

G. The value contained within data-name-1 must be unique, duplicates are invalid.

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

This key field allows secondary indices. There may be up to 5 additional key fields.

See Rules C through F under RECORD KEY.

Seconday indices cannot be embedded within the primary index.

Specification of WITH DUPLICATES for an ALTERNATE RECORD allows keys containing the same value to be placed in the file. WITH DUPLICATES must be specified when the file template is created; it cannot be changed at the program level.

6. [FILE STATUS IS data-name-3]

The FILE STATUS is a two-character (one word) unsigned field described in the Working-Storage section. After each access to the operating system, a status code is placed in this field. For a successful read or write, etc., the status code contains 00. If the INVALID KEY or DECLARATIVES were involved, the error status code is returned. The programmer then can determine what action to take. Refer to the following table for error codes:

FILE ORGANIZATION		STATUS KEY 1		STATUS KEY 2
Indexed	Ø	- Successful completion	Ø	- No further information
	1	- End of file[a]	Ø	- 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]	Ø	- No further information
	9	- Implementor - defined	Ø	- Locked record[g]
			1	- Unlocked record[h]
			2	- Value in key already in the database and duplicates not specified when CREATK was run.[d]
			3	- Indices specified in the program do not match indices used when CREATK was run.
			5	- Index size does not size used on creation.
			6	- The disk is full.
			9	- System error, call analyst.

[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 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 file status is 92, a duplicate key 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 a 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 on 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 19-1. File Status Key Definitions, Indexed Sequential Files

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 16, 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 DECLARATIVE 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 19-1.

CLOSE STATEMENT

FORMAT:

CLOSE index-file-name.

GENERAL RULE:

This is the only option possible for an indexed file.

DELETE STATEMENT

FORMAT:

DELETE file-name [INVALID KEY imperative-statement]

GENERAL RULES:

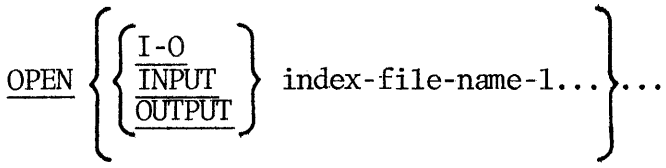
1. The DELETE statement logically removes a data record from the indexed file together with all the indices.

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 STATEMENT

FORMAT:



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 with lock record.

4. The table below specifies the types of OPEN statements which are permissible with the different ACCESS modes.

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

Table 19-2. OPEN Statements Vs. Access Mode, Indexed I-O

READ STATEMENT

FORMAT 1: (SEQUENTIAL or DYNAMIC)

READ file-name [NEXT RECORD] [INTO data-name-1]

[AT END imperative-statement . . . .].

FORMAT 2: (SEQUENTIAL, RANDOM or DYNAMIC)

READ filename [INTO data-name-1]

[KEY IS data-name-2]
[INVALID KEY imperative-statement].

GENERAL RULES:

1. Format 1, Option 1 (SEQUENTIAL ACCESS ONLY):

   READ file-name [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 2, Option 2 READ statement. Thereafter, the file can be read with a Format 1, Option 1 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 1, Option 2 (DYNAMIC and SEQUENTIAL ACCESS):

   READ file-name [NEXT RECORD] [INTO data-name-1]

   [AT END imperative-statement . . .].

   A. 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.

B. FOR SEQUENTIAL ACCESS:

The NEXT RECORD is not required with sequential access; it is automatically accessed.

3. Format 2, Option 1:

READ file-name [INTO data-name-1]

[INVALID KEY imperative-statement].

A. 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 19-1 are encountered.

B. 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 19-1). The INTO clause moves data to data-name-1.

4. Format 2, Option 2:

READ filename [INTO data-name-1]

[KEY IS data-name-2]
[INVALID KEY imperative-statement].

This format is used for sequential access, allowing the file to be retrieved sequentially based on the ALTERNATE RECORD KEYS (secondary indexes) via the KEY IS clause. Once this format is executed, the Format 1 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 STATEMENT

FORMAT:

REWRITE record-name [FROM data-name-1]

[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 successfully 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-1 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-1 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 DECLARATIVE. One or the other of these statements must be present, or the program will terminate if the invalid statement is activated. Refer to Table 19-1 for status codes.

START STATEMENT

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 1:

```
START file-name.
```

This option positions the file to the value contained in the RECORD KEY data-name. If that record is not present on the file, control is passed to the DECLARATIVE section if present; otherwise the program terminates.

3. Option 2:

```
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 on the file, control is passed to the DECLARATIVES - otherwise the program terminates.

4. Option 3:

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

[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. This option allows the keys to contain partial values.

The INVALID clause or DECLARATIVES is taken if there is no data satisfying data-name, and the STATUS code returned is 23 on a full key.

5. START does not retrieve a record, but only positions to a desired record.

EXAMPLE:

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'.
     Ø1  FILE-1-RECORD.
         Ø3  NAME    PIC X(10).
         Ø3  COMPANY PIC X(25).
```

A pictoral view of this file is presented below.

data-name	NAME	COMPANY
PICTURE	PIC X(10)	PIC X(25)
Values:	BLYE CLAPP GRIER HARPER KEANE	REPORTCO MERGANTHALER AUTOMATION DESIGNERS REPORTCO

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, a partial (or full) key is placed in the key field (NAME).

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 partial key 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 record.
READ FILE-1 NEXT RECORD.	This action will retrieve the desired record. In this example, it will be the record 'GRIER AUTOMATION'.
READ FILE-1 NEXT RECORD. . . .	This action will retrieve the next sequential record, 'HARPER DESIGNERS'.
READ FILE-1 NEXT RECORD.	This action will retrieve the next sequential record, 'KEANE REPORTCO'. Examination will indicate all desired records have been obtained.

WRITE STATEMENT

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 19-1 for error conditions.

SECTION 20

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 / RANDOM / DYNAMIC }]

RELATIVE KEY IS data-name-1

[FILE STATUS IS data-name-3]

GENERAL RULES:

1. SELECT file-name

This clause specifies the name of the relative file. Refer to Environment Division for rules.

2. ORGANIZATION IS RELATIVE

This specifies 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.

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

This clause specifies how a relative file is written or retrieved.

A. 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.

B. 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.

C. DYNAMIC:

When this access method is specified, the program can read or write randomly or sequentially.

4. RELATIVE KEY IS data-name-1

The RELATIVE KEY clause specifies the data item within Working-Storage which is used for the primary index.

A. Data-name-1 must not be defined in the Record Description.

B. Data-name-1 must not be specified with an OCCURS clause, or be contained within a group affected by an OCCURS clause.

C. Data-name-1 must not be specified with a P character in its PICTURE clause, or be described with a separator sign (/).

D. Data-name-1 must be a valid numeric integer, and cannot contain a value greater than 999,999.

E. The value contained within data-name-1 must be unique; duplicates are invalid.

The RELATIVE KEY is optional if access is sequential. In this case, no RELATIVE KEY need be specified. However, in the creation of the template, a RELATIVE KEY size equal to the maximum (48 bits), must be given.

5. [FILE STATUS IS data-name-3]

The FILE STATUS is a two-character (one word), unsigned field described in the Working-Storage section. After each access to the operating system, a status code is placed in this field. For a successful read or write, etc., the status code contains 00. If the INVALID KEY or DECLARATIVES were involved, the error status code is returned. The programmer then can determine what action to take. Refer to Table 20-1 for error codes.

FILE ORGANIZATION		STATUS KEY 1		STATUS KEY 2
Relative	Ø	- Successful completion	Ø	- No further information
	1	- End of file[a]	Ø	- 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]	Ø	- No further information
	9	- Implementor - defined	Ø	- Locked record[g]
			1	- Unlocked record[h]
			2	- Record already exits on Data Base
			6	- Space relative key contains larger value than used when CREATK was used.
			9	- System error, call analyst.

[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 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.

[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 a relative file: trying to write beyond the predefined boundaries of the file. For indexed file: trying to write a record containing a key which already exists on 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 20-1. File Status Key Definitions, Relative I-O

PROCEDURE DIVISION

The COBOL statements listed in this section apply to their application in RELATIVE file processing.

A complete description of all COBOL verbs, their functions, formats, and rules, is provided in Section 16, 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 DECLARATIVE 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 20-1).

CLOSE STATEMENT

FORMAT:

CLOSE index-file-name.

GENERAL RULE:

This is the only option possible for a Relative file.

DELETE STATEMENT

FORMAT:

DELETE file-name [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 STATEMENT

FORMAT:

```
OPEN { I-O    } index-file-name-1 . . . .
     { 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.

4. The table below specifies the types of OPEN statements which are permissible with the different ACCESS modes.

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

Table 20-2. OPEN Statements vs. Access Mode, Relative I-O.

READ STATEMENT

FORMAT 1 (SEQUENTIAL or DYNAMIC):

READ file-name [NEXT RECORD] [INTO data-name-1]

[AT END imperative-statement . . .].

FORMAT 2 (SEQUENTIAL, RANDOM or DYNAMIC):

READ filename [INTO data-name-1]

[INVALID KEY imperative-statement].

GENERAL RULES:

1. Format 1, Option 1 (SEQUENTIAL ONLY):

   READ file-name [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 . . .].

   A. 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 RELATIVE KEY data-name.

   B. FOR SEQUENTIAL ACCESS:

      The NEXT RECORD is not required with sequential access.

3. Format 2, Option 1:

READ filename [INTO data-name-1]

[INVALID KEY imperative-statement].

A. 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 20-1 are encountered.

B. FOR DYNAMIC and RANDOM ACCESS:

This format retrieves data based on the value contained in RELATIVE KEY or data-name. If the record is not found, or any other error status is encountered, control is passed to the INVALID KEY clause. Refer to Table 20-1.
The INTO clause moves data to data-name-1.

REWRITE STATEMENT

FORMAT:

REWRITE record-name [FROM data-name-1]

[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-1 option allows the record to be created in another area. It is equivalent to a MOVE data-name-1 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 DECLARATIVE. One or the other of these statements must be present. Refer to Table 20-1 for status codes.

START STATEMENT

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 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 RELATIVE KEY data-name. If that record is not present on the file, control is passed to the DECLARATIVE section if present; otherwise, the program terminates.

3. Option 2:

```
START file-name KEY IS data-name
```

This option will position the file to the value contained in data-name as defined in RELATIVE KEY. If the record is not contained on the file, control is passed to the DECLARATIVES, otherwise they will terminate.

4. Option 3:

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

[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. This option allows the keys to contain partial values.

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 STATEMENT

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 20-1 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-1.

UTILITIES

REFERENCE

SECTION 21

COMPILER REFERENCE INFORMATION

COBOL COMPILER PARAMETERS

Prime COBOL Compiler Mnemonics

Mnemonic parameters, which are the Prime-supplied default parameters (i.e., those which need not be included), are underlined. The system manager may have changed the defaults; if so, the programmer should obtain a list of the installation-specific defaults.

B[INARY] { treename / YES / NO }

Specifies the binary (object output file. If <treename> is given, that will be the name of the binary file. If YES is used, the name of the binary file will be B←PROGRAM (where PROGRAM is the source filename). If NO is used, then no binary file is created and it is a syntax check only. Omitting the parameter is equivalent to the inclusion of -BINARY YES. (See Table 21-1.)

EXP[LIST]

Prints an expanded listing (in addition to the source code listing) in the listing file. This parameter has no effect for compilations in 64R mode. No listing is generated unless an output device file is specified using LISTING (see NOEXPLIST).

I[NPUT] treename

Specifies the name of the input source program (see Table 22-1). This parameter must not be used if the source filename immediately follows the COBOL command; otherwise, it must be included in the parameter list.

L[ISTING] { treename / YES / NO / TTY / SPOOL }

Specifies the listing device/filename.

treename	- opens this file for the listing.
NO	- no listing file is created.
TTY	- the listing file is printed on the user terminal.
SPOOL	- the listing file is spooled directly to the line printer.

If this parameter is omitted from the parameter list, it is equivalent to the -LISTING YES parameter inclusion.

COMPILER MNEMONICS	INPUT	LISTING	BINARY
treename	looks for file named treename as source file	opens file named treename as listing file	opens file named treename as binary (object) file.
YES		uses default filename for listing file. L←PROGRM	uses default filename 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

Table 21-1. Compiler File Specifications

NOEXPLIST — Do not generate an expanded listing. This parameter is meaningful only for completion in 64V mode.

64R — Generates binary code suitable for loading with the Linking Loader. The user is given 64K words (128K bytes) of user memory. When loading the loader's MOVE command must be used to change load mode to 64R. (See 64V.)

64V — Generates binary code which must be loaded with the SEG loader. This must be used for generating shared procedures and/or programs requiring more than 128K bytes of user space; it provides a user area up to 1.9 (or 3.9) megabytes (15 or 31 segments of 128K bytes each). It may be run on any Prime 400 (or higher system) under PRIMOS IV or V. (See 64R.)

Explicit Setting of the A Register

The COBOL compiler is invoked by the COBOL command to PRIMOS

```
COBOL treename [1/A-register]
```

where treename is the treename of the COBOL source file, and A-register is the (octal) value of the A Register.

The default value of the A Register is:

```
'000777 (binary = 0000000111111111)

    Input file is on disk
    No expanded listing
    Listing file is on disk
    Binary file is on disk
    Compile in 64R mode
```

If the default values are used, the A-register parameter may be omitted.

Bit values corresponding to the mnemonic parameters are: (defaults are underlined).

MNEMONIC	BIT(S)	SET TO
B[INARY]	14, 15, 16	000,001,111 (see table 21-2)
EXP[LIST]	4	1
I[NPUT]	8, 9, 10	000,001,111 (see table 21-2)
L[ISTING]	11, 12, 13	000,001,111 (see table 21-2)
NOEXPLIST	4	0
64R	6	0
64V	6	1

Binary bit settings are converted to octal A-register values by:

1. Grouping bits by threes, starting from bit 16.
2. Converting each group to its octal value.

EXAMPLE:

Bit Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Binary Value	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
Octal Value	0	0			0			7			7			7		

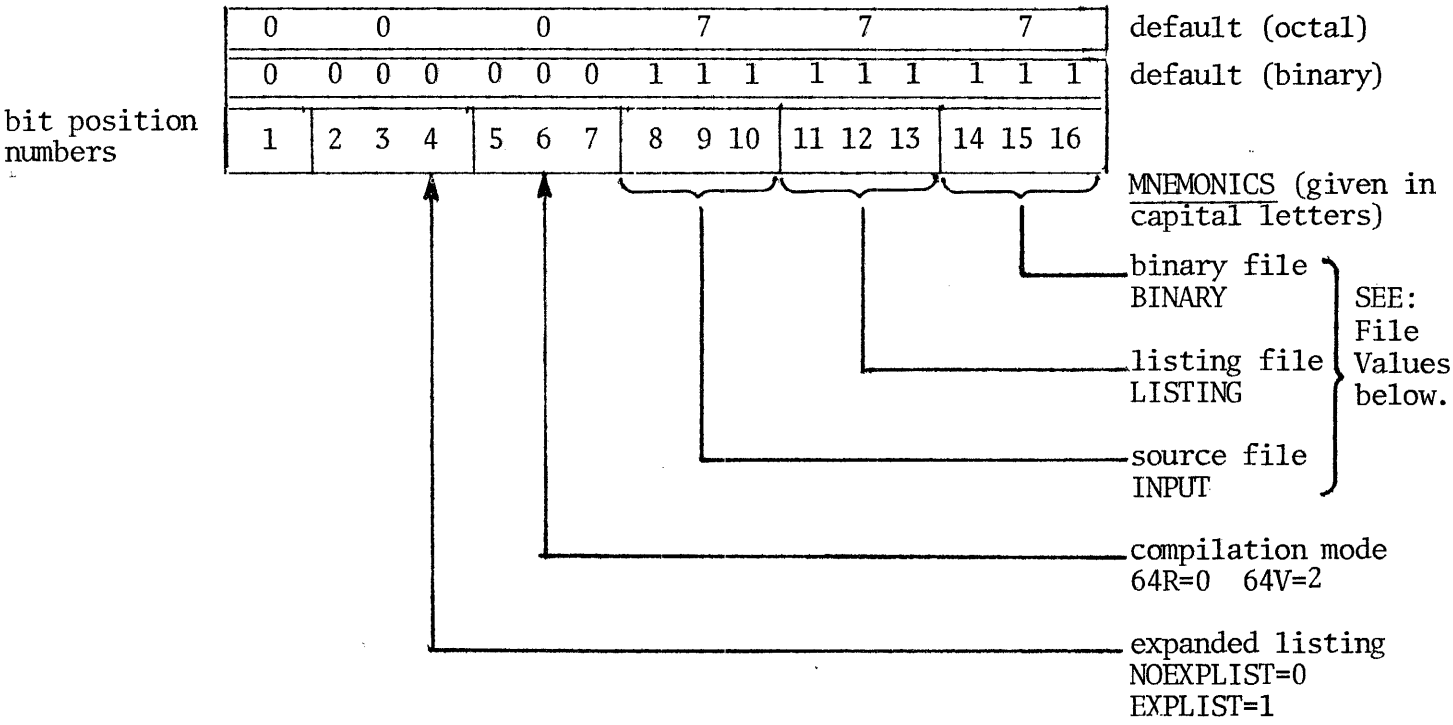
Figure 21-1. Bit Conversion, Binary/Octal

Bit specifications for input/output devices are given in the table below:

Bits	Octal	Device	Mnemonic Parameter
000	0	None	NO
001	1	User Terminal	TTY
111	7	Disk (PRIMOS file system)	

Table 21-2. Input/Output Device Bit Specification

NOTE: Other values (2-6) are reserved for future use. The default is 7.



FILE VALUES

Bits 8 9 10 / 11 12 13 / 14 15 16	Octal	Device/File	Mnemonic
0 0 0	0	None	NO
0 0 1	1	User Terminal	TTY
1 1 1	7	Disk	
Others	2-6	Reserved for future use	

Figure 21-2. Bit-Mnemonic Correspondence, A Register

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 in Table 21-3 below.

File Type	PRIMOS file unit
Source	1
Listing	2
Object	3

Table 21-3. PRIMOS File Units

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←file-name. The default convention for an object file is B←file-name.

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 treename, 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. Its format is:

```
LISTING filename-2
```

where filename-2 is the actual name under which the listing file will be stored.

The command LISTING SAMLST would open a listing file in the current UFD, on PRIMOS file unit 2, under the filename SAMLST instead of L←SAM. NOTE: In this instance, A-register bits 11-13 must be set to '7 or nothing will be written into the file.

File Manipulation

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 1/A-register
   .
   .
   .
COBOL Source-n 1/A-register
   .
   .
   .
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.

NOTE: The default value of bits 14-16 of the A-register is '7 - disk file system. If not using the default A-register values 14-16 to '7 or nothing will be written into the object file. Object files can also be concatenated in the same manner as listing files.

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

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

C[LOSE] { 2 / Filename-2 } { 3 / Filename-3 }

or

C[LOSE] ALL

SECTION 22

SEG REFERENCE

COMMAND SUMMARY

A complete list of SEG commands is given in this section in alphabetical order. Underlining shows the acceptable command abbreviations. Items in brackets ([ ]) are optional.

SEG Commands

ATTACH [ufd-name] [password] [ldisk] [key]

Attaches to another UFD.

ufd-name is the name of the UFD to be attached to; omission is home UFD.

password is password of UFD to be attached to if password-protected.

ldisk is logical disk on which MFD is to be searched for UFD specified.

'0 (or omitted)	search logical disk 0
'100000	search all logical disks
'177777	search logical disk on which current UFD is located

key is key for attach/set information.

0	attach to UFD; do not set home
1	attach to UFD; set home to new current file
2	attach to sub-UFD in current UFD; do not set home to new current UFD
3	attach to sub-UFD in current UFD; set home to new current UFD

A/SYMBOL sname [segtype] segno size

Places a symbol and reserves 0 or more locations in memory for it.

sname is the name of the symbol

segtype is the type of segment either DATA or PROCEDURE; if omitted, a data segment is assumed. If the segment does not yet exist, it will be created.

segno is the absolute octal segment number

size is the number of locations (octal) to be reserved for the symbol; if omitted 0 is assumed.

COMMON ABS segno

Specifies segment into which COMMON will be loaded.

segno is the absolute octal segment number into which COMMON will be loaded.

COMMON REL segno

Establishes a relative assignment number for segment(s) into which COMMON will be loaded.

segno is the segment number into which COMMON will be loaded; it is a small octal number.

DELETE [filename]

Deletes saved SEG runfile with name filename. If filename is omitted, the established runfile is deleted.

D/xx

Perform load operation with same numeric parameters as previous load command.

xx represents one of the load commands: LOAD, LIBRARY, RL, PL, IL.

D/ may be combined with P/ as either D/P/xx or P/D/xx

EXECUTE [1/a-reg] [2/b-reg] [3/x-reg]

First SAVEs the program with the register settings specified by the user, or the default values if the register setting is not specified. It then executes the program. After execution command is returned directly to PRIMOS. The default values are almost always used.

a-reg initial value of A register
b-reg initial value of B register
x-reg initial value of X register

F/xx [filename] [addr psegno lsegno]

F/S/xx [filename] [addr psegno lsegno]

Forceloads all routines in an object file.

xx — is one of the load commands LOAD, LIBRARY, RL, PL, or IL. filename is the object file to be forceloaded.

xx	filename
LOAD or RL	required
PL or IL	omitted
LIBRARY	optional (if omitted PFTNLB and IFTNLB forceloaded)

addr — is the starting address in psegno for the procedure part of the binary file. If 0 is specified, the current PBRK is used.

NOTES:

1. Simple forceload of object file.

   psegno — relative assignment number of segment into which procedure is to be loaded.

   lsegno — relative assignment number of segment into which link frames are to be loaded.

   If psegno and/or lsegno are 0, SEG's default segments are used.

2. Forceload of object file to specific segments.

   psegno — absolute octal number of segment into which procedure is to be loaded.

   lsegno — absolute octal number of segment into which link frame is to be loaded.

F/S/xx may be written S/F/xx

F/ may also be combined with D/ or P/ as D/F/xx (or F/D/xx) and P/F/xx (or F/P/xx).

HELP

Prints a list of the SEG commands at the user's terminal.

IL [addr psegno lsegno]

Loads the impure FORTRAN library IFTNLB. This form of the command is rarely used; loading to specific segments is more usual.

addr	is the starting address in psegno for the procedure part of the binary file. If 0 is specified, the current PBRK is used.
psegno	relative assignment number of segment into which procedure is to be loaded.
lsegno	relative assignment number of segment into which link is to be loaded.

If psegno and/or lsegno are 0, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

INITIALIZE [filename]

Initializes SEG's loader and restarts it.

filename	is name of SEG runfile to be initialized and/or opened. If omitted, the established runfile name is used.

LIBRARY [filename] [addr psegno lsegno]

Loads a library file from UFD=LIB.

filename	is the name of the library file to be loaded; if omitted, the FORTRAN library files PFTNLB and IFTNLB are loaded.
addr	is the starting address in psegno for the procedure part of the binary file. If 0 is specified, the current PBRK is used.
psegno	relative assignment number of segment into which procedure is to be loaded.
lsegno	relative assignment number of segment into which link frames are to be loaded.

If psegno and/or lsegno are 0, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

LOAD synonym for VLOAD
LOAD * synonym for VLOAD *

LOAD filename [addr psegno lsegno]

Loads a binary file.

filename	is the name of the binary file to be loaded.
addr	is the starting address in psegno for the procedure part of the binary file. If 0 is specified, the current PBRK is used.
psegno	relative assignment number of segment into which procedure is to be loaded.
lsegno	relative assignment number of segment into which link frames are to be loaded.

If psegno and/or lsegno are 0, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

MAP filename-1 [filename-2] map-option
MAP * [filename-2] map-option

Prints specified loadmap of SEG runfile to user's terminal or to a file.

filename-1	name of SEG runfile for which map is to be generated.
filename-2	name of file into which map is to be written. If omitted, map is printed at user's terminal.

map-option	type of loadmap to be generated
0 (or omitted)	Full map
1	Extent map only
2	Extent map and base areas
3	Undefined symbols
4	Full map (identical to 0)
5	System programmer's map
6	Undefined symbols, alphabetical order
7	Full map, sorted alphabetically

NOTES:

1. Used to get a loadmap of a runfile other than the established runfile.

2. Used to get a loadmap of the established runfile.

MAP [filename] map-option

Prints a loadmap of currently established runfile to user's terminal or to a file.

filename is name of file into which load map is to be written; if omitted, map is printed at user's terminal.

map-option type of load map to be generated. Map-options are the same as in SEG's MAP command.

MODIFY [filename]

Invokes the modification sub-processor.

filename is the name of the SEG runfile to be processed; if omitted, the established runfile is used.

NEW filename

Duplicates all portions of the established runfile resident above segment '4000, under the specified new name. The full map and all references to segments below '4000 are preserved.

filename is the name of the new SEG runfile which is to be created.

OPERATOR option

Allows the creators of specialized software to override basic restrictions in SEG's loader. Its use is dangerous unless the programmer is very careful. It is not considered to be useful for the applications programmer. The actual implementation of OPERATOR may change from revision to revision and it is not considered to be a supported function of SEG.

Option	Function
0	reinstate restrictions
1	relax restrictions

PATCH segno baddr taddr

Modifies the save range of an existing segment. Writes to the disk the portion of the runfile specified as patched. It may not be used with specifically addressed segments.

segno is absolute octal number of patched segment

baddr is lowest octal location of the patch

taddr is highest octal location of the patch

PL [addr psegno lsegno]

Loads the pure FORTRAN library PFTNLB. This form of the command is rarely used; loading to specific segments is more usual.

addr is the starting address in psegno for the procedure part of the binary file. If 0 is specified, the current PBRK is used.

psegno relative assignment number of segment into which procedure is to be loaded.

lsegno relative assignment number of segment into which link frames are to be loaded.

If psegno and/or lsegno are 0, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

PSD

Invokes the VPSD debugging utility.

P/xx [filename] option [psegno lsegno]

Loads an object file on a page boundary. A page boundary is an address of the form 'yy000 where yy is an even number.

xx is a load command: LOAD, LIBRARY, RL, PL, or IL.

filename is the object file to be loaded.

xx	filename
LOAD or RL	required
PL or IL	omitted
LIBRARY	optional (if omitted, PFTNLB and IFTNLB are loaded)

option determines what shall be loaded

PR	load only procedure on a page boundary
DA	load only link frames on a page boundary
(omitted)	load both procedure and link frames on a page boundary

psegno — absolute octal number of segment into which procedure will be loaded.

lsegno — absolute octal number of segment into which link frames will be loaded.

Default segments will be those of the current procedure and/or link frame pointers; if necessary SEG will create new segments. If either PR or DA is specified for option, loading in the non-specified segment begins at its current load point. Only the first routine in the file is placed on a page boundary.

P/ may be compounded with F/ to forceload on a page boundary F/P/xx or P/F/xx (see F/xx).

QUIT

Returns user to PRIMOS command level (in SEG).

QUIT

Returns user to PRIMOS command level. Does not SAVE runfile (in SEG's Loader).

RESTORE [filename]

Restores a SEG runfile to user memory.

filename — is the SEG runfile to be restored; if omitted, the established runfile is used.

RESUME [filename]

or

RESUME [filename]

Restores runfile to memory, if necessary, and then executes it.

filename — is the name of the SEG runfile; if omitted, the established runfile is used.

RETURN

Returns the user to the SEG command level. Unlike the RETURN command in the Modification sub-processor this command does not SAVE the runfile.

RETURN

Writes entire runfile to disk and then transfers control to the SEG command level (in SEG's Modification subprocessor).

RL filename [addr psegno lsegno]

Logically replaces a binary subprogram in the established runfile.

filename	is the name of the module to be replaced.
addr	is the starting address in the psegno for the procedure of the binary file. If 0 is specified, the current PBRK is used.
psegno	relative assignment number of segment into which procedure is to be loaded.
lsegno	relative assignment number of segment into which link frames are to be loaded.

If psegno and/or lsegno are 0, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

R/SYMBOL sname [segtype] segno size

Places a symbol and reserves 0 or more locations in memory for it.

sname	is the symbol name
segtype	is the type of segment, either DATA or PROCEDURE; if omitted, a data segment is assumed.
segno	is relative segment reference number. If 0, the first available segment of current type is used. If segment does not yet exist, a new segment will be created.
size	is number of locations to be reserved for the symbol; if omitted, 0 is assumed.

SAVE synonym for MODIFY

SAVE [1/a-reg] [2/b-reg] [3/x-reg]

SAVEs the result of the load by writing all buffers to the disk and setting the stack into the first available segment (unless the user has specified the stack with the loader's ST command). The user has the option of setting the initial register values, but this is rarely ever done.

a-reg	value of A register to be saved
b-reg	value of B register to be saved
x-reg	value of X register to be saved

SEG filename

SEG

SEG filename 1/1

SEG 1/1

Invokes the segmented-address runfile utility.

NOTES:

1. filename is the name of the SEG runfile to be executed. Loads the runfile into memory and starts execution.

2. Accesses the SEG commands to load, modify, and/or execute a SEG runfile.

3. filename is the name of the SEG runfile restored to memory prior to transfer of control to the VPSED debugging utility. Control may be returned to SEG by VPSD's, Q, or QU command and the program may then be executed.

4. Allows the currently existing memory image to be examined and/or modified with the VPSD debugging utility. Control may be returned to SEG by VPSD's, Q, or QU command but the resulting memory image cannot be executed at the SEG command level.

SHARE [filename]

Converts portions of the SEG runfile corresponding to segments below '4001 into runfiles resembling those for RMODE.

filename	is the name of the SEG runfile which is to be split out for sharing. If omitted, the established runfile will be used.

SEG responds to the SHARE command by asking for a two-character ID as:

TWO CHARACTER FILE ID:

A separate runfile is created for each segment below '4001; the filenames are the two-character ID followed by the (octal) segment number.

SINGLE [filename] segno

Creates a runfile for specified segment number resembling one for RMODE.

filename — is the name of the SEG runfile from which an RMODE runfile is to be split. If omitted, the established runfile is used.

segno — is the absolute octal number of the segment for which the RMODE runfile is to be created.

SEG responds to the SINGLE command by asking for a two-character ID as:

TWO CHARACTER FILE ID:

The RMODE runfile is created with a filename composed of the two-character ID followed by the (octal) segment number specified.

SK Ssize

SK Ssize 0 segno

NOTES:

1. Specifies stack size

   ssize — is minimum required stack size in octal words; if 0 is specified, the default value of '6000 is used. ssize = '177774 reserves an entire segment for the stack.

2. Specifies stack location

   segno — is absolute octal segment number for the stack.

   addr — is octal starting address for the stack in the specified segment. Addr must be at least 4; locations 0 to 3 must be reserved with R/SY.

3. Specifies stack size and segment for extension stack

   ssize — is minimum size of stack to be allocated.

   segno — is absolute octal number of first segment available for the extension stack.

4. Specifies primary stack location and segment for extension stack.

   ssegno — is absolute octal number of segment in which stack begins.

addr — is octal starting location of stack in starting segment.

segno — is absolute octal number of first segment available for extension stack.

In 3 and 4, the extension stack-frame begins in segno followed by segno+1, segno+2, etc., if needed.

At least '15 (12) words must be available in the starting stack segment.

SPLIT segno addr

SPLIT addr

SPLIT addr ssegno saddr esegno

Breaks a segment into procedure (lower) and data (upper) portions.

segno — is the absolute octal number of the segment to be split.

addr — is the octal location of the split in the segment. addr must be a multiple of '4000.

NOTES:

1. Splits segment as specified.
2. Splits segment '4000 and loads RMODE interlude program RUNIT starting at location '4000.
3. Splits segment '4000, loads RUNIT and supports extension stacks.

   addr — is address (octal) of split in segment '4000.

   ssegno — is absolute octal number of segment in which stack will begin.

   saddr — is address (octal) at which stack begins in segno.

   esegno — is absolute octal number of first segment available for stack extensions.

   At least '15 (12) words must be available in the starting stack segment.

STACK ssize

Sets the minimum stack size.

ssize — is the minimum required stack size (octal). ssize = '177774 forces use of an entire segment for the stack.

START segno addr

Sets a new address for start of execution.

segno — is the absolute octal segment number.

addr — is the new ECB address word (octal) in the specified segment for start of execution.

SYMBOL [sname] segno addr

Defines a symbol at a specific location in memory (actually an entry in the symbol table). SYMBOL may only be used to define a symbol before it is referenced. It cannot be used to define initialized COMMON or to satisfy unsatisfied references.

sname — is the symbol name.

segno — is the absolute octal segment number in which the symbol is to be located.

addr — is the octal address of the symbol in the specified segment.

S/xx [filename] addr psegno lsegno

Loads an object file to specified absolute segments.

xx — is a load command LOAD, LIBRARY, RL, PL, or IL.

filename — is the object file to be loaded.

xx	filename
LOAD or RL	required
PL or IL	omitted
LIBRARY	optional (if omitted, PFTNLB and IFTNLB are loaded)

addr — is the starting load address (octal in the procedure segment. If 0 is specified, loading starts at the current pointer position (PBRK).

psegno — is the absolute octal segment for loading procedure.

lsegno — is the absolute octal segment for loading the link frames.

If segments do not already exist, they will be created.

S/ may be combined with F/ as either S/F/xx or F/S/xx.

TIME [filename]

Prints at user's terminal, time of creation or last saved modification of the runfile.

filename	is the SEG runfile name; if omitted, the established runfile is used.

VLOAD [filename]

Accesses the SEG loader.

filename	name of SEG runfile; if omitted, established runfile is used. If filename is name of an existing SEG runfile, that runfile is initialized.

VLOAD * [filename]

Access the SEG Loader, preserving the contents of the specified runfile.

filename	is the name of the SEG runfile to be accessed; if omitted, the established runfile is used.

WRITE

Rewrites to the disks all segments of the established runfile above segment '4000.

If NEW is given before WRITE, the segments will be written into the new runfile; otherwise, the established runfile name will be used.

XPUNGE dsymbol dbase

Expunges some or all defined symbols from the symbol table.

dsymbol	Action
0	delete only entry points, leaving COMMON areas
1	delete all defined symbols - including COMMON area

dbase	Action
0	retain all base information
1	retain only sector zero information
2	delete all base area information

XP dsymbol	is equivalent to XP Dsymbol 0
XP	is equivalent to XP 0 0

APPENDIX A

PRIME COBOL SUMMARY

FEATURES

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

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

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

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

Module	Features Available in Prime COBOL
Nucleus	All of level 1, plus these features of level 2: Levels 77, 01-30, 88; Value series or range, level 88 conditions; AND OR = < > in conditions; Procedure-names consisting of digits only; COMPUTE with multiple receiving fields; PERFORM VARYING one index; Mnemonic-names for ACCEPT or DISPLAY devices; Qualification of Names (Procedure Division); Sign test; String; Unstring; ACCEPT { DAY / TIME / DATE } .
Sequential I-O	All of level 1 plus these features of level 2: RESERVE clause and variable form of BLOCK; Multiple operands in OPEN & CLOSE, with individual option per file.
Relative I-O	All of level 1 plus: RESERVE clause; DYNAMIC access mode (with READ next); START (with key relations EQUAL, GREATER, or NOT LESS).

Module	Features Available in Prime COBOL
Indexed I-O	All of level 1 plus: RESERVE clause; DYNAMIC access (with READ next); RANDOM access mode with READ by KEY; START (with key relations EQUAL, GREATER, NOT LESS).
Library	Level 1
Table Handling	All of level 1 plus: SEARCH
Inter-program Communication	Level 1

SYSTEM FILES

To utilize COBOL, the following files must be available on the system in the UFD's specified:

UFD	FILE-NAME
CMDNCO	COBOL
SYSOVL	C$$DAT C$$DAR C$$GEN C$$FIN C$$END C$$64V (*)
LIBRARY	COBLIB COBKID VCOBLB (*)

*Denotes new files for 64V mode.

VCOBLB

The new VCOBLB Library contains the following common COBOL subroutines.

C$ADAT = returns current date in format YMMDD

C$ADAY = returns Julian date in format YYDD

C$ATIM = returns current time in format HHMMSSFF

H = Hour
M = Minutes
S = Seconds
F = Hundreth of seconds

C$INSP = INSPECT statement

C$UNSI/C$UNS1 = STRING statement

C$STR1/C$STR2 = STRING statement

C$IN = File assignment initialization

C$OS = Open sequential file

C$CS = Close sequential file

C$RS = Read sequential file

C$XS = Rewrite sequential file

C$WS = Write sequential file

COI/COR = Open indexed/relative file

CSI/CCR = Close indexed/relative file

APPENDIX B

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 know in advance which index is to be used to locate a data entry.

APPENDIX C

CREATING ISAM AND RELATIVE FILES THE MIDAS TEMPLATE

THE ISAM FILE

To initiate an INDEXED file for COBOL, a program called CREATK must be run. This program creates a template for the ISAM file. The following is a summary of CREATK, modified to reflect COBOL terms. For more complete information, see Section 11 of this manual and PDR 3061 Reference Guide, Multiple Index Data Access System (MIDAS).

CREATK is a conversational program. A typical dialogue is as follows (all user responses are underlined):

Minimum Dialogue

User responses are underlined.

Prompt	Response	Remarks
OK,	CREATK	
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.
FILENAME? [volume name>ufd passwd ldisk>]	filename	Volume name>UFD: specifies the name of 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/RECORD KEY)

Prompt	Response	Remarks
KEY TYPE:	B	
KEY SIZE=:	B number	Number is the number of bits in the primary key. It is equal to 8 times the number of characters in the key; e.g., 2 characters in a key = 16 bits. The maximum size for an indexed file is 32 characters or 256 bits.
DATA SIZE=:	number	Number of words for a data record, where number equals the record length divided by 2. For COBOL programs, this includes the key size, and a remainder factor of 1 if it applies.

(SECONDARY INDEX/Alternate Record Keys - 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 DECLARATIVE section will be activated.
KEY TYPE:	B	
KEY SIZE=:	B number	Enter the number of bits in the key; use same formula as for primary index.

Prompt	Response	Remarks
USER DATA SIZE=:	Ø (CR)	No data may be entered for secondary keys. The response must be Ø, (CR), or Ø (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.

An actual example for sample program REF2 appears at the close of Section 16.

THE RELATIVE FILE

To initiate a relative file, a program called CREATK must be run. This program creats a template for the relative file. The following is a summary of a CREATK run used in creation of a relative file template. For more information, refer to Section 11 of this manual, and PDR 3061 Reference Guide, Multiple Index Data Access System (MIDAS).

CREATK is a conversational program. A typical dialogue is as follows (all user responses are underlined):

Minimum Dialogue

User responses are underlined.

Prompt	Response	Remarks
OK,	CREATK	
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.
FILE NAME? [volume name>ufd passwd ldisk>]	filename	Volume name>UFD: specifies the name of 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?	YES	For a new, relative file.
KEY TYPE:	B	
KEY SIZE=:	B number	Number is the number of bits in the relative key; i.e., characters in the key X 8. The maximum size is 6 characters, or 48 bits. In sequential mode with no key, size must be specified at maximum: 48.

Prompt	Response	Remarks
DATA SIZE=:	number	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.
NUMBER OF ENTRIES TO ALLOCATE?	number	number is the number of entries to allocate in the new KI/DA file. Entries are numbered 1-n inclusive; any references 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.

APPENDIX D

REFERENCE TABLES

PRIME COBOL VERBS INDEX

VERB	CATEGORY (Depending on Format	Special Application	PAGE
ACCEPT	I/O		16-7
ADD	Arithmetic or Conditional		16-9
ALTER	Procedure Branch		16-11
CALL	Procedure Branch	Interprogram Communication	16-12, 17-3
CLOSE	I/O	File Handling	16-14, 19-7, 20-6
COMPUTE	Arithmetic or Conditional		16-16
COPY	Compiler Directing		16-17
DELETE	I/O or Conditional	File Handling	16-19, 19-8, 20-7
DISPLAY	I/O		16-20
DIVIDE	Arithmetic or Conditional		16-21
ENTER	Compiler Directing	Interprogram Communication	16-23, 17-3
EXHIBIT	I/O	Debugging	16-24
EXIT	Procedure Branch		16-25
EXIT PROGRAM	Procedure Branch	Interprogram Communication	16-26, 17-3
GO TO	Procedure Branch		16-27
IF[a]	Conditional or Arithmetic		16-28
INSPECT	Data Movement		16-32
MOVE	Data Movement		16-34
MULTIPLY	Arithmetic or Conditional		16-36
OPEN	I/O	File Handling	16-37, 19-9, 20-8
PERFORM	Procedure Branch		16-39
READ	I/O or Conditional	File Handling	16-42, 19-10, 20-9
READY TRACE	TRACE MODE Directing	Debugging	16-44
RESET TRACE	TRACE MODE Directing	Debugging	16-45
REWRITE	I/O or Conditional	File Handling	16-46, 19-12, 20-11
SEARCH	Table Handling		16-48, 18-5
SET	Table Handling		16-52, 18-5
START	I/O or Conditional	File Handling	16-54, 19-13, 20-12
STOP	I/O or Ending		16-56
STRING	Data Movement		16-57
SUBTRACT	Arithmetic or Conditional		16-60
UNSTRING	Data Movement		16-62
USE	I/O Conditional	File Handling	16-67
WRITE	I/O or Conditional	File Handling	16-69, 19-16, 20-14

[a]IF is a verb in COBOL, although not a verb in the grammatical sense in English.

Table D-1. Prime COBOL Verb Index

FILE STATUS KEY DEFINITIONS

FILE ORGANIZATION		STATUS KEY 1		STATUS KEY 2
Sequential	Ø	- Successful completion	Ø	- No further information
	1	- End of file[a]	Ø	- No further information
	3	- Permanent I-O Error[b]	Ø	- No further information
			4	- Boundary violation[c]
Relative	Ø	- Successful completion	Ø	- No further information
	1	- End of file[a]	Ø	- 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]	Ø	- No further information
	9	- Implementor - defined	Ø	- 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	Ø	- Successful completion	Ø	- No further information
	1	- End of file[a]	Ø	- 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]	Ø	- No further information
	9	- Implementor - defined	Ø	- Locked record[g]
			1	- Unlocked record[h]
			2	- Value in key already in the database and duplicates not specified when CREATK was run.[d]
			3	- Indices specified in the program do not match indices used when CREATK was run.
			5	- Index size does not match size used on creation.
			6	- The disk is full
			9	- System error, call analyst

[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 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 file status is 92, a duplicate key 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 a 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 D-2. File Status Key Definitions

PERMISSIBLE INPUT/OUTPUT STATEMENTS

File Organization	File Access Mode	Procedure Statement	Statement		
			Input	Output	I-O
Sequential Indexed Relative	SEQUENTIAL	READ WRITE REWRITE START DELETE	X X 	 X 	X X X X
Indexed Relative	RANDOM	READ WRITE REWRITE START DELETE	X 	 X 	X X X X
Indexed Relative	DYNAMIC	READ WRITE REWRITE START DELETE	X 	 X 	X X X X X

Table D-3. Permissible Input/Output Statements-Open Statements and Access Modes.

PERMISSIBLE MOVES

SENDING \ RECEIVING	ALPHABETIC	BINARY	ALPHANUMERIC EDITED	NUMERIC	NUMERIC EDITED	ALPHANUMERIC
ALPHABETIC	X		X			X
BINARY		X		X	X	X (A)
ALPHANUMERIC EDITED	X		X (C)			X
NUMERIC		X		X	X	X (B)
NUMERIC EDITED			X (C)			X (C)
ALPHANUMERIC	X			X	X (D)	X

NOTES:

(A) If receiving operand length L is less than or equal to 18, target Picture 9(L) is assumed. Otherwise, the MOVE is disallowed.

(B) 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.

(C) The source is considered as a character string.

(D) If source length L is less than or equal to 18, source Picture 9(L) is assumed. Otherwise, the MOVE is disallowed.

Table D-4. Permissible Moves

APPENDIX E

ASCII CHARACTER SET

COLLATING SEQUENCE

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 characters are arranged in ascending value from top to bottom as in Table.

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

(Characters with no punched card codes are not supported for punched card entry.)

ASCII CHARACTER SET

ASCII Character	PRIME REPRESENTATION Hexadecimal	Octal	Punched Cards
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
$\overline{0}$ (-zero)	FD	375	11-0
DEL (high-value)	FF	377	

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 - Compiler 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.

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.

REPORT ITEM OR EDIT SYMBOLS - Used in report item picture clauses.	
. decimal point (insertion character)	Used to insert an actual decimal in the indicated position of a report item.
, comma (insertion character)	Used to insert a comma in the indicated position(s) of a report 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 a report 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.

REPORT ITEM OR EDIT SYMBOLS (continued . . .)

= equal (insertion character)	Used to insert an actual equal symbol in the indicated position of a report 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 a report 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 a report 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 a report 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.
Ø 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 a report item.
CR credit (fixed sign control character)	Used as a fixed sign control character in the low-order (right-most) position of a report 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 a report 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.

APPENDIX G

ERROR MESSAGES

TYPES OF ERROR MESSAGES

This Appendix contains the following categories of errors:

- COMPILE-TIME ERRORS
- COMPILE-TIME WARNING MESSAGES
- RMODE RUN-TIME ERROR MESSAGES
- VMODE RUN-TIME ERROR MESSAGES
- SEG 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 SEARCH, AT END is invalid.

'DATA DIVISION ASSUMED.'

DATA DIVISION omitted; correct and recompile.

'DELETE/START NOT VALID FOR THIS FILE.'

See Table 16-3 OPEN Statements and Access Modes. Correct coding, recompile.

'DISPLAY LIMITED TO 72 ON CONSOLE, 132 ON PRINTER.'

The file exceeds limitations. Correct and recompile.

'ERRONEOUS ASSIGNMENT.'

Device does not match file; correct and recompile.

COMPILE-TIME ERROR MESSAGES

'ERRONEOUS FILE-NAME.'

SELECT file-name does not match FD file-name.

'ERRONEOUS QUALIFICATION; LAST DECLARATION USED.'

Data-name not unique, needs qualification.

'ERRONEOUS SELECT-SENTENCE; RESUMES AT NEXT SELECT OR AREA-A.'

The flagged SELECT is ignored. Correct errors, recompile.

'ERRONEOUS SUBSCRIPTING; STATEMENT DELETED.'

Refer to rules governing subscripting, Section 12, and subscripting, OCCURS clause. Correct errors, recompile.

'EXCESSIVE OCCURS NESTING IS IGNORED.'

Restate, using a 'long-hand' form; recompile.

'FD-VALUE IGNORED SINCE LABELS OMITTED.'

Value of File-ID or owner ID specified with labels omitted. 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.

COMPILE-TIME ERROR MESSAGES

'FILE NOT SELECTED; ENTRY BYPASSED.'

FD entry has no corresponding SELECT statement. Correct and recompile.

'FILE SECTION ASSUMED.'

Correct and recompile.

'GROUP ITEM; PIC/VALUE/JUST/BLANK/SIGN/SYNC IGNORED.'

These clauses are not permitted at the group level. Delete and recompile.

'GROUP SIZE >32,767; SET TO 1.'

Group and/or record size exceeds maximum. Correct and recompile.

'ILLEGAL MOVE OR COMPARISON IS DELETED.'

Check IF and MOVE statements. Correct errors, recompile.

'IMPROPER OCCURS COUNT IGNORED.'

OCCURS is greater than 1024. Check rules for OCCURS clause; correct and recompile.

'IMPROPER REDEFINITION IGNORED.'

Check rules for REDEFINES clause. Correct errors; recompile.

COMPILE-TIME ERROR MESSAGES

'INCOMPLETE/TOO LONG STATEMENT DELETED.'

Check syntax; correct and recompile.

'INCONSISTENT READ USAGE.'

OPEN statement and USAGE do not agree.

'INCONSISTENT WRITE USAGE.'

OPEN statement and USAGE do not agree.

'INVALID BLOCKING IS IGNORED.'

BLOCK CONTAINS clause in error; 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.'

Correct coding and recompile.

'KEY MUST BE DECIMAL OR CHARACTER ITEM, MAX. 255 BYTES. STATEMENT DELETED.'

Key specification in error. Correct and recompile.

COMPILE-TIME ERROR MESSAGES

'LABEL RECORDS OMITTED ASSUMED FOR UNIT-RECORD FILE.'

Check LABEL clause vis a vis device.

'LABELS ASSUMED FOR DISK FILE.'

Check LABEL clause vis a vis device.

'LEVEL 01 ASSUMED.'

Check coding; correct and recompile.

'MISORDERED/REDUNDANT SECTION PROCESSED AS IS.'

Correct coding sequence; recompile.

'NAME OMITTED; ENTRY BYPASSED.'

Unrecognizable data-name/syntax error. Correct and recompile.

'NON-UNIQUE SUBSCRIPT; LAST DECLARATION USED.'

Non-unique data-name. Qualification is required; recompile.

'OCCURS DISALLOWED AT LEVEL 01.'

Delete error and recompile.

'PARAGRAPH DECLARATION REQUIRED HERE.'

Paragraph-name required; recompile.

COMPILE-TIME ERROR MESSAGES

'PERIOD ASSUMED AFTER PROCEDURE-NAME DEFINITION.'

Period missing after a paragraph-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.'

Correct discrepancy, recompile.

'REDUNDANT CLAUSE IGNORED.'

Remove and recompile.

'REDUNDANT FD.'

Multiple FD's. Delete and recompile.

'"SECTION" ASSUMED HERE.'

Insert SECTION and recompile.

'SINGLE-SPACING ASSUMED DUE TO IMPROPER ADVANCING COUNT.'

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.

COMPILE-TIME ERROR MESSAGES
'STATEMENT DELETED DUE TO ERRONEOUS 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.' Correct syntax and recompile.
'UNIT-RECORD FILE BLOCKING IS IGNORED.' Device and BLOCK clause are incompatible.
'UNRECOGNIZABLE ELEMENT IS IGNORED.' Correct and recompile.
'UNRESOLVED PROCEDURE-NAME; STATEMENT DELETED.' Correct and recompile.
'USING-LIST LEVELS MUST BE 01/77.' Correct and recompile.

COMPILE-TIME ERROR MESSAGES

'VALUE CLAUSE IGNORED.'

Delete and recompile.

'VALUE DELETED DUE TO TYPE CONFLICT.'

PICTURE and VALUE disagree. 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 ERROR MESSAGES, System Level

INCONSISTENT READ USAGE
INCONSISTENT WRITE USAGE

A file has been defined to have usage of READ, WRITE or both, but I/O statements in the program show differently. For example, a file opened for I/O with only READ statements present will generate one of these errors. Correct errors; recompile.

PRWFIL UNIT NOT OPEN

Several conditions may prompt this error:

1. UFD full condition.

2. Misspelled or missing division header.

3. Unrecognized division. This problem is related to the one above. A division is not being recognized because of some other error. For example:

   A. No period on last item in Working-Storage causes the Procedure Division to be unrecognized.

   B. Erroneous literal or continuations in the vicinity of a division will cause an item to be unrecognized.

Check to see that at least two temporary files will fit in the current UFD. Correct errors, recompile.

TBL-GROUP-ERROR

This error indicates an overflow of an internal table in the COBOL compiler. Possible causes:

1. An excessive number of literals in one paragraph. Separate the sentences into two paragraphs.

COMPILE-TIME ERROR MESSAGES, System Level

2. A SELECT clause does not match an FD statement. For example, the specified key does not exist in the Record Description.

3. An IF statement has an implied subject, implied relation, or parentheses. Correct and recompile.

COMPILE-TIME WARNING MESSAGES

COMPILE-TIME WARNING MESSAGES

'"COMP" IGNORED FOR 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.

'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'

The literal is too large as specified. Reduce its size or enlarge the item size; recompile.

'MOVE IS DONE WITHOUT CONVERSION.'

Data representation 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.

RMODE RUN-TIME ERROR MESSAGES

RMODE RUN-TIME ERROR MESSAGES

BASE REGISTER = 0

A program item referenced by a Base Register is finding the register clobbered or unset. Each 01 in the Linkage Section and each FD in the File Section will use Base Registers. Possible problems are:

1. A reference to data item located in a file description before that file is opened or after it is closed;

2. A reference to an item in the Linkage Section when that item was not present in the CALL statement;

3. A reference to a table entry with an out of range subscript resulting in a faulty Base Register setting for the next physical item.

4. An improper REDEFINES on item prior to an 01 with Base Register problems.

Determine and correct the error. Recompile.

GENERATED CODE

Incorrect source program coding is causing the compiler to generate faulty object code.

Determine and correct faulty coding. Recompile.

INPUT/OUTPUT ERROR LINE XXXX

This error is caused by one of the following I/O statements:

OPEN, READ, WRITE, REWRITE, DELETE, START.

xxxx refers to the program line number. If the error involves an OPEN statement, file assignments are incorrect. The program is attempting to open for reading a file which does not exist.

Determine and correct the error. Recompile.

RMODE RUN-TIME ERROR MESSAGES

NON-NUMERIC DATA ERROR LINE XXX

Possible causes include the figurative constant, SPACES, erroneous subscripting, incorrect redefinition of data areas, signed data in an unsigned field, etc.

PERFORM OVERFLOW

The program has encountered a nesting of PERFORM statements in excess of current capacity; the maximum depth is 24.

Rewrite the appropriate program sections. Recompile.

PERFORM OVERLAP

The program is performing a section of code which contains the end point of execution for another section of code. See the PERFORM statement in the COBOL REFERENCE SECTION.

Rewrite the appropriate program section. Recompile.

REDUNDANT OPEN

The program is attempting to OPEN a file which the program has currently open.

Remove the OPEN statement or insert a CLOSE. Recompile.

SUBSCRIPT FAULT

The user has attempted to reference a table item with a subscript value of zero or a negative number.

Correct the program. Recompile.

RMODE RUN-TIME ERROR MESSAGES

BAD SVC

This error is most often caused by an incorrect specification of parameters for system subroutine calls from the COBOL program. For example, a CALL to TIMDAT with incorrect parameters will produce this error. The incorrect parameters may be:

1. Item not in word boundary;

2. Use of external decimal in COBOL program when subroutine expects a single precision integer.

Correct the errors; recompile.

KIDA-generated messages

Error messages relating to MIDAS (KIDA) are described in a separate document; PDR 3061, MIDAS.

Consult MIDAS manual. Correct errors; recompile program if necessary.

VMODE RUN-TIME ERROR MESSAGES

The general format for run-time I-O errors generated by a 64V mode 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 discussion of MIDAS error messages, refer to PDR3061 Reference Guide, Multiple Index Direct Access System. 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.

VMODE RUN-TIME ERROR MESSAGES

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.

VMODE RUN-TIME ERROR MESSAGES

ERROR PROCESSING START STATEMENT

An unexpected error occurred 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.

TO 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.

VMODE RUN-TIME ERROR MESSAGES

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.

SEG LOADER ERROR MESSAGES

SEG LOADER ERROR MESSAGES

BAD OBJECT FILE

(VLOAD) User is attempting to load file which has faulty code. The file may not be an object file or it may be incorrectly compiled. FATAL, the load must be aborted

CAN'T LOAD IN SECTORED MODE

(VLOAD) The Loader is attempting to load code in sectored mode which has not been compiled in sectored mode. This could arise if trying to load a module compiled or assembled in 16S or 32S mode. It is unlikely the average applications programmer will encounter this. FATAL, abort load.

CAN'T LOAD IN 64V OR 64R MODE

(VLOAD) The Loader is attempting to load code in 64V mode which is not compiled in that mode. This would arise if:

1. A program was compiled in a mode other than 64V.

2. A PMA module is written in code other than 64V and its mode is not specified.

In case 1, the user should recompile the program. In case 2, which the average applications programmer is unlikely to encounter, the PMA module must be modified. FATAL, abort load.

COMMAND ERROR

(SEG) An unrecognized command was entered or the filenames/parameters following the command are incorrect. Usually not fatal.

EXTERNAL MEMORY REFERENCE TO ILLEGAL SEGMENT

(VLOAD) An attempt was made to load a 64R mode program, causing a reference to an illegal segment number. Recompile in 64V mode. FATAL, abort load.

SEG LOADER ERROR MESSAGES

ILLEGAL SPLIT ADDRESS

(VLOAD) Incorrect use of the Loader's SPLIT command. Segments may be split only at '4000 boundaries only (i.e., '4000, '10000, '14000, etc.). Not FATAL; resplit segment.

MEMORY REFERENCE TO COMMON IN ILLEGAL SEGMENT

(VLOAD) An attempt was made to load a 64R mode program wherein COMMON would be allocated to an illegal segment number. Recompile in 64V mode. FATAL, abort load.

NO FREE SEGMENTS TO ASSIGN

(VLOAD) All SEG's segments have been allocated; no more are available at present. Use SYMBOL command to eliminate COMMON from assigned segments, thus reducing the number of assigned segments required. (User may need larger version of SEG and PRIMOS.) Fatal, abort load.

NO ROOM IN SYMBOL TABLE

(VLOAD) Unlikely to occur; no user solution. A new issue of SEG with a bigger symbol table is required; check with analyst. As a temporary measure, user may try to reduce number of symbols used in program. FATAL, abort load.

REFERENCE TO UNDEFINED SEGMENT

(VLOAD) Almost always caused by improper use of the SYMBOL command to allocate initialized COMMON. Initialized COMMON cannot be located with the SYMBOL command; use R/SYMBOL or A/SYMBOL instead.

SEG LOADER ERROR MESSAGES

SECTOR ZERO BASE AREA FULL

(VLOAD) Extremely unlikely to occur. Not correctable at applications level. Check with analyst. FATAL, abort load.

SEGMENT WRAP AROUND TO ZERO

(VLOAD) An attempt has been made to load 64R mode program. The program has exceeded 64K and is trying to be loaded over code previously loaded. Recompile in 64V mode. FATAL, abort load.

APPENDIX H

RESERVED WORDS

ACCEPT	DATA	HIGH-VALUES	NEXT
ACCESS	DATE	I-O	NOT
ADD	DATE-COMPILED	I-O-CONTROL	NUMBER
ADVANCING	DATE-WRITTEN	ID *	NUMERIC
AFTER	DAY	IDENTIFICATION	OBJECT-COMPUTER
ALL	DECIMAL-POINT	IF	OCCURS
ALPHABETIC	DECLARATIVES	IN	OF
ALTER	DELETE	INDEX	OFF
ALTERNATE	DELIMITED	INDEXED	OFFLINE-PRINT *
AND	DELIMITER	INITIAL	OMITTED
ARE	DEPENDING	INPUT	ON
AREA	DISPLAY	INPUT-OUTPUT	OPEN
AREAS	DIVIDE	INSPECT	OR
ASCII *	DIVISION	INSTALLATION	ORDS
ASSEMBLER*	DOWN	INTO	ORGANIZATION
ASSIGN	DUPLICATES	INVALID	OUTPUT
AT	DYNAMIC	IS	OWNER *
AUTHOR	ELSE	JUST	PAGE
BEFORE	END	JUSTIFIED	PERFORM
BLANK	ENTER	KEY	PFMS *
BLOCK	ENVIRONMENT	LABEL	PIC
BY	EQUAL	LEADING	PICTURE
CALL	ERROR	LEFT	POINTER
CHARACTER	EVERY	LENGTH	POSITION
CHARACTERS	EXCEPTION	LESS	POSITIVE
CLOSE	EXHIBIT *	LIFE-CYCLE *	
COBOL	EXIT	LINE	PRINTER *
CODE-SET	EXTEND	LINES	PROCEDURE
COMMA	FD	LINKAGE	PROCEDURES
COMP *	FILE	LOCK	PROCEED
COMP-3	FILE-CONTROL	LOW-VALUE	PROGRAM
COMPUTATIONAL	FILE-ID *	LOW-VALUES	PROGRAM-ID
COMPUTATIONAL-3 *	FILLER	MODE	PUNCH *
COMPUTE	FIRST	MOVE	QUOTE
CONFIGURATION	FOR	MT7 *	QUOTES
CONSOLE *	FROM	MT9 *	RANDOM
CONTAINS	GIVING	MULTIPLY	READ
COPY	GO	NAMED *	READER *
COUNT	GREATER	NATIVE	READY *
CURRENCY	HIGH-VALUE	NEGATIVE	RECORD

* Prime reserved words

RESERVED WORDS

RECORDS	SUBTRACT
REDEFINES	SYNC
REEL	SYNCHRONIZED
REFERENCES	TABLE
RELATIVE	TALLYING
REMARKS *	TAPE
REMOVAL	TERMINAL
REPLACING	THAN
RERUN	THEN *
RESERVE	THROUGH
RESET	THRU
RESTART-FILE *	TIME
REVERSED	TIMES
REWIND	TO
REWRITE	TRACE *
RIGHT	TRAILING
ROUNDED	UNCOMPRESSED *
RUN	UNIT
SAME	UNSTRING
SEARCH	UNTIL
SECTION	UP
SECURITY	UPON
SELECT	USAGE
SENTENCE	USE
SEPARATE	USING
SEQUENTIAL	VALUE
SET	VALUES
SIGN	VARYING
SIZE	WHEN
SOURCE-COMPUTER	WITH
SPACE	WORKING-STORAGE
SPACES	WRITE
SPECIAL-NAMES	ZERO
STANDARD	ZEROES
START	ZEROS
STATUS	
STOP	
STRING	

* Prime reserved words

APPENDIX I

CONVERSION TABLES

HEXADECIMAL AND DECIMAL CONVERSION

	XX HEX	XX DEC	XX HEX	XX DEC	XX HEX	XX DEC	XX HEX	XX DEC
65,536	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
	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	

OCTAL AND DECIMAL CONVERSION

	OCT	XXX DEC	OCT	XXX DEC	OCT	XXX DEC	OCT	XXX DEC	OCT	XXX DEC
32768	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
	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	

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 Hex.

APPENDIX J

EXPANDED LISTING FOR V MODE

V-MODE

In 64V mode (Prime 400 and Prime 500 units), 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. For 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 X$HHHH

where: X is the label type (see below)
HHHH is a hexadecimal identifier.

LABEL TYPES

A - Paragraph or section entry point
B - Alter or Perform indirect word
C - Iteration variable
D - Decimal constant
E - Picture string
F - Character string
G - Generated label for program flow control
H - Passed parameter
S - Generated label - any usage allowed
Y - FCB - See VCOBLB listing for FCB formats
Z - File buffer

Other labels used are:

SB% - Stack base relative - used for temporary storage.
XB% - Temporary base relative - used for LINKAGE SECTION address.
WRKST$ - Working-Storage
WSECT$ - Working-Storage extension - used for items that are statically allocated but not explicitly in working storage. For example, the declaration 'indexed by data-name', would place 'data-name' in WSEXT$.

EXAMPLE:

```
003233:      001310      EAFA  1, Z$0027+72C
003234:  001000.000725L
```

The example above says: At Relative Location'3233 in the procedure area EAFA 1, File Buffer (ID=X0027) + 72 character offset. Note that the word offset is '725 in the link frame.

An expanded listing file example is presented on the next several pages. It represents a portion of an actual listing for sample program REF2 presented earlier.

For additional information pertaining to expanded code, and the Program Statistics page which follows it, the user is referred to Section 5 COMPILING A SOURCE PROGRAM, and the PMA User Guide, PDR 3059.

```
(0378)          READ-3.
(0379)                  MOVE LOW-VALUES TO STATE.
(0380)                  START DIRECTORY-FILE KEY IS NOT LESS THAN STATE.
(0381)                  GO TO READ-FILE-GO.
(0382)          READ-4.
(0383)                  MOVE ZEROS TO BIRTHD.
(0384)                  START DIRECTORY-FILE KEY IS NOT LESS THAN BIRTHD.
(0385)                  GO TO READ-FILE-GO.
(0386)          READ-5.
(0387)                  MOVE LOW-VALUES TO FIRST-NAME.
(0388)                  START DIRECTORY-FILE KEY IS NOT LESS THAN FIRST-NAME.
(0389)          READ-FILE-GO.
(0390)                  READ DIRECTORY-FILE NEXT RECORD
(0391)                      AT END MOVE ZEROS TO PERFORM-COUNT
(0392)                      GO TO READ-FILE-EXIT.
(0393)                  DISPLAY DISPLAY-DIR.
(0394)          READ-FILE-EXIT.
(0395)                  EXIT.
(0396)         *
(0397)         *
(0398)         *
(0399)         *
(0400)          WRAPUP.
(0401)                  PERFORM LIST-DIR.
(0402)                  MOVE 'END OF INDEXED TEST TO CHANGE FILE' TO PRINT-LINE.
(0403)                  DISPLAY 'END OF INDEXED TEST'.
(0404)                  CLOSE LIST-FILE, DIRECTORY-FILE.
(0405)                  STOP RUN.
(0406)         *
(0407)         *
(0408)         *
(0409)          FORMAT-INPUT.
(0410)                  MOVE SPACES TO WS-RECORD.
(0411)                  DISPLAY 'ENTER LAST NAME'.
(0412)                  ACCEPT WS-LAST-NAME.
(0413)                  DISPLAY 'ENTER FIRST NAME'.
(0414)                  ACCEPT WS-FIRST-NAME.
(0415)                  DISPLAY 'ENTER ADDRESS '.
(0416)                  ACCEPT WS-ADDRESS.
(0417)                  DISPLAY 'ENTER CITY '.
(0418)                  ACCEPT WS-CITY.
(0419)                  DISPLAY 'ENTER PHONE NUMBER '.
(0420)                  ACCEPT WS-PHONE-NO.
(0421)                  DISPLAY 'ENTER STATE XX'.
(0422)                  ACCEPT WS-STATE.
(0423)                  DISPLAY 'ENTER BIRTHDAY MMDDYY'.
(0424)                  ACCEPT WS-BIRTHD.
E X P A N D E D   L I S T I N G   F O R   --   REF2
       000000:      001300          EAFA  0,WRKST$+6C
       000001: 001000.000427L
       000003:      001320          STFA  0,Y$0027+20C
       000004: 001000.000564L
       000006:      001300          EAFA  0,Z$0027+8C
       000007: 001000.000664L
```

```
000011:       001320          STFA  0,Y$0027+88C
000012: 001000.000626L
000014:       001300          EAFA  0,Z$0027+72C
000015: 001000.000724L
000017:       001320          STFA  0,Y$0027+96C
000020: 001000.000632L
000022:       001300          EAFA  0,Z$0027+74C
000023: 001000.000725L
000025:       001320          STFA  0,Y$0027+104C
000026: 001000.000636L
000030:       001300          EAFA  0,Z$0027+23C
000031: 101000.000673L
000033:       001320          STFA  0,Y$0027+112C
000034: 001000.000642L
000036: 061432.000376L          PCL   =C$IN  ,*
                        A$0001  EQU   *
*   SRC LINE 112
000040:       001300    A$0002  EAFA  0,F$81F1
000041: 000000.000000F
000043:       001310          EAFA  1,SB
000044: 000400.000012S
000046: 001313.000032A          LFLI  1,26
000050:       001115          ZMVD
000051: 061432.000374L          PCL   =TNOU  ,*
000053: 000500.000012S          AP    SB
000055: 000300.000000F          AP    ='32,SL
*   SRC LINE 113
000057:       001300          EAFA  0,F$8209
000060: 000000.000000F
000062:       001310          EAFA  1,SB
000063: 000400.000012S
000065: 001313.000032A          LFLI  1,26
000067:       001115          ZMVD
000070: 061432.000374L          PCL   =TNOU  ,*
000072: 000500.000012S          AP    SB
000074: 000300.000056F          AP    ='32,SL
*   SRC LINE 114
000076: 061432.000372L          PCL   =I$AA12,*
000100: 000100.000000F          AP    ='0,S
000102: 000500.000012S          AP    SB
000104: 000300.000000F          AP    ='1,SL
000106:       001300          EAFA  0,SB
000107: 000400.000012S
000111:       001310          EAFA  1,WRKST$+2C
000112: 001000.000425L
000114: 001313.000001A          LFLI  1,1
000116:       001115          ZMVD
*   SRC LINE 115
000117:       001300          EAFA  0,WRKST$+2C
000120: 001000.000425L
000122: 001303.000001A          LFLI  0,1
000124:       001310          EAFA  1,F$8224
000125: 000000.000000F
000127: 001313.000001A          LFLI  1,1
```

```
        000131:       001117          ZCM
        000132: 141603.000000F        BCNE  G$0014
*   SRC LINE 116
        000134: 061432.000370L        PCL   =C$OS  ,*
        000136: 001100.002262L        AP    Y$0001,S
        000140: 000100.000000F        AP    ='2,S
        000142: 000300.000000F        AP    F$8230,SL
*   SRC LINE 117
        000144:     01.000000F        JMP   A$001F
                                G$0014 EQU  *
*   SRC LINE 119
        000145:     02.000000F  A$000A LDA  ='240
        000146:       001310          EAFA  1,WRKST$+14C
        000147: 001000.000433L
        000151: 001313.000120A        LFLI  1,80
        000153:       001116          ZFIL
*   SRC LINE 120
        000154: 061432.000370L        PCL   =C$OS  ,*
        000156: 001100.002104L        AP    Y$0011,S
        000160: 000100.000105F        AP    ='1,S
        000162: 000300.000143F        AP    F$8230,SL
*   SRC LINE 120
        000164: 061432.000370L        PCL   =C$OS  ,*
        000166: 001100.002262L        AP    Y$0001,S
        000170: 000100.000141F        AP    ='2,S
        000172: 000300.000163F        AP    F$8230,SL
*   SRC LINE 121
        000174:     02.000101F        LDA   ='0
*   SRC LINE 121
        000175:     04.000620L        STA   Y$0027+76C
        000176: 061432.000366L        PCL   =C$OI  ,*
        000200: 001100.000552L        AP    Y$0027,S
        000202: 000300.000171F        AP    ='2,SL
*   SRC LINE 122
        000204:       001300          EAFA  0,WRKST$+94C
        000205: 001000.000503L
        000207:       001310          EAFA  1,Z$0001
        000210: 001000.002370L
        000212: 001313.000144A        LFLI  1,100
        000214: 001303.000111A        LFLI  0,73
        000216:       001114          ZMV
*   SRC LINE 122
        000217: 061432.000364L        PCL   =C$WS  ,*
        000221: 001100.002262L        AP    Y$0001,S
        000223: 000100.000000F        AP    S$0000,S
        000225: 000100.000000F        AP    ='62,S
        000227: 000300.000000F        AP    ='40077,SL
                                S$0000 EQU  *
*   SRC LINE 124
        000231: 061432.000362L  A$0011 PCL  =C$RS  ,*
        000233: 001100.002104L        AP    Y$0011,S
        000235: 000100.000000F        AP    S$0002,S
        000237: 000300.000000F        AP    S$0001,SL
        000241:     01.000000F  S$0001 JMP  G$0015
```

```
*   SRC LINE 124
        000242:     01.000000F  S$0002 JMP   A$0017
*   SRC LINE 125
        000243:       001300    G$0015 EAFA  0,Z$0011
        000244: 001000.002212L
        000246:       001310           EAFA  1,Z$0001
        000247: 001000.002370L
        000251: 001313.000144A         LFLI  1,100
        000253: 001303.000120A         LFLI  0,80
        000255:       001114           ZMV
*   SRC LINE 126
        000256: 061432.000364L         PCL   =C$WS  ,*
        000260: 001100.002262L         AP    Y$0001,S
        000262: 000100.000000F         AP    S$0003,S
        000264: 000100.000226F         AP    ='62,S
        000266: 000300.000174F         AP    ='0,SL
*   SRC LINE 127
        000270:     02.000145F  S$0003 LDA   ='240
        000271:       001310           EAFA  1,Z$0027
        000272: 001000.000660L
        000274: 001313.001224A         LFLI  1,660
        000276:       001116           ZFIL
*   SRC LINE 128
        000277:       001300           EAFA  0,Z$0011
        000300: 001000.002212L
        000302:       001310           EAFA  1,Z$0027+8C
        000303: 001000.000664L
        000305: 001313.000100A         LFLI  1,64
        000307:       001115           ZMVD
*   SRC LINE 129
        000310:       001300           EAFA  0,Z$0011+64C
        000311: 001000.002252L
        000313:       001310           EAFA  1,Z$0027
        000314: 001000.000660L
        000316: 001313.000010A         LFLI  1,8
        000320:       001115           ZMVD
*   SRC LINE 130
        000321:       001300           EAFA  0,Z$0011+72C
        000322: 001000.002256L
        000324:       001310           EAFA  1,Z$0027+72C
        000325: 001000.000724L
        000327: 001313.000010A         LFLI  1,8
        000331:       001115           ZMVD
*   SRC LINE 131
        000332: 061432.000360L         PCL   =C$WI  ,*
        000334: 001100.000552L         AP    Y$0027,S
        000336: 000100.000000F         AP    S$0004,S
        000340: 000300.000000F         AP    G$0016,SL
        000342:     01.000341F         JMP   G$0016
*   SRC LINE 132
        000343:       001300    S$0004 EAFA  0,WRKST$+6C
        000344: 001000.000427L
        000346:       001310           EAFA  1,SB
        000347: 000400.000012S
```

```
        000351: 001313.000002A          LFLI  1,2
        000353:       001115            ZMVD
        000354: 061432.000374L          PCL   =TNOU  ,*
        000356: 000500.000012S          AP    SB
        000360: 000300.000203F          AP    ='2,SL
*   SRC LINE 133
        000362:     01.000231   G$0016 JMP   A$0011
*   SRC LINE 135
        000363: 061432.000356L  A$0017 PCL   =C$CS  ,*
        000365: 001300.002104L         AP    Y$0011,SL
*   SRC LINE 135
        000367: 061432.000354L         PCL   =C$CI  ,*
        000371: 001300.000552L         AP    Y$0027,SL
*   SRC LINE 136
        000373:       001300           EAFA  0,F$825D
        000374: 000000.000000F
        000376:       001310           EAFA  1,Z$0001
        000377: 001000.002370L
        000401: 001313.000144A         LFLI  1,100
        000403: 001303.000022A         LFLI  0,18
        000405:       001114           ZMV
*   SRC LINE 137
        000406: 061432.000364L         PCL   =C$WS  ,*
        000410: 001100.002262L         AP    Y$0001,S
        000412: 000100.000000F         AP    S$0005,S
        000414: 000100.000265F         AP    ='62,S
        000416: 000300.000000F         AP    ='3,SL
                                S$0005 EQU   *
*   SRC LINE 139
        000420:     02.000270F  A$001F LDA   ='240
        000421:       001310           EAFA  1,Z$0001
        000422: 001000.002370L
        000424: 001313.000144A         LFLI  1,100
        000426:       001116           ZFIL
*   SRC LINE 140
        000427:       001300           EAFA  0,F$826C
        000430: 000000.000000F
        000432:       001310           EAFA  1,SB
        000433: 000400.000012S
        000435: 001313.000014A         LFLI  1,12
        000437:       001115           ZMVD
        000440: 061432.000374L         PCL   =TNOU  ,*
        000442: 000500.000012S         AP    SB
        000444: 000300.000000F         AP    ='14,SL
*   SRC LINE 141
        000446:     02.000267F         LDA   ='0
*   SRC LINE 141
        000447:     04.000620L         STA   Y$0027+76C
        000450: 061432.000366L         PCL   =C$OI  ,*
        000452: 001100.000552L         AP    Y$0027,S
        000454: 000300.000417F         AP    ='3,SL
*   SRC LINE 142
        000456:       001300           EAFA  0,WRKST$+2C
        000457: 001000.000425L
```

```
        002270>       000007          OCT   7
        002271>       000000          OCT   0
        002272>                       IP    Z$0001
        002274>       100000          OCT   100000
        002275>       000000          OCT   0
        002276>       000000          OCT   0
        002277>       000062          OCT   62
        002300>       000062          OCT   62
        002301>       000000          OCT   0
        002302>       000000          OCT   0
        002303>       000000          OCT   0
        002304>       000003          OCT   3
        002305>       000000          OCT   0
        002306>       000000          OCT   0
        002307>       000000          OCT   0
        002370>                 Z$0001 DATA  50('  ')
_ P R O G R A M   S T A T I S T I C S

 EXECUTABLE CODE SIZE: 2588 WORDS.
 CONSTANT POOL SIZE: 511 WORDS.
 TOTAL PURE PROCEDURE SIZE: 3099 WORDS.

 WORKING-STORAGE SIZE: 168 BYTES.
 TOTAL LINKFRAME SIZE: 1096 WORDS.

 STACK SIZE: 102 WORDS.

 TRACE MODE: OFF.

 NO ARGUMENTS EXPECTED.

 424 SOURCE LINES.


 0000 ERRORS  0000 WARNINGS,  P400/500 COBOL REV 14.0 <REF2>
```

APPENDIX K

REVISION 15 COBOL

This appendix is composed of pages 1-17 of PTU48. The remainder of the PTU: corrections, typographic errors, etc. have been incorporated into the guide itself.

System Files

To utilize COBOL, the following files must be available in the UFD SYSOVL:

```
C$$COB  (*)
C$$ENV  (*)
C$$DAT
C$$DAR
C$$GEN
C$$FIN
C$$END
C$$64V
```

(*) Denotes new files at Rev. 15.

NOTE: Rev. 14 and earlier COBOL libraries are incompatible with the Rev. 15 compiler.

Streamlined Compiler

The Rev. 15 COBOL compiler is roughly twice as fast as older compilers. In addition, the working set size has been significantly reduced. This should improve compilation speed considerably on small memory systems.

At Rev. 15, the compiler will flag milestones during compilation: Phases I through VI.

```
OK, COBOL  -INPUT READ.IF  -64V
GO
 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

0000 ERRORS  0000 WARNINGS,  P400/500 COBOL REV 15.00 <TEST>
```

Messages relating to unsuccessful compilations are provided in one or both of the following forms:

1. On the terminal, after Phase VI:

```
 PHASE VI

UNSUCCESSFUL COMPILATION; TERMINAL ERROR ON LINE  20 .

0002 ERRORS  0000 WARNINGS,  P400/500 COBOL REV 15.00 <TEST>

 COMPILATION ABORTED (COBOL)

ER!
```

NOTE: PRIMOS will return with the error prompt (ER!) rather than OK. The user should never attempt to LOAD or SEG the output of an ABORTED or UNSUCCESSFUL compilation.

In such instances, the listing file will confirm the unsuccessful compilation and provide additional diagnostics.

Example:

```
REV 15.0  COBOL     SOURCE FILE:  READ.IF                    04/05/78


REV 15.0  COBOL     SOURCE FILE:  READ.IF                    04/05/78
(0001)   XXXXXX IDENTIFICATION DIVISION.
(0002)   XXXXXX PROGRAM-ID.
(0003)   XXXXXX     DRWTST.
(0004)          ENVIRONMENT DIVISION.
(0005)          INPUT-OUTPUT SECTION.
(0006)          FILE-CONTROL.
(0007)               SELECT IN-FILE ASSIGN TO PFMS.
(0008)   XXXXXX DATA DIVISION.
(0009)          FILE SECTION.
(0010)          FD  IN-FILE
(0011)              LABEL RECORDS ARE STANDARD
(0012)              VALUE OF FILE-ID IS 'INPUT'.
(0013)          01  IN-REC1 PIC X(40).
(0014)   XXXXXX WORKING-STORAGE SECTION.
(0015)          77  ACT-VALUE     PIC X VALUE SPACE.
(0016)          PROCEDURE DIVISION.
(0017)          OPEN-FILES.
(0018)              OPEN INPUT IN-FILE.
(0019)              IF  ACT-VALUE = '1'
(0020)              READ IN-FILE AT END GO TO END-PAR.
(0021)              GO TO OPEN-FILES.
(0022)          END-PAR.
(0023)              CLOSE IN-FILE.
(0024)              STOP RUN.
```

```
0020  CONDITIONAL I/O STATEMENT DISALLOWED WITHIN "IF".
UNSUCCESSFUL COMPILATION; TERMINAL ERROR ON LINE 20.

***** COMPILATION TERMINATED (INTERNAL ERROR 106). OBJECT FILE UNUSABLE. *****

0002 ERRORS  0000 WARNINGS,  P400/500 COBOL REV 15.00 <DRWIST>
```

2. At the end of the listing file.

Example:

```
  .
  .
  .
(5001)                  05  RN-COMPLETION-CD.
(5002)                      07  RN-ID                    PIC XX.
(5003)                      07 RN-ERR-CO                 PIC 99.
(5004)              03  RN-STUDENT-DATA.

*** ERROR *** UNSUCCESSFUL COMPILATION
INTERNAL ERROR =   FULL DIRECTORY
  0001  ERROR   0000    WARNINGS
```

NOTE: In both instances, the object file is unusable!

The COBOL compiler issues a new warning:

'LITERAL TRUNCATED TO ITEM SIZE'.

VALUE OF FILE-ID has been specified by a literal of greater than 8 characters; or,

OWNER IS has been specified by a literal of greater than 6 characters.

The literal is truncated to the correct size.

Larger Address Space

At Rev. 14 and earlier, COBOL programs were restricted to a maximum of a 64K byte total address space. This was further diminished by 4K bytes for each file declared and for each argument passed. The size of a data item (group or elementary) could not exceed 4K bytes.

At Rev. 15, these restrictions have been relaxed or removed. The new characteristics are:

- The total address space which a program uses no longer has an implicit limit.
- The maximum record size is now 32K bytes.

- The OCCURS count may not exceed 32,767. The maximum table size is 32,767 bytes.
- Up to 126 declared files are permitted.

In R mode, the total program together with library size must not exceed 64K words. In V mode, extended addressing is manipulated through compiler generated common blocks.

Language Extensions

- OPEN EXTEND for sequential disk files

    OPEN EXTEND filename-1...[filename-n]

The EXTEND phrase option can be used only for sequential disk files in connection with the WRITE statement.

When the EXTEND phrase is specified, the OPEN statement opens the file, then positions to the bottom of the file (immediately following the last logical record). Subsequent WRITE statements to the file will add records to that file as though it had been opened with the OUTPUT phrase.

- Full IF statements (except arithmetic expression operands)

    The logical connective NOT .
    Negated simple conditions.
    Combined and negated combined conditions.
    Implied subjects on IF statements (simple relation condition and combined relation condition).

V Mode Mag Tape Support

LABEL is a PRIMOS utility which creates ANSI level 1 volume labels on magtape for R mode and V mode. For information on the LABEL utility, the USER should issue the command:

OK, LABEL -HELP

The LABEL utility will respond with:

GO

The LABEL program is used to create an ANSI level 1 volume label on a magtape. To use, enter the command:

LABEL MTn -VOLID volume-id -OWNER owner-id -ACC access

following the 'OK,' prompt issued by PRIMOS.

MTn	specifies the unit number of the tape to be labelled... n must be within the range Ø-7.
volume-id	is a 1-6 character alphanumeric string which uniquely identifies this tape volume.

owner-id — is a 1-14 character string which identifies the owner of the reel. If omitted, the user login-name is defaulted.

access — is a single alphanumeric character which defines accessibility to this volume. This is disregarded by Prime software and is provided only for the sake of completeness. If omitted, this field is left blank.

LABEL will also print the volume and file label information by typing:

```
LABEL MTn
```

OK,

Enhancements and Corrections

- 'DECIMAL POINT IS COMMA' is functional in 64V.
- Level 88 (decimal) is functioning properly in all cases.
- Syntax only compilation (-B NO) is now working correctly.
- COPY statements may contain text after the COPY clause.

In the listing file, the line numbering of the COPY file is now independent of the line numbers of the source.

For example, in the current UFD a file MASTER contains the following:

```
02  BADGE-ID.
    03  FILLER      PIC XX.
    03  ID-INC      PIC 9(5) VALUE ZERO.

02  NAME .
    03  LAST-NAME    PIC  X(14).
    03  FIRST-NAME   PIC  X(14).
    03  MIDDLE-INIT  PIC X.
```

and a source program reads:

```
    .
    .
    .
01  MASTER-RECORD COPY MASTER.
01  EMPLOYMENT-HISTORY.
    .
    .
    .
```

the corresponding listing file would look like:

```
(0059)        .
(0060)        .
(0061)        .
(0062)    01  MASTER-RECORD COPY MASTER.
[0001]        02  BADGE-ID.
[0002]            03  FILLER    PIC XX.
[0003]            03  ID-INC    PIC 9(5) VALUE ZERO.
[0004]        02  NAME.
[0005]            03  LAST-NAME     PIC X(14).
[0006]            03  FIRST-NAME    PIC X(14).
[0007]            03  MIDDLE-INIT   PIC X.
(0062)    01  MASTER-RECORD COPY MASTER.
(0063)    01  EMPLOYMENT-HISTORY.
(0064)        .
(0065)        .
(0066)        .
```

- Compiler now performs all rounding operations properly.
- Magnetic tape input-output is now supported in virtual mode.
- Varying a single digit subscript in a perform statement will now function properly.
- The figurative constant SPACES will not abort the compiler if placed on an alpha-numeric group level.
- Comparison of a numeric field with a non-numeric literal now functions properly.
- Signed data-names will now be considered numeric in level 88 conditional testing.
- A nested conditional within an IF statement will be flagged as illegal.
- Closing an unopened sequential disk file will not produce misleading diagnostics.
- Problems existing with Rev. 14.2 SORT libraries for calling internal sort subroutines from COBOL have been fixed.
- Overwriting a sequential disk file will properly truncate the old file.
- A single digit level 88 conditional will now work as expected.

- Using an invalid level number will now be flagged by the compiler as illegal, rather than abending.
- Attempts to add to a group level will now properly be flagged as illegal.
- Expanded code statistics for number of source lines no longer tally only to 2047 and then wrap around to zero.
- Compiler aborted when a VALUE ZERO clause was placed on an alpha-numeric group item. Abort occured when processing the Data Division with a table group error.
- A VALUE clause not followed by a period and followed by the Procedure Division on the next line will no longer cause the compiler to write to the listing file until the disk is filled.
- The optional word 'SIGN' is no longer flagged as required in the following statement. (CURRENCY 'SIGN' IS '>').
- SEARCH verb now functions correctly in virtual mode.
- Opening a sequential disk file with the OPEN EXTEND verb will now execute properly.
- An erroneous period in a SELECT statement will now be flagged as illegal.
- Writing to a sequential disk file with an invalid key clause will now be properly flagged as illegal.

REV. 15 PRIMOS AND UTILITIES

At Rev. 15 many PRIMOS commands and subsystems have been enhanced. This part describes new features that may be of interest to the COBOL programmer.

The Virtual Loader (RMODE)

At Rev. 15, the virtual loader can be used to replace all copies of the loader under PRIMOS IV, PRIMOS III, and 64K PRIMOS II.

While functionality has been enhanced, the command format of the new Loader is the same as that of the old Loader. For the average COBOL application, the new Loader behaves exactly as the old Loader does; changes required should be minimal.

- Temporary File

  If additional buffer space is required by the Loader, buffers may be paged out of memory into a temporary file which is opened when LOAD is first invoked and not closed until a QUIT or EXECUTE command is given. This temporary file is opened in the HOME UFD, where the user remains attached. For this reason, a conflict may result with existing command files "counting" on a temporary attach point; such command files must be changed.

  If a BREAK or CNTL-P is used to exit from the Loader, the temporary file will remain open on unit 4. The file should be deleted by the user after it has been closed.

- Treenames

  The new Loader will accept either treenames or local file names.

- Command Files

  Command files using temporary attach points must be modified. See Temporary File above.

  Command files using HILOAD must be modified. HILOAD is no longer supported and should be changed to LOAD.

- SAving and EXecuting LOAD Programs

  Although LOAD is a virtual loader, when possible it uses the actual memory locations referenced as its buffers. Programs lying entirely within its buffer space can be started by the EXECUTE command without first SAving the runfile. As delivered, the buffer space is all of memory below '122000.

  For most COBOL programs compiled in 64R mode, the user will be required to save the loaded image before executing.

- Error Messages

  Error reporting for the new Loader has been improved over older versions to include short text descriptions; most, therefore, are self-explanatory. The following are of particular interest to the COBOL user:

  PROGRAM-COMMON OVERLAP - the module being loaded is attempting to load code into an area reserved for common. Use the loader's COMMON command to increase the octal location of common (maximum setting is '177777).

  XXXXXX MULTIPLE INDIRECT - a module loading in 64R mode requires a second level of indirection at location XXXXXX. Insert a Mode D64R command in the load sequence.

  BASE SECTOR Ø FULL - all locations in the sector zero base area have been used. Use the AUtomatic command to generate additional base areas.

- EXAMPLES

  Load sequence for COBOL application without MIDAS (underlining indicates user input):

```
OK, LOAD                 invoke loader
GO
$ LO B<-PROGRAM          load COBOL object file
$ LI COBLIB              load COBOL library
$ LI                     load FORTRAN library
LOAD COMPLETE            load is complete
$ SAVE *PROGRAM          save memory image
$ QUIT                   return to PRIMOS
OK,
```

Load sequence for large COBOL application, and/or one using MIDAS:

```
OK, LOAD                invoke loader
GO
$ MO D64R               set mode if program is large
$ COMMON '177777        move common out of the way
$ LO B←PROGRAM          load COBOL object file
$ AU 20                 set automatic base areas
$ LI COBKID             load COBOL MIDAS library
$ LI                    load FORTRAN library
LOAD COMPLETE           load is complete
$ SAVE *PROGRAM         save memory image
$ QUIT                  return to PRIMOS
OK,
```

NOTE: The map of the new loader has been reformatted. COMMON block names are now in a separate section of the map. Two numeric fields follow the common block name. The first is the location of the COMMON block. The second is the length of the COMMON block in octal, if this value is known. Such information is useful in determining where to set COMMON for large applications.

SEG Enhancements

There are several enhancements to the SEG utility at Rev. 15. These should have little application to COBOL users except for the following:

- Load times are significantly reduced.
- Map format has been slightly changed. 8-character filenames are accepted, and the COMMON block section has been reformatted to include the length of the COMMON block, when known, in octal.
- All commands which leave SEG's Loader (QUIT, RETURN, EXECUTE) now perform a SAVE function.

USING PRIMOS WITH NETWORKS (Rev. 15)

Many Prime installations contain two or more processors connected in a network - a combination of communications hardware and PRIMOS software called PRIMENET. If your system is using PRIMENET, you can do the following:

- LOGIN to a UFD on a remote system and use that CPU to do your processing. (Only terminal I/O is sent across the network.)
- ATTACH to directories on disk volumes connected to any other processor in the network, and access files in such directories. (File data is transmitted across the network; your local CPU does the processing.)
- Enter a CX job in one of your local directories into the CX queue on another processor in the network.
- Make sure a spool file is printed on your local spool queue (if more than one processor is running a spool queue).

In a network, the processor your terminal is connected to is your "local" processor, while all other processors are considered "remote". Each processor in the system is assigned a "nodename" during system configuration. You must know the nodenames of any remote processors you want to access. You may also need to know the local logical disk numbers of disks connected to remote processors. (These are also assigned by your system operator during system configuration.) You can determine the nodename and local logical disk numbers for remote processors with the STATUS command (described later).

For more information on the inner workings of PRIMENET, see the System Administrator's Guide, IDR31Ø9. PRIMENET also supports network-primitive subroutine calls for program-level communication between processes running on different processors. These subroutines are described in PTU52.

Remote Login

The LOGIN command accepts a nodename argument that enables you to log in to a remote system:

```
LOGIN ufd-name [password] [-ON nodename]
```

If -ON nodename is omitted, an attempt is made to log into ufd-name on the local system only. If nodename is the name of the local node, the login attempt is done locally without the use of PRIMENET.

If the LOGIN command fails for any reason (e.g., NOT FOUND, NO RIGHT, BAD PASSWORD), the user's PRIMENET connection is broken, and the terminal is reconnected to the local process (not logged in).

On a terminal logged in to a remote processor, the command LOGOUT logs

out the process, breaks the remote connection over PRIMENET, and reconnects the terminal to its local process (not logged in). Due to network delays, all input characters typed between the LOGOUT command and the response OK are discarded.

Forced Logout

The operator of the local processor system can enter the supervisor terminal command

```
LOGOUT -userno
```

to force the logout of a specified user connected via PRIMENET. userno is the number of a local user process, as shown in the NO column of a STATUS USERS listing (described later).

This command unconditionally logs out the specified user and returns the process to a pool of available remote login server processes; the PRIMENET connection for this terminal/process is broken, and the terminal is reconnected to its local process (not logged in).

Network Information in STATUS Printouts

The STATUS command prints network-related information that identifies local and remote user numbers, logical and physical disk assignments, and line number assignments.

STATUS USERS distinguishes between local and remote users:

```
OK, STATUS USERS

USER   NO LIN PDEVS
PENNY   7  5  50460
CLEMLI 11 11  21460
SUREN  12 12  61060
DOUG.V 14 14  61060
DOUROS 17 17  61060
BD     20 22  10460
COTTON 21 23  21460
HANIF  22 24  61060
HOWIEC 26 30  10460
EMBERS 28 32  21460
TEKMAN 29 33  50460
TEKMAN 30 34  50460
LINDA  32 36  61060
TEKMAN 33 37  50460
SPORER 39 45  21460
BARRIE 40 46  21460
MAGGIE 41 47  50460
TEKMAN 43 51  50460
BD     45 53  460 21460
STEVEN 49 75  10460 (FROM SYSD   USR#43)
SYSTEM 57 77  460
FAM    58 77  460 (2)
```

```
SYSTEM 59 77  61060
SYSTEM 62 77  460
```

This example shows that user STEVEN is local user number 49, is a remote login on line 75 (one of the PRIMENET lines), is currently accessing local physical device 10460, and is logged in from nodename SYSD, where he is user number 43.

STATUS DISKS now shows logical disk number assignments for the local system, including disk volumes on other nodes:

```
OK, STATUS DISKS


DISK   LDEV  PDEV  SYSN
SPOOLH   0    460
PERIPH   1  10460
CPUGRP   2  21460
DOCUMN   3  50460
PRI550   4  61060
SPOOLB   5    460  SYSB
SOFTWR   6   3462  SYSB
DBTEST   7  71063  SYSB
M150A1  10  60460  SYSB
M150B1  11  70460  SYSB
SPOOLD  12    460  SYSD
TRANS   13  21460  SYSD
DBGRP   14  51060  SYSD
TEST    15  71061  SYSD
DTEST   16   2062  SYSD
```

This example shows the status of a three-node system. The first two columns are the packnames and logical device numbers for the local system, and the fourth column shows the nodenames of the remote processors.

The STATUS NETWORK command gives the names and states of all nodes in the network:

```
OK, STATUS NETWORK


SMLC NETWORK

     NODE  STATE
    HARDWR  ****
    RSRCH1   UP

IPC  NETWORK

     NODE  STATE
    HARDWR  ****
    SYSB     UP
    SYSD     UP
```

This shows the state of a four-node network as it would be printed for a local user on the HARDWR node. The UP state means that the node is configured and functioning.

Attaching to Remote Directories

To attach to a directory located in a disk volume at another node, specify the logical disk number of the remote disk (determined from a STATUS DISKS printout) as the ldisk parameter of the ATTACH command:

```
ATTACH directory [password] [ldisk] [key]
```

If ldisk is not specified, the attempt to ATTACH to the remote disk will work only if there is no directory of the same name on a lower logical device number.

Selecting CX Queues on Other Nodes

The CX command line now allows you to place jobs on, or check status of, the CX queue on a remote system:

```
CX{filename} [-ON ldisk]
  {option  }
```

ldisk is the (local) logical disk number of a remote disk containing a CX queue.

Selecting Home Spool Queue

In a network with more than one spool queue in operation, any SPOOL request is intercepted by the first spooler which is ready to accept a job and has the right form type. To make sure the printout takes place on your local spooler, use the -HOME argument in the SPOOL COMMAND:

```
SPOOL filename [-HOME]
```

MODIFIED COMMANDS AND SUBSYSTEMS

Commands and subsystems that have user-visible changes at Rev. 15 are described below in alphabetical order. (See the preceding section on networks for changes to ATTACH, CX, LOGIN,, and SPOOL.)

DELSEG

DELSEG is an internal command which releases segments assigned to the user by SEG. The command format is:

```
DELSEG{segno}
      { ALL }
```

where segno is the number of the segment to be freed. segno must be

greater than or equal to 2ØØØ (octal) and not equal to 6ØØØ (octal). Specifying ALL as the argument frees all segments assigned to the user issuing the command. Deleting an already nonexistent segment has no effect. Attempting to delete an illegal segment number yields the error message BAD PARAMETER.

FUTIL

Three new commands have been put into FUTIL at Revision 15. These commands are SRWLOC, TRESRW and UFDSRW. They set the per-file read-write lock for a file, a tree, and all files in the current UFD, respectively. The format of these commands correspond to the format of the protect-class commands, i.e.:

```
SRWLOC filename lock-number
TRESRW pathname lock-number
UFDSRW lock-number n-levels
```

lock-number is the read-write lock. If omitted, Ø is the default. n-levels is the number of levels to go down doing the setting. The read-write lock is interpreted as follows: Ø means use the system read-write lock, 1 means allow multiple readers or one writer, 2 means allow multiple readers and one writer, 3 means allow multiple readers and multiple writers.

To output a file's read-write lock, use the RWLOCK option in the LISTF command in FUTIL. A read-write lock of Ø appears as "SYS", 1 appears as "W/NR", 2 appears as "1WNR", and 3 is shown by "NWNR".

FUTIL now ignores null lines and accepts lower case input. However, passwords must be entered in the same case as they were assigned.

The CLEAN command no longer leaves protection for files below current level at 7 Ø. Instead, it leaves them the way they were.

Volume names or numbers used as a prefix (i.e., beginning with <) must now also end with >.

The first digit of a segment directory file or sub-segment directory, (i.e., the first digit of the number in parentheses) must be a digit.

MIDAS

MIDAS for Rev. 15 contains no new features. The only enhancement to MIDAS has been the creation of a version of MIDAS which can be shared on the Prime 4ØØ (or higher) for V-mode programs. For details, see PTU54.

SORT

Sort accepts upper and lower case characters. Lower case characters are sorted as if they were upper case, but they appear as lower case

characters in the output file.

TERM COMMAND

The TERM command is a useful tool to control the duplex of a terminal as well as setting the kill and erase characters and enabling or disabling the BREAK key or enabling the X-ON/X-OFF option. The command line for the REV. 15 TERM command will look for options to be preceded by a dash (-), the old way (options without the dash) will still work for compatibility. The rest of this document will be dedicated to explaining the different command line formats for the TERM command.

A.) TERM

Typing TERM without any options will have the program print a general list of possible command line formats.

B.) TERM -ERASE (char)

This will set the erase character from its current value to that of char which is specified in the command line.

C. TERM -KILL (char)

This will set the kill character from its current value to that of char which is specified in the command line.

Note:

char must be a single character and the parenthesis are not to be specfied.

D.) TERM -BREAK ON

This enables the BREAK or [CONTRL-P] key.

E.) TERM -BREAK OFF

This disables the BREAK or [CONTRL-P] key.

F.) TERM -HALF -[XOFF or NOXOFF] -[LF or NOLF]

The parameters in the brackets are optional. The HALF duplex key will not echo back input from the terminal. The NOLF will not echo a line feed after a carriage return. A LF will echo a line feed after a carriage return. An XOFF will enable the X-OFF/X-ON feature, a NOXOFF will disable the X-OFF/X-ON feature. If the [XOFF or NOXOFF] option is omitted the TERM command will default to the state of the X-OFF/X-ON that existed before the TERM command was invoked. When enabled, CONTROL-S performs the X-OFF and CONTROL-Q the X-ON function.

G.) TERM -FULL -[XOFF or NOXOFF]

The FULL duplex key will echo back input from the keyboard to the terminal screen. The [XOFF or NOXOFF] feature will work as described in paragraph F.

H.) TERM -[XOFF or NOXOFF]

This form will set the terminal to FULL duplex (default value) and enable or disable the X-OFF/X-ON according to the specified command in the command line.

I.) TERM -DISPLAY

This format will print out the terminal's kill and erase chacacters as well as whether the terminal is in full or half duplex or if the X-ON/X-OFF feature is enabled, or if an X-OFF (CONTRL-S) has been received.

SHARED LIBRARIES

At Rev. 15 certain V-Mode libraries can be established as shared libraries by the System Administrator. For more information see the Rev. 15 version of IDR31Ø9, the System ADministrator's Guide.