PRIME

Preliminary Documentation Release

PDR3057 FORTRAN PROGRAMMER'S GUIDE

P/N 3064-001

PRIME'S FORTRAN IV PROGRAMMER'S GUIDE

This guide documents Prime FORTRAN IV and all supporting PRIMOS operating system features as implemented at Master Disk Revision Level 14. It is organized to make life easier for you, the FORTRAN IV application programmer.

We assume you know FORTRAN, 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 FORTRAN application running.

To save you the 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 FORTRAN programmer
- Presenting these capabilities in the usual order of FORTRAN 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 FORTRAN application programs.

In exceptional cases, you may need to refer to supporting reference documents (illustrated). 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.

The accompanying table gives guidelines on the tasks that are fully described in this guide and the extent to which the reference documents apply.

We hope you will find this to be a helpful guide to the particulars of FORTRAN 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 R. Lewis, Technical Writer
Technical Publications Department
Prime Computer, Inc.
145 Pennsylvania Avenue
Framingham, MA 01701

Acknowledgements:

We wish to thank the members of the FORTRAN PROGRAMMERS GUIDE team and also the non-team members, both customer and Prime, who contributed to and reviewed this PDR.

PRIME DOCUMENTATION TYPES

IDR Initial Documentation Release: provides usable, accurate advanced information without regard to style and format.

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 produced at a high standard of graphic quality

MAN Manual: early reference documents to be phased out by PDR's and FDR's.

PTU Prime Technical Update: interim updates to existing documents.

First Printing November 1977

TABLE OF CONTENTS

PART I - OVERVIEW

PART II - USING FORTRAN UNDER PRIMOS

SECTION 12 SHARED CODE AND OTHER ADVANCED SEGMENTED PROGRAM TECHNIQUES

SECTION 13 INTERFACE TO OTHER SYSTEMS AND LANGUAGES

SECTION 14 OPTIMIZATION AND OTHER HELPFUL HINTS

PART IV - FORTRAN LANGUAGE REFERENCE

SECTION 15 FORTRAN LANGUAGE ELEMENTS

SECTION 16 FORTRAN STATEMENTS

ILLUSTRATIONS AND TABLES

ILLUSTRATIONS

TABLES

PART I

OVERVIEW

SECTION 1

OVERVIEW

INTRODUCTION

This document is a comprehensive guide for the Prime FORTRAN programmer. It contains everything normally necessary for writing, compiling, loading, and executing FORTRAN programs. The user is assumed to be familiar with the FORTRAN language but not with its implementation and use on a Prime computer. Users unfamiliar with the language should read one of the commercially available instruction books; two examples are:

McCracken, Daniel D., A Guide to FORTRAN IV Programming, John Wiley and Sons, inc.

Organick, Elliott I., A FORTRAN IV Primer, Addison-Wesley Publishing Company.

The current definitive standard for the FORTRAN IV language is the American National Standards Institute publication X3.9-1966 (USA Standard FORTRAN).

This Version

This is a Preliminary Documentation Release, documenting Prime FORTRAN IV and supporting utilities at software revision level 14 (Rev. 14). It is more complete than the previous documentation on the use of FORTRAN under PRIMOS (Prime Computer Operating System) and replaces the following documents:

FORTRAN IV USER GUIDE, MAN1674
Rev. 10 VFTN, PTU24
Rev. 11 FTN for PRIMOS II, III, IV, PTU26
FTN (Rev. 13), PTU35
PROGRAM DEVELOPMENT SOFTWARE USER GUIDE, MAN1879
(Sections 5, 7, and 9)
PDS UPDATES (Rev. 13), PTU33

This document is not yet in its final form. Certain sections, less central to the major purposes of this guide, have not been included; they are represented here by a detailed outline with references to existing documentation from which the desired information may be extracted. These sections will be included in the Final Documentation Release which will be a typeset manual.

Organization

The guide is composed of five major parts:

Part 1. An introductory section including an overview of FORTRAN as it is implemented on the Prime Computer. This includes Prime extensions to the language, supporting utilities, systems, and software, plus where to find information in this document. (Section 1).

Part 2. Using the Prime computer for FORTRAN programming. This is a tutorial, arranged to follow the normal sequence of program development. A single pass through this part will enable the user to perform all the usual FORTRAN programming functions. The order of information presented is (see Figure 1-1):

accessing the system (Section 2)
creating a program (Section 3)
compiling (Section 4)
loading for relative address code (Section 5)
or segmented-address code (Section 6)
executing (Section 7)

A final section covers debugging concepts and the use of debugging tools available to the FORTRAN programmer (Section 8).

System utilities are introduced and all concepts and PRIMOS-level commands necessary for the large majority of uses are discussed with examples. A user wishing to go beyond these for special programming concepts, more efficient program creation, program optimization, etc., will find references to the information (either in this document or another reference document) at the appropriate place. In most cases, it is unnecessary to use any document other than this one.

Part 3. Advanced Techniques. Sections 9-14 cover a range of specialized topics including program optimization with the segmented loader, loading for shared procedure, introduction to the MIDAS, DBMS, and FORMS systems in the FORTRAN environment, and additional details on extended use of the operating system and file management system.

Part 4. FORTRAN language reference. Sections 15-17 form a reference for the FORTRAN language as implemented on Prime computers. The Prime extensions to the standard language are given along with examples of their usage.

Part 5. Utility Reference. Provides more detailed and extended information about the use of the utilities supporting FORTRAN. In addition, libraries are listed and the library functions and subroutines which are particularly useful are described in detail. The user is told of the existence and functionality of less useful, lower level subroutines and where to find complete information about them.

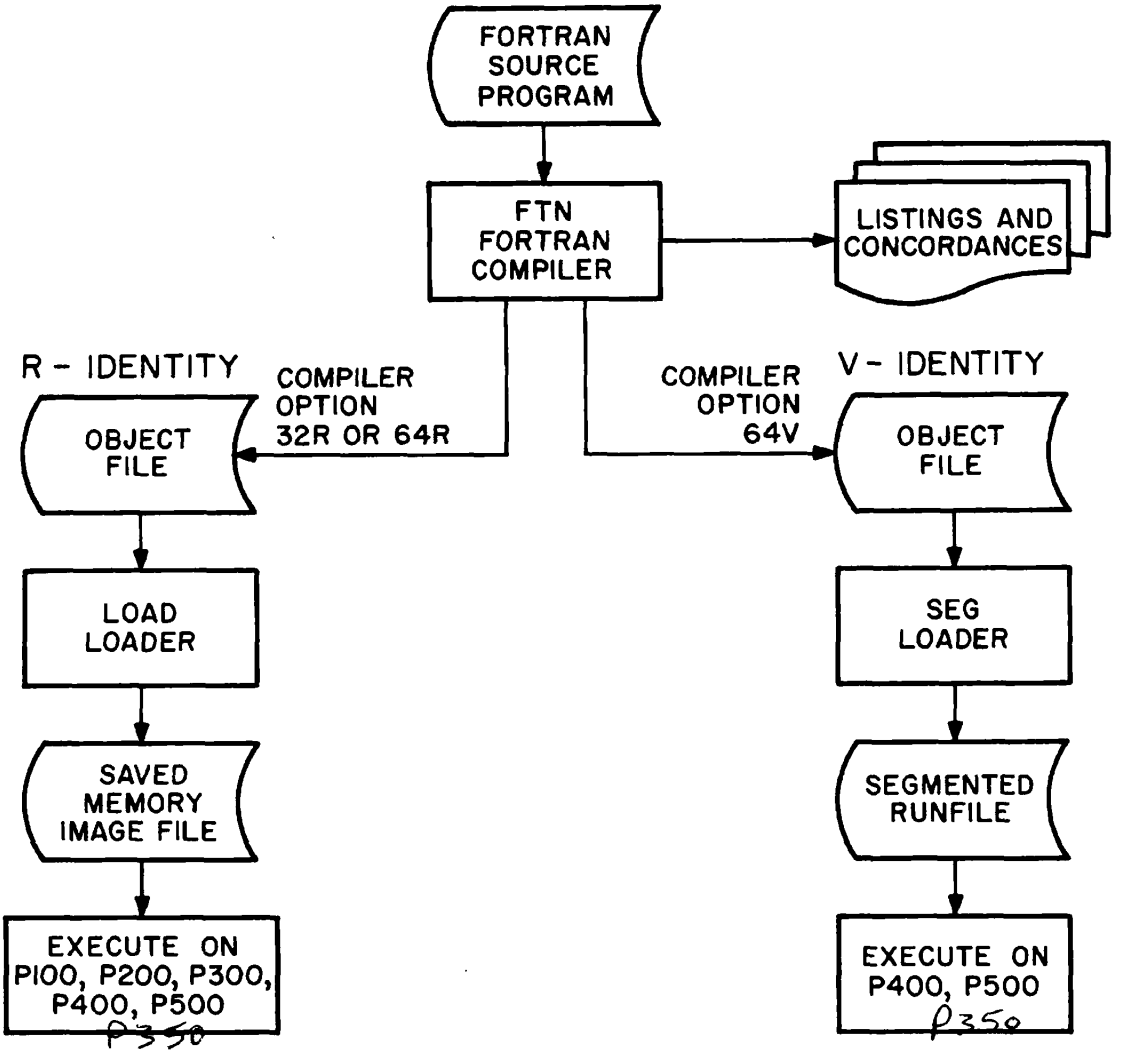


Figure 1-1. Sequence of FORTRAN Program Development

A complete list of compiler, loader , and run-time error messages and their meanings (Appendix A) and system defaults and constants (Appendix B) and ASCII character set (Appendix C) are included.

Related Documents

The following documents contain detailed reference information on the PRIMOS system and utilities.

Operating System Reference

PRIMOS INTERACTIVE USER GUIDE, MAN2602
updated by PTU31 and PTU42

REFERENCE GUIDE, FILE MANAGEMENT SYSTEM (FMS), PDR3110

REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106

Software Subsystem Reference

THE NEW USER'S GUIDE TO EDITOR AND RUNOFF, PDR3104

REFERENCE GUIDE, MULTIPLE INDEX DATA ACCESS SYSTEM (MIDAS), PDR3061

REFERENCE GUIDE FOR DBMS SCHEMA DDL, IDR3044

FORTRAN REFERENCE GUIDE FOR DBMS, IDR3045

FORMS MANAGEMENT SYSTEM (FORMS), IDR3040
Updated by PTU45

CONVENTIONS AND DEFINITIONS

Terminal Functions

(CR) or CR	Carriage return
"	Character erase; deletes preceeding character.
?	Line kill; deletes all characters in current line.
↑xxx	Escape ~~key for entry of~~ non-printing characters with ASCII codes xxx, as in ↑207 for CTRL-G or Bell code.

symbol used in text editor to enter octal codes of

Typographic Conventions

[ ]	Brackets enclose optional item(s).
{ }	Braces enclose a number of items, one of which must be selected.
ALLCAPS	A literal item that is to be entered exactly as shown. Acceptible abbreviations are underlined (see below).
lower-case	A parameter whose legal value is to be selected by the user.
_____	Underlining. There are three major uses;

- Indicate acceptable abbreviations in a command line, e.g.,

  ASSIGN device -WAIT

  is equivalent to

  AS device -WAIT

- Indicate user input in an interactive session, e.g.,

  OK, SEG
  GO
  # VLOAD #BENCH

  etc.

- Emphasize command line parameters being described in text following command lines, e.g.,

  SPOOL filename [-option-1] [-option-2]...

  places a file in the queue to be printed by the line printer.

  filename is the file to be placed in the line printer spool queue; option-1 ... (etc.)

CR or (CR)	A carriage return; terminates an input line at the user terminal.
2048	A decimal number.
'4000 or :4000	An octal number. The second form is that used in FORTRAN programs.
K	1024 (decimal); '2000 (octal).

Example: 64K words of memory is

64x1024 = 65536 words (decimal)

or

'100x'2000 = '200000 words (octal)

Filename conventions

filename	Source file
B←filename	Binary (object) file; compiler convention
L←filename	Listing file; compiler convention
M←filename	Map file
*filename	Saved executable memory image (R-identity)
#filename	Saved executable segmented runfile (V-identity)
C←filename	Command Input File ~~Command file~~
O←filename	Command Output File

Filenames may be up to 32 characters long (6 characters on old file partitions).

On some output devices ← may print as underscore (_).

Basic Concepts

file	An organized collection of information stored on a disk (or a peripheral storage medium such as tape). Each file 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 which is in a UFD or another 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 labeled by an octal number called the logical disk number.

volume name — A literal name corresponding to a logical disk, e.g., logical disk 4 may have volume name DOCUMN.

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 forms:

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

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 on.

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 FORTRAN compiler. Such files are in the format required as input to the linking loader or segmented loader.

runfile — The 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 or SEG.

mode	An addressing scheme. The mode used determines the construction of the computer instructions by the compiler. Modes available to the FORTRAN programmer are relative-addressed (32R or 64R) and segmented-addressed (64V). (The number is the user memory size in K's of 16-bit words.)
identity	The addressing mode plus its associated repetoire of computer instructions. Programs compiled in 32R or 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

FORTRAN FEATURE SUMMARY

Material for this sub-section is described in detail in the Language Reference Section of this document (Sections 15-17). A summary will be included here in the final version. Extensions to standard FORTRAN which the user should inspect are:

- Use of the $INSERT command for file insertion at compilation
- B Format
- TRACE command for debugging
- List-directed input/output
- Long integers
- Parameters
- IMPLICIT specification
- Subprogram structure

FORTRAN UNDER PRIMOS

Program Conversion

There are a number of factors which must be taken into account when converting FORTRAN programs from one computer system to another. These are the language statements, input/output, functions, subroutines, and control flow. Any particular program may have special conversion needs but these are the major areas to consider.

Language: Make certain that all statements perform the same operations on both systems. The major sources of possible incompatibility are device and input/output statements. The 1966 standard FORTRAN does not

fully describe certain statements such as ENDFILE or REWIND; consequently, their exact performance is installation dependent. Extensions to the standard READ and WRITE statements are not uniform throughout the industry; these extensions must be changed to Prime's usage or the appropriate Prime subroutines inserted. Levels of nesting in DO loops and IF statements will present no problems as there is no syntactical limit on such nesting in Prime FORTRAN. Similarly, there is no syntactical limit to the number of statement labels in computed GO TO statements.

Input/Output: FORTRAN logical unit numbers must agree with those given in Section 16 of this document (or such others as are established by the system manager). As PRIMOS is an interactive multi-user system there is no need for a job control language, all users have access to disk files. Use of peripheral storage devices is obtained by assigning the device to the user (see Section 3) after which file operations may be performed.

Functions: Prime supplies a large number of the normal mathematical functions plus a set of (Boolean) functions. These are listed in Section 20. The user should check these to be sure all functions in the original source program are implemented under PRIMOS. It is unlikely that the average programmer will be using functions not on this list. User-defined functions should be written as specified in Section 17.

Subroutines: Inasmuch as all operating system or file system calls are installation-dependent, all such calls must be replaced by their PRIMOS equivalents. Subroutines for all normal usages will be found in Section 20, especially in the Applications Library, which is given here in its entirety. Subroutines for extended usage or special cases will be found in REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106. User-defined subroutines should be written to the specifications in Section 17.

Control Flow: To insure an orderly return from the main program to the PRIMOS level the last logical statement of a main program must be

```
CALL EXIT
```

This is analogous to the RETURN statement which is the last logical statement of a function subprogram or subroutine.

Programs executing in the R-identity may be "chained" by use of the RESU$$ subroutine described in Section 20, Operating System Library.

Program Environments

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

- Interactive
- Phantom user
- Sequential job processing

Interactive: Program execution is initiated directly by the user (Section 7). The program runs in real time and is "connected" to the terminal. The program will accept input from the terminal and will print at the terminal any output specified by the program as well as user- or system-generated error messages. This environment is the one most often used. Major uses are:

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

Phantom User: The phantom environment (Section 9) allows programs to be executed while "disconnected" from a terminal. This frees the terminal for other uses. Phantom users accept input from a command file instead of a terminal; output directed to a terminal is either ignored or directed to a file.

Users may interrupt a program running as a phantom. Major uses of phantoms are:

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

Sequential Job Processing:

The number of phantom users on a system is fixed. Sequential job processing queues requests for phantom users and then executes these jobs ~~as phantoms become available~~ (Section 9) one at a time.

This environment is especially useful when phantom usage is heavy and real time execution of programs is not a requirement.

PRIMOS Command Summary

There are in excess of one hundred commands which may be given to the operating system (PRIMOS-level commands). Commands which are of use to the FORTRAN applications programmer are listed below with a brief explanation of their function. Most of these commands are discussed in this document. Consult the index under the command name to locate where the command is discussed. Commands not discussed in this document are treated in detail in PRIMOS INTERACTIVE USER GUIDE, MAN 2602 (with its updates PTU31 and PTU42).

Certain PRIMOS commands are not listed here because they are specific to other high-level languages, assume a knowledge of assembly language, are network commands, or are operator commands. They will not be of use to the FORTRAN applications programmer. They are discussed in detail in the PRIMOS interactive documentation.

Command	Function
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
CLOSE	Closes named files or file units as specified
CMPRES	Compresses ASCII file
CNAME	Changes a filename
COMINPUT	Switches command stream from terminal to file and vice-versa
COMOUTPUT	Switches terminal output to file and vice-versa
CPMPC	Punch cards on parallel interface card punch
CPPMPC	Punch cards and print text on parallel interface card punch
CREATE	Creates a sub-UFD in the current UFD
CRMPC	Reads cards from the parallel interface card reader
CRSER	Reads cards from the serial interface card reader
CX	Invokes the sequential phantom job execution utility
DATE	Prints system time and date at terminal
DELETE	Deletes a filename from the UFD
ED	Invokes Prime's text editor
EXPAND	Expands a file previous compressed with CMPRES
FILMEM	Fills the user memory space with zeroes
FILVER	Compares two binary files and prints differences
FTN	Invokes FORTRAN compiler
FUTIL	Invokes Prime's file manipulation utility
HILOAD	Same as LOAD, but restores the loader higher in memory
INPUT	Opens file for reading on PRIMOS unit 1
LISTF	Prints list of entries in current UFD
LISTING	Open a file for writing on PRIMOS unit 2
LOAD	Invokes the Linking Loader (R-identity)
LOGIN	Logs the user in to the system
LOGOUT	Logs the user off the system
MAGNET	Invokes the magtape/disk tranfer/translation utility
MAGRST	Transfers files from 9-track tape to disk
MAGSAV	Transfers files from disk to 9-track tape
MDL	Punches paper tape of memory image in self-loading format

MESSAGE	Transmits message from user terminal to system console
OPEN	Opens a file on a specified PRIMOS unit
PASSWD	Sets passwords for current UFD
PHANTOM	Initiates job execution released from user terminal
PRMPC	Print on parallel interface driven line printer
PROTECT	Sets owner/non-owner rights for files and sub-UFDs
PRSER	Print on serial interface driven line printer
PRVER	Prints a file on the Versatec(TM) printer/plotter
PTCPY	Duplicates and verifies paper tapes
PUSS	Compares two ASCII files
RESUME	Restores a file to user's memory and begins execution
SEG	Invokes the segmented-address (V-identity) utility
SHARE	Incorporates files into shared segments
SIZE	Gives size of file in records
SLIST	Prints contents of file to user's terminal
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
TIME	Prints connect, compute, 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
* or /*	Comment line for command files

File System Summary

PRIMOS allows the user to access up to 16 (15 under PRIMOS II) files at one time. These disk files may be created, modified and deleted through the use of the Applications Library subroutines (Section 20) and the file management subroutines of the Operating System (Section 20). The file system is discussed in Section 10. Files, opened by these subroutines, may be accessed by FORTRAN I/O statements such as READ, WRITE, ENCODE, DECODE. See Section 16 for a complete discussion of these commands.

SYSTEM RESOURCES SUPPORTING FORTRAN

There are a large number of libraries and utilities in PRIMOS supporting the use of FORTRAN on the Prime computer. A brief description of some of the major ones follows.

Libraries

Library functions and subroutines of use to the FORTRAN applications programmer are in Section 20 of this document. A complete treatment of all library and system subroutines is in REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106.

A summary of the FORTRAN mathematical functions is given in Figure 1-2. There are also FORTRAN functions for the (logical) operations of AND, OR, XOR, NOT, right shift, right truncate, left shift, and left truncate. Conversion between data modes is supported by a set of conversion functions. For more advanced mathemetical usage, a matrix library is provided (see Figure 1-3 for a summary). Complete descriptions of the In-memory Sort Library and the Applications Library are in Section 20. Finally, the operating system subroutines and their functions are listed. Those which will be of use at the applications level are described in detail.

Compiler

Prime's FORTRAN IV compiler operates on FORTRAN source code to generate highly optimized object code. The user has the option, at compilation time, of generating object code for execution in either the R-Identity or V-identity. Additional options control I/O specifications, listings, concordances, memory usage, and other useful operations. The compiler is described in Section 4 with a detailed reference in Section 18.

Linking Loader

The R-identity loader combines into an executable program, program modules, subroutines, and libraries that have been compiled separately. It handles symbol cross references and module linkages. Maps of the load are available at the terminal or written into files. The Linking Loader is described in Section 5.

SEG Utility

SEG is the V-identity program loading and execution utility. It combines separately compiled program modules, subroutines, and libraries into an executable program. Program lengths can be up to 960K or 1984K words (depending upon the version of SEG installed). All memory management, symbol tables, linkages, etc. are handled by SEG's loader. Various types of loadmaps may be obtained. The SEG utility has many functions, they are described as follows:

Normal usage (Section 6)
Extended usage (Section 11)
Using SEG for loading shared procedure (Section 12)
Reference (Section 19)

Editor

Prime's text editor is a line-oriented editor enabling the programmer to enter and modify source code and text files. Information for these purposes is in Section 3; a complete description of the Editor is in THE NEW USER'S GUIDE TO EDITOR AND RUNOFF, PDR3104.

Multiple Index Data Access System (MIDAS)

MIDAS is a system of utilities and subroutines for creating and maintaining keyed-index/direct-access files. All housekeeping functions on the index and data sub-files are performed by MIDAS subroutines called from FORTRAN programs. An overview of MIDAS is in Section 13, the complete documentation is REFERENCE GUIDE, MULTIPLE INDEX DATA ACCESS SYSTEM (MIDAS), PDR3061.

Database Management System (DBMS)

Prime's DBMS is a CODASYL-compliant system for management of large amounts of data. DBMS can be accessed from either FORTRAN or COBOL programs. Complete information on using DBMS in the FORTRAN environment is in REFERENCE GUIDE FOR DBMS SCHEMA DDL, IDR3044 and FORTRAN REFERENCE GUIDE FOR DBMS, IDR3045.

Forms Management System (FORMS)

FORMS is a system for creation, maintenance, and use of screen forms for interactive file maintenance. These screen forms are an extremely useful tool for the ~~FORTRAN~~ applications programmer writing ~~file maintenance programs~~ data entry programs. Details are in FORMS MANAGEMENT SYSTEM (FORMS), IDR3040 and its update PTU45.

Language Interfaces

Under the PRIMOS operating system FORTRAN programs may call or be called by PMA (Prime Macro Assembly) language programs. FORTRAN subroutines may be called from COBOL programs. Details are in THE PMA PROGRAMMER'S GUIDE, PDR3059 and THE COBOL PROGRAMMER'S GUIDE, PDR3056.

SAMPLE PROGRAM DEVELOPMENT

This sub-section will be included in the final version. It will serve as an annotated map to this document.

It is not intended to teach the programmer the use of the system but to provide a framework on which to hang more detailed information presented later in the manual.

Operation	Data Mode of Argument and Value: Integer	Single-Precision	Double-Precision	Complex
Sine	n/a	SIN	DSIN	CSIN
Cosine	n/a	COS	DCOS	CCOS
Arctangent	n/a	ATAN	DATAN	
Arctangent of ratio	n/a	ATAN2	DATAN2	
Hyperbolic tangent	n/a	TANH		
Log-base e (Ln)	n/a	ALOG	DLOG	CLOG
Log-base 2	n/a		DLOG2	
Log-base 10	n/a	ALOG10	DLOG10	
Exponential	n/a	EXP	DEXP	CEXP
Square root	n/a	SQRT	DSQRT	CSQRT
Absolute value	IABS	ABS	DABS	CABS
Remainder (modulus)	MOD	AMOD	DMOD	n/a
Truncation to Integral value	n/a	AINT	DINT	n/a
Positive difference	IDIM	DIM		n/a
Magnitude of first number times sign of second	ISIGN	SIGN	DSIGN	n/a
Complex conjugate	n/a	n/a	n/a	CONJG
Random number	IRND(1)	RND		n/a
Maximum of List	AMAX0(2)	AMAX1	DMAX1	n/a
	MAX0	MAX1(3)		n/a
Minimum of List	AMIN0(2)	AMIN1	DMIN1	n/a
	MIN0	MIN1(3)		n/a

Notes

n/a - Not applicable.

1 - Accepts short integer argument only; all other integer functions accept combinations of short and long integers.

2 - Value mode is single-precision.

3 - Value mode is integer.

Figure 1-2. FORTRAN Mathematical Functions

Operation	Data Mode of Matrix Elements: Integer	Single-Precision	Complex	Double-Precision	
Setting matrix to identity matrix	IMIDN	MIDN	CMIDN	DMIDN	*
Setting matrix to constant matrix	IMCON	MCON	CMCON	DMCON	
Multiplying matrix by a scalar	IMSCL	MSCL	CMSCL	DMSCL	
Addition of matrices	IMADD	MADD	CMADD	DMADD	
Subtraction of matrices	IMSUB	MSUB	CMSUB	DMSUB	
Matrix Multiplication	IMMLT	MMLT	CMMLT	DMMLT	
Calculating transpose matrix	IMTRN	MTRN	CMTRN	DMTRN	*
Calculating adjoint matrix	IMADJ	MADJ	CMADJ	DMADJ	*
Calculating inverted matrix	n/a	MINV	CMINV	DMINV	*
Calculating signed cofactor	IMCOF	MCOF	CMCOF	DMCOF	*
Calculating determinant	IMDET	MDET	CMDET	DMDET	*
Solve a system of linear questions	n/a	LINEQ	CLINEQ	DLINEQ	
Generate permutations	PERM				
Generate combinations	COMB				

Notes

n/a - Not applicable

* - For square matrices only

Figure 1-3. Matrix Operations Subroutines

PART II

USING FORTRAN UNDER PRIMOS

SECTION 2

ACCESSING PRIMOS

The following information will be discussed in the final version of this document. For detailed information, refer to: PRIMOS INTERACTIVE USER GUIDE, REVISION A, MAN2602.

PRELIMINARY ELEMENTS

- Character Set:
  - Legal characters
  - Control characters
  - Reserved characters
- Terminal:
  - Operation
  - Special keys

USER FILE DIRECTORY OPERATIONS

- Directory Structure (MFD, UFDs, sub-UFDs, treenames)
- PRIMOS command format
- Logging in - The LOGIN command
- Creating and deleting UFDs
- Passwords for directories
- Directory Contents - The LISTF command
- Logging out - The LOGOUT command

PRIMOS COMMANDS FOR THE FORTRAN PROGRAMMER

- Alphabetical summary

SECTION 3

ENTERING AND MANIPULATING SOURCE PROGRAMS

ENTRY FROM OTHER MEDIA

Existing source programs resident on punched cards, magnetic tape, or punched paper tape can easily be read into 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)
2. Transfer programs with appropriate utility
3. Relinquish exclusive use of the device (Unassigning)

Assigning A Device

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

ASSIGN device [-WAIT]

device is a mnemonic for the appropriate peripheral:

CR	Card Reader
MTn	Magnetic Tape Unit n
PTR	Paper Tape Reader

-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 CTRL/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 is relinquished by the command:

UNASSIGN device

device is the same mnemonic used in the ASSIGN command.

Reading Punched Cards

Assign use of the parallel interface card reader by:

AS CR -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 are 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, the reader should be reloaded; reading is resumed by the command:

S

at the terminal.

The command:

C ALL

or

C ~~treename~~ Filename

will close the file.

Example of card reading session:

```
OK, AS CR -WAIT
OK, CRMPC old-program-1
OK, UN CR
OK,
```

If a serial interface card reader is used, the process is similar with slightly different reader commands.

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

CARDR may be abbreviated to CAR.

Reading Magnetic Tape

Assign use of the magnetic tape drive by:

```
AS MTx -WAIT
```

where x is the tape drive unit number: 0,1,2, or 3.

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

```
OK, MAGNET
GO

MAGNET 14.0  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 then asks a series of questions about the tape format:

```
MTFILE# =    tape-file-number
```

tape-file-number	is the file number on the tape. A positive integer causes the tape to be rewound and then positioned to the file number; a 0 causes no repositioning of the tape.

LOGICAL RECORD SIZE = 80

This is the number of bytes/line image; normally this is 80 for a FORTRAN source program.

BLOCKING FACTOR = blocking-factor

blocking-factor is the number of logical records per tape record.

ASCII, BCD, BINARY, OR EBCDIC? data-representation

data-representation	action
ASCII	transfer ~~with bit unpacking~~
BCD	translation to ASCII from 7-track tape ~~no unpacking~~
BINARY	transfer verbatim
EBCDIC	~~translation~~ to ASCII ~~with bit unpacking~~ transfer

FULL OR PARTIAL RECORD TRANSLATION? answer

answer is FULL or PARTIAL. The question is asked only for BCD or EBCDIC representations. Partial translation allows specified bytes in the record to be transferred to disk without translation to ASCII. The partial option is useful when transferring data files, but almost all source programs will be transferred with the full option.

OUTPUT FILENAME: filename

filename is the name of the file in the UFD into which the magnetic tape is to read. If the filename already exists in the UFD, the question:

OK TO DELETE OLD <filename>? answer

will be asked. A NO will cause the request for an output filename to be repeated. A YES will cause the transfer to begin; upon completion, the following message will be printed out:

DONE, <tape-records> RECORDS READ, <disk-records> DISK RECORDS OUTPUT OK,

Use of the tape drive unit should then be relinquished by

UN MTx

Reading Punched Paper Tape

Source programs punched on paper tape in ASCII representation can be read into a disk file with the Editor utility. First load paper tape into reader; then assign tape reader.

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

ENTERING AND MODIFYING PROGRAMS - THE EDITOR

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

```
ED
```

which places the Editor in the INPUT mode. When modifying an existing filename, the Editor is invoked by

```
ED filename
```

which places the Editor in the EDIT mode.

A 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 done with the backslash (\) character. Each backslash represents the first, second, etc. tab setting; the default tabs are at columns 6,

15, and 30. These settings 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 50 commands are available, although users will find that a small subset of these will suffice for most purposes. The 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 RETURN without any preceding characters puts the Editor into the INPUT mode.

Special Characters

In either mode, a single character can be erased with the erase symbol (default is "). For each " typed, a character is erased (from right to left). The entire current line may be deleted by typing the kill character (default is ?). A line followed by a ? is null, and a RETURN at that point will switch the Editor into the other mode.

Do not use the semicolon (;) character unless you are familiar with the Editor; it has a special meaning as a control character. In input mode ; ≡ CR (ends a line of input). In edit mode ;'s in a character string are treated as printing characters; otherwise, semicolons separate multiple commands entered on the same line.

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 it does exist. If an existing file is being modified, the command

FILE

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

Useful Techniques

The following will aid the user in adapting to Prime's Editor.

Tab Settings: When entering source code, much time can be saved using the TABSET command. In INPUT mode, each \ character is interpreted as one tab setting; the default values are columns 6, 15, and 30. Tabs may be set to whatever values each programmer finds useful. Setting a tab near column 45 makes entry of in-line ~~comments~~ comments simple; the use of such ~~commands~~ comments in programs is strongly advised.

Moving Lines of Code: Any number of lines can be moved from one location to another using the DUNLOAD command. DUNLOAD deletes these lines as it writes them into an auxiliary file. A LOAD command loads the new file at the desired point. Any number of lines can be copied from one location in a program to another using the UNLOAD command. UNLOAD does not delete these lines as it writes them into an auxiliary file. A LOAD command loads the copy from the new file at the desired point.

Overlaying Comments After Code is Written: Comments may be easily added to an existing source program with the OVERLAY command using tabs.

Finding a Line by Statement Number: The FIND command may be used to locate a statement number in a FORTRAN program.

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

Sample Editing Session

See the list following this example for an explanation of the commands.

```
OK, ED
GO
INPUT

EDIT
TABSET 7 45

INPUT
C THIS IS A SIMPLE TEST TO DEMONSTRATE USE OF THE EDITOR
C THE TABS HAVE BEEN SET TO COLUMNS 7 AND 45
C
\PRINT 1, 'THIS IS A TEX"ST'\/* NOTE USE OF ERASE CHARACTER
  1    ?C THIS LINE HAS BEEN DELETED

EDIT
TOP
PRINT 20
.NULL.
C THIS IS A SIMPLE TEST TO DEMONSTRATE USE OF THE EDITOR
C THE TABS HAVE BEEN SET TO COLUMNS 7 AND 45
C
     PRINT 1, 'THIS IS A TEST'              /* NOTE USE OF ERASE CHARACTER
C THIS LINE HAS BEEN DELETED
BOTTOM
FILE TEST99

OK, ED TEST99
GO
EDIT
TABSET 7 45
FIND(8) LINE
C THIS LINE HAS BEEN DELETED
DELETE
INSERT \CALL EXIT /* FOR AN ORDERLY EXIT TO PRIMOS
INSERT \END

INPUT
P"TOP
PRINT 20
.NULL.
C THIS IS A SIMPLE TEST TO DEMONSTRATE USE OF THE EDITOR
C THE TABS HAVE BEEN SET TO COLUMNS 7 AND 45
C
      PRINT 1, 'THIS IS A TEST'             /* NOTE USE OF ERASE CHARACTER
      CALL EXIT                             /* FOR AN ORDERLY EXIT TO PRIMOS
      END
BOTTOM
FILE

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, PDR3104.

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 of 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/[n]~~[c]~~ {G} {^}	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]	Deletes 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 from current file and writes them into filename. (Default n=1.)
DUNLOAD filename TO string	Same as DELETE...TO, but writes deleted lines into filename.
ERASE character	~~Resets current~~ Sets erase character to character.
• FILE [filename]	Writes the contents of the current file into filename and QUITs to PRIMOS.

FIND string	Moves the pointer down to the first line beginning with string.
• FIND(n) string	Moves the pointer down to first 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	~~Resets current~~ 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 first line containing string, which may contain leading and trailing blanks.
MODE COLUMN	Displays column numbers whenever INPUT mode is entered.
MODE COUNT start increment width {PRINT BLANK SUPPRESS}	Turns on the automatic incremented counter.
MODE NCOLUMN	Turns off the column display (default).
MODE NCOUNT	Suspends counter incrementing (default).
MODE NUMBER	Displays line numbers in front of printed line.
MODE NNUMBER	Turns off the line number display (default).

MODE PRALL	Prints lower case characters if device has that capability.
MODE PRUPPER	Prints all characters as upper case. Precedes lower case characters with an ^L and precedes upper case characters with an ^U if the device is upper case only.
MODE PROMPT	Prints prompt characters for INPUT & EDIT modes.
MODE NPROMPT	Stops printing of INPUT and EDIT prompt characters (default).
MODIFY/string-1/string-2/[n]=[G] {G} {n}	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 / }	Moves one line of text from buffer-2 into buffer-1. Buffer names are STRA, STRB, STRC, INLIN and EDLIN.
• NEXT [n]	Moves the pointer n lines forward or backward (default n=1).
NFIND string	Moves pointer down to first line NOT beginning with string.
NFIND(n) string	Moves pointer down to first 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. Use ! to delete existing characters.
PAUSE	Returns to operating system without changing the Editor state.
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 without filing text.

RETYPE string — The current line is replaced 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 — Unloads 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. See MOVE.

*[n] Repeat symbol. Causes preceding command to be repeated n times as in:

F /;D;N;*10
;D; *10

which deletes the next ten lines beginning with / . If n is omitted, the command repeats until the bottom of file is reached.

LISTING PROGRAMS

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 the FORTRAN programmers are:

-FTN Causes the FORTRAN output conventions to control the line printer when printing a file. These control characters are discussed in Section 16 under Formatted Printer Control.

-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 (13:05) or 12-hour format (9:25 PM).

The -FTN and -LNUM options are incompatible.

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 FILES

Renaming

Files may be renamed with the PRIMOS-level command:

CNAME oldname newname

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

Deleting

Files may 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.

CAUTION

Do not use the DELETE command to delete a UFD, subUFD, or segmented runfile (see Section 6).

SECTION 4

COMPILING

INTRODUCTION

Prime's FORTRAN IV Compiler, a one-pass compiler, produces highly optimized code and is supported by extensive function and subroutine libraries to do file-handling, and both mathematical and logical operations.

Source programs must meet the requirements of Prime FORTRAN IV as specified in this manual.

The compiler generates object code for either the R-identity or V-identity. R-identity code is loaded with Prime's Linking Loader (LOAD), described in Section 5; V-identity code is loaded with Prime's segmented-addressing utility (SEG), described in Section 6. Segmented-addressing code can be executed on Prime 400 (or higher) computers.

USING THE COMPILER

The FORTRAN Compiler is invoked by the FTN command to PRIMOS:

```
FTN treename [-parameter-1] [-parameter-2]...[-parameter-n]

                        or

FTN [-parameter-1] -I treename...[-parameter-n]
```

treename	is the treename of the FORTRAN source program file.
parameter-1, etc	are the mnemonics for the options controlling compiler functions such as I/O device specification, listings, and others.

All mnemonic parameters must be preceded by a dash "-". The name of the source program file must be specified either as the first expression following FTN or as -I treename (alternatively, -S treename) but not both.

Examples:

```
FTN TEST1 -XREFL -64V -LISTING SPOOL

                    or

FTN -LISTING SPOOL -XREFL -INPUT TEST1 -64V
```

are equivalent.

The meanings of the parameters will be discussed later in this section.

END OF COMPILATION MESSAGE

After the compiler has completed a pass of the specified input file, and generated object code and listing output to the devices specified by the parameter list, it prints one End of Compilation message at the user's terminal after each END statement encountered.

The format of the compiler message is:

```
xxxx ERRORS [<yyyyyy>FTN-REVzz.z]
```

xxxx — is the number of compilation errors; ØØØØ indicates a successful compilation.

yyyyyy — is: .MAIN. for a main program,

.DATA. for a BLOCK DATA subprogram,

the program entry name (up to 6 characters) for a subroutine or function.

zz.z — is the PRIMOS revision number.

Example:

```
ØØØØ ERRORS [<.MAIN.>FTN-REV14.Ø]
```

indicates the successful compilation of a main FORTRAN program by the REV.14 Compiler.

After compilation of all routines in the source file, control returns to PRIMOS.

COMPILER ERROR MESSAGES

The general format of the error message is:

```
**** LINE nnnn [context] name - message
```

nnnn — is the source line number that the statement in error started on.

All lines read from an insert file have the same source line number as the line with the $INSERT command on it.

If an error is detected in an EQUIVALENCE statement, the word 'EQUIVALENCE' is substituted for 'LINE nnnn'.

context — Context consists of the last 1-10 nonblank characters processed by the compiler before detecting the error. This field can be used to isolate the position in the statement that error occurs.

name — If the error is directly related to the misuse of a specific name, that name will be included in the error message. Otherwise, the field will be omitted.

message — A message up to 20 characters in length describing the error. A list of all messages is given in Appendix A.

Example:

```
**** LINE 0010 [WRUT] UNRECOGNIZED STMT
```

Note that the name field has been omitted.

COMPILER PARAMETERS

Normally, the source file will be stored in the disk file system, the binary (object) file will be created on the disk, and the listing file (if any) will be created either on the disk, at the user's terminal, or spooled directly to the line printer. In these cases, all instructions to the compiler are given by mnemonics in the FTN command line.

The A- and B-register settings are the instructions to the FORTRAN compiler (set at compilation time) telling it which functions and modes are to be enabled, and specifying the I/O. Using the mnemonic parameters establishes the values of these registers for the user automatically. Most users will have no need to set the octal values in these registers explicitly.

It is possible for a user to employ other peripheral devices (paper tape punch/reader, card punch/reader, magnetic tape) for making source, listing, or binary files. It would generally be preferable to bring the source program onto the disk, compile using the parameter mnemonics, and then transfer the listing and/or binary files to the desired device using PRIMOS commands. If for some reason this is not possible, the user may explicitly set the A- and B-register values to allow direct access to and from these devices. The previous method of specifying compiler options (by setting the A- and B-register values explicitly) is still valid with the new compiler. This means existing command files which set the A- and B-registers need not be changed. (See Section 18).

Compiler Functions

The compiler functions enabled by the mnemonic parameters may be considered to fall into four groups (Table 4-1).

- Specify Input/Output Devices
- Enable Listings/Cross References
- Memory Usage
- Operations

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

Table 4-1. Compiler Parameter Mnemonics
(● indicates Prime-supplied defaults)

Specify Input/Output Devices

●	BINARY	Specify binary (object) file
●	INPUT	Specify source program file
	LISTING	Specify listing file
●	SOURCE	Specify source file (same as INPUT)

Enable Listings/Cross References

	ERRLIST	Print error-only listing
●	ERRTTY	Print error messages at user terminal
	EXPLIST	Print full listing
●	LIST	Print source program and error listing
	NOERRTTY	Suppress error messages to terminal
●	NOTRACE	Suppress global trace
●	NOXREF	Suppress cross-reference listing
	TRACE	Enable global trace
	XREFL	Print full cross-reference listing
	XREFS	Print partial cross-reference listing

Memory Usage

	BIG	Handle arrays spanning segment boundaries (64V only)
	DEBASE	Conserve Loader base areas
	DYNM	Enable dynamic allocation of local storage (64V only)
●	NOBIG	No arrays spanning segment boundaries
●	SAVE	Static allocation of local storage
●	32R	32K words of relative-addressed user space
	64R	64K words of relative-addressed user space
	64V	15 (or 31) x 64K words of segmented user space

Operations

	DCLVAR	Flag undeclared variables
●	FP	Generate floating-point skip instructions
	INTL	INTEGER default is INTEGER*4 (long)
●	INTS	INTEGER default is INTEGER*2 (short)
●	NODCLVAR	Do not flag undeclared variables
	NOFP	Suppress generation of floating-point skip instructions
	SPO	Special library compilation

Specify Input/Output Devices

These parameters 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. (alternatively -SOURCE) (example: -I TEST or -S TEST)
-I treename	The source program filename 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 BTEST)
-B NO	No binary file will be created. This might be chosen if only the listing file were desired at earlier stages of program development.
-B YES	The binary file is created with the default name B-filename, where filename is the name of the source program file in the UFD in which the source program file resides. The binary file, however, is created in the UFD to which the user is attached when invoking the compiler. If the BINARY parameter is not included in the command line parameter list, 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.

If this parameter is not included in the command line parameter list, it is equivalent to -L NO.

Enable Listings/Cross References

These parameters enable or suppress program listings, error listings, and cross-reference listings (concordances). In all cases except ERRTTY (defined below) the enabling has no effect unless an output device or file is specified by the -L parameter.

The program-, error-, and cross-reference listings discussed below are generated for the following FORTRAN program example, POOH:

```
OK, SLIST POOH
GO
 310  X=48
      B=I*5
      C=5-I
      I=3
 20   GO TO (100,310,320), I
 320  A=B + C
      I=1
      GO TO 20
 100  Y=A*X
      WRUTE (1,110) X
 110  FROMAT (I5)
      CALL EXIT
      END
```

In all the cases that follow the usual default error messages are suppressed by including NOERRTTY in the parameter list to avoid duplication since the listing device is the user's terminal.

Three errors will be found in this program:

1. The unrecognized statement WRUTE (1,110) X, where WRITE has been misspelled.

2. The unrecognized statement 110 FROMAT (I5), where R and O have been interchanged.

3. Statement 110 has an error in it and consequently there is no label 110. This will generate an undefined statement number error.

ERRTTY/NOERRTTY

ERRTTY, which is the default, prints error messages at the user's terminal. This feature may be suppressed by including NOERRTTY in the parameter list.

In these examples, the error total is printed twice: as the last statement of the listing, and in the compiler message to the user, which is always printed at the user's terminal after compilation.

The first line of the program is printed at the top. The system printing routine does this for all files assuming that the first line of a file is to be treated as a header.

LIST/ERRLIST/EXPLIST

These are mutually exclusive parameters; each creates a type of listing in the listing file/device. These parameters override the program statements LIST, FULL LIST, and NO LIST.

ERRLIST - prints only the error messages on the listing device/file.

```
OK, FTN POOH -L TTY -NOERRTTY -ERRLIST
GO
  310 X=48
**** LINE 0010  [ WRUT ] UNRECOGNIZED STMT
**** LINE 0011  [ FROM ] UNRECOGNIZED STMT
**** LINE 0011  [ END ] _110 - UNDEFINED STMT NO.
0003 ERRORS [<.MAIN.>FTN-REV14.0]
0003 ERRORS [<.MAIN.>FTN-REV14.0]
```

LIST - prints the source program with line numbers, and the error messages. This is the default condition (if a listing file/device is specified).

```
  OK, FTN POOH -L TTY -NOERRTTY -LIST
  GO
  310 X=48
(0001)  310  X=48
(0002)       B=I*5
(0003)       C=5-I
(0004)       I=3
(0005)  20   GO TO (100,310,320),I
(0006)  320  A=B + C
(0007)       I=1
(0008)       GO TO 20
(0009)  100  Y=A*X
(0010)       WRUTE (1,110) X
**** LINE 0010  [ WRUT ] UNRECOGNIZED STMT
(0011)  110 FROMAT (I5)
**** LINE 0011  [ FROM ] UNRECOGNIZED STMT
(0012)       CALL EXIT
```

```
(0013)       END
**** LINE 0011 [ END ] _110 - UNDEFINED STMT NO.
0003 ERRORS [<.MAIN.>FTN-REV14.0]
0003 ERRORS [<.MAIN.>FTN-REV14.0]
```

EXPLIST - prints the full listing: the source program, with line numbers, the Prime Macro Assembler (PMA) code generated by the FORTRAN statements and the error messages.

```
OK, FTN POOH -L TTY -NOERRTTY -EXPLIST
GO
  310  X=48
(0001)   310  X=48
        000000:  ELM
        000001:  JMP   000000
        000001:  LINK  000001
(0002)        B=I*5
(0003)        C=5-1
(0004)        I=3
(0005)   20   GO TO (100,310,320),I
        000001:  FLD   =24756
        000003:  FST   X
        000005:  LDA   I
        000006:  MPY   =5
           .
           .
           .
        000020:  LDA   =3
        000021:  STA   I
(0006)   320  A=B + C
        000022:  LDA   I
        000023:  JST   F$CG
        000024:  OCT   000004
        000025:  DAC   _100
        000026:  DAC   _310
        000027:  DAC   _320
        000030:  LINK  _320
(0007)        I=1
(0008)        GO TO 20
(0009)   100  Y=A*X
        000030:  FLD   C
        000032:  FAD   B
        000034:  FST   A
        000036:  LT
        000037:  STA   I
        000040:  JMP   _20
        000041:  LINK  _100
(0010)        WRUTE (1,110) X
**** LINE 0010  [ WRUT ]  UNRECOGNIZED STMT
(0011)   110  FROMAT (I5)
**** LINE 0011  [ FROM ]  UNRECOGNIZED STMT
(0012)        CALL EXIT
(0013)        END
        000041:  JST   EXIT
        000042:  LINK  A
        000042:  OCT   000000
        000043:  OCT   000000
        000044:  LINK  B
        000044:  OCT   000000
           .
           .
           .
```

```
        000055:  LINK  24576
        000055:  OCT   060000
        000056:  OCT   000206
        00004l:  DAC   _100
**** LINE 0011  [ END ]  _110 - UNDEFINED STMT NO.
        000022:  DAC   _20
        000001:  DAC   _310
        000030:  DAC   _320
0003 ERRORS [<.MAIN.>FTN-REV14.0]
0003 ERRORS [<.MAIN.>FTN-REV14.0]
```

NOXREF/XREFL/XREFS

NOXREF is the default. XREFS and XREFL generate concordances (cross-references); they are mutually exclusive in the parameter list. XREFS appends a partial concordance to the end of the listing in the listing file/device; XREFL appends a complete concordance. Concordances are cross-reference tables between program symbols, their line numbers and storage locations in memory. In the partial concordance, symbols referenced only in specification statements are not included. This is useful if there are COMMON blocks with many variables of which only a few are used in the particular program unit being compiled. The default condition, which is no concordance can be obtained by not specifying any cross reference parameter or by including NOXREF in the parameter list.

An example of the concordance is:

```
OK, FTN POOH -L TTY -NOERRTTY -XREFS
GO
  310  X=48
(0001)   310  X=48
(0002)        B=I*5
(0003)        C=5-I
(0004)        I=3
(0005)   20   GO TO (100,310,320),I
(0006)   320  A=B + C
(0007)        I=1
(0008)        GO TO 20
(0009)   100  Y=A*X
(0010)        WRUTE (1,110) X
**** LINE 0010  [ WRUT ]  UNRECOGNIZED STMT
(0011)   110  FROMAT (I5)
**** LINE 0011  [ FROM ]  UNRECOGNIZED STMT
(0012)        CALL EXIT
(0013)        END
**** LINE 0011  [ END ]  _110 -UNDEFINED STMT NO.
```

```
A       R           000042   0006M   0009
B       R           000044   0002M   0006
C       R           000046   0003M   0006
EXIT    R EXTERNAL  000000   0012
I       I           000050   0002    0003   0004M  0005   0007M
X       R           000051   0001M   0009
Y       R           000000   0009M

_100                000041   0005    0009D
_110                000000   0011
_20                 000022   0005D   0008
_310                000001   0001D   0005
_320                000030   0005    0006D

          0003 ERRORS [<.MAIN.>FTN-REV14.0]
          0003 ERRORS [<.MAIN.>FTN-REV14.0]
```

The first column is the symbol, the second is the data mode (R for real, I for integer, etc.). The first numerical column is the storage address, the following numbers are line numbers of the statements in which the symbols appear. If a symbol is modified (appears on the left hand side of the = sign) the letter M is suffixed. The letter D suffix for statement label line numbers identifies the line number at which that statement label is defined. A complete list of data mode codes and line number suffixes appears in Table 4-2.

NOTRACE/TRACE

NOTRACE is the default. The TRACE mnemonic produces a trace for each variable in the program. This parameter takes precedence over any TRACE statement within the source program.

At object program run time (see Section 7), any trace coding inserted by the compiler causes a line to be typed consisting of a variable name, an array name, or a statement number, followed by an equals sign, followed by the current decimal value assigned to that name. The decimal value is typed in INTEGER, FLOATING POINT, or COMPLEX format.

Example: a FORTRAN program PRIME has been written to print a list of prime numbers between 2 and 50. The program will be compiled with the TRACE parameter (the default binary file name B_PRIME is used). After the program has been successfully compiled it will be loaded and executed using the Prime Linking Loader. (See Section 5 for an explanation of this.) Sample lines of TRACE information as typed at object run-time are shown.

```
OK, FTN PRIME -TRACE
GO
0000 ERRORS [<.MAIN.>FTN-REV14.0]
GO,
```

Table 4-2. Concordance Codes

Code	Data Mode (second concordance column)
A	ASCII
C	COMPLEX
D	DOUBLE PRECISION (REAL*8)
I	SHORT INTEGER (INTEGER*2)
J	LONG INTEGER (INTEGER*4)
L	LOGICAL
R	REAL (REAL*4) - single precisions

Line Number Suffixes

A	Symbol is contained in the argument list of a function or subroutine.
D	Symbol is defined at this line number (statement label).
I	Symbol is initialized at this line (DATA statement).
M	Symbol is modified (left hand side of assignment statement).
S	Symbol is in a data mode specification statement.

```
OK, LOAD
GO
$ LO B_PRIME
$ LI
LC
$ SA *PRIME
$ EX
FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2Ø TO 5Ø
                  2
                  3
                  5
                  7
K=       3
(2)
                 11
(4)
K=       3
(2)
                 13
 .
 .
 .
K=       6
(4)
K=       6
(2)
(2)
                 47
(4)
K=       7
(2)
(2)
(4)
THIS IS THE END OF THE LIST

****ST

OK,
```

Memory Usage

32R/64R/64V

32R mode is the default. The compiler modes 32R, 64R, and 64V are mutually exclusive. They cause the compiler to generate object code suitable for operations in a user address space of 32K words (relative-address), 64K words (relative-address) and 15x64K or 31x64K words (segmented-address) respectively.

NOBIG/BIG

In the 64V mode, arrays which exceed 64K words of memory must be handled by inserting BIG into the parameter list. BIG forces the 64V mode and thus cannot be used in the 32R or 64R modes. A 64V mode program which does not have arrays which span segment boundaries may be compiled with BIG. It will compile, load, and execute properly, although slower than if it had not been compiled with BIG.

NOBIG is the default parameter (see Secton 12 for details on large arrays).

SAVE/DYNM

In the 64V mode, the inclusion of DYNM in the parameter list enables dynamic allocation of local storage. This allows the use of recursive subroutines (subroutines which call themselves). DYNM forces the 64V mode and thus cannot be used in the 32R and 64R modes. If recursive subroutines are used, DYNM is mandatory.

The default parameter is SAVE which enables static local storage allocation. Static storage allocation is the only method used in the 32R and 64R modes.

DEBASE

Conserves Loader base areas. This parameter may be included for programs compiled in 32R or 64R mode it should not be used for programs compiled in 64V mode.

The default is obtained by omitting DEBASE from the parameter list. (See the LOAD Section 5 for explanation of base areas.)

Operations

NODCLVAR/DCLVAR

Flags variables which have not been declared in specification statements. NODCLVAR is the default.

FP/NOFP

Suppress generation of floating-point skip operation. FP is the default. The compiler will normally generate instructions from the floating point skip set when testing the result of a floating-point operation. If machine does not have the floating-point hardware, suppressing these instructions will speed up execution.

SPO

System Program Optimization. (This also forces DCLVAR.) Generates code in a special library compilation mode. Certain errors are not flagged and some statements are interpreted differently then usual. NOT recommended for general users. The default is the normal compilation mode(s).

INTS/INTL

The Prime FORTRAN system has both Long (INTEGER*4) and Short (INTEGER*2) integers. In the default (or INTS) condition the INTEGER statement in a program is taken to be INTEGER*2. If INTL is included in the parameter list then the INTEGER statement is taken to be INTEGER*4. This parameter eases the conversion of existing programs to the Prime FORTRAN System.

A complete list of all parameters with more detailed comments on the consequences of their usage will be found in the reference section (Section 18).

Prohibited Parameter Combinations

The following combinations of parameters should not be used in a command line:

Parameter Used	Conflicting Parameter(s)
<parameter>	NO<parameter>
NO<parameter>	<parameter>
BIG	32R or 64R
DEBASE	BIG, DYNM, 64V
DYNM	NOBIG, SAVE, 32R, or 64R
ERRLIST	EXPLIST or LIST
EXPLIST	ERRLIST or LIST
INTL	INTS
INTS	INTL
LIST	ERRLIST or EXPLIST
NOBIG	DYNM or 64V
NODCLVAR	SPO
NOXREF	XREFL or XREFS
SAVE	DYNM
SPO	NODCLVAR
XREFL	NOXREFS or XREFS
XREFS	NOXREFS or XREFL
32R	BIG, DYNM, 64R, or 64V
64R	BIG, DYNM, 32R, or 64V
64V	DEBASE, NOBIG, 32R, or 64R

The command line is parsed from left to right. Thus, the right-most mnemonics take precedence over those to the left of them. Using the prohibited combinations above will yield diverse results depending upon

the specific case. In almost all cases, the result will be undesirable.

SECTION 5

LOADING AND LINKING

INTRODUCTION

The Prime Linking Loader utility (LOAD) operates on code produced by the FORTRAN compiler (FTN) in the 32R (default) or 64R modes; code produced in the 64V (segmented addressing) mode should be processed by the SEG utility (Section 6).

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

Prime's Linking Loader offers the following features:

- The loader is capable of loading code and COMMON anywhere in 64K, above or below itself (but not on top of itself or its symbol table!). (LOAD)
- The location of COMMON is movable 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 Ø 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 has the convenience of executing 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 locations ´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 and 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 5-1.)

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

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:

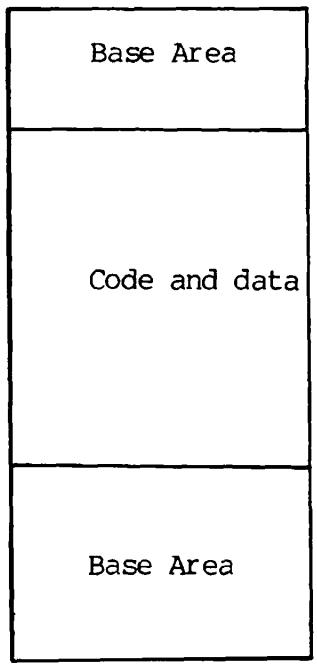
```
FILMEM

  or

FILMEM ALL
```

and has the result below:

Command	Operating System	
	PRIMOS II	PRIMOS III, IV, V
FILMEM	clears locations ´100 to ´47777 except those occupied by PRIMOS II	clears locations ´100 to ´77777
FILMEM ALL	clear all user space except locations occupied by PRIMOS II	clears locations ´100 to ´177777



Initial location of *PBRK

Location of *PBRK at end of load

Location of *PBRK for start of next load

Figure 5-1
*PBRK locations before and after
loading an object module.

USING THE LOADER UNDER PRIMOS

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. It is sometimes desireable to relocate the loader, thus allowing initialized COMMON to occupy locations ´60000 to ´63777. In this case, the loader should be invoked by the PRIMOS command:

```
HILOAD
```

which 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.

Note

> HILOAD may be used even if the FORTRAN program has been compiled in 32R mode and will be loaded in 32R mode (see MODE).

All loader functions are available through user terminal keyboard commands. When the LOAD 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 executing a command successfully, the loader types the $ prompt character. If the load is complete (i.e., there are no missing routines or modules), the loader will type the message LC indicating that all external references have been satisfied. (However, LC does not imply that all UII requirements have been satisfied (see MAP and HARDWARE).

Example:

```
OK, LOAD                    invoke loader
GO
$ LO B↑TEST                 load object program
$ LI                        load FORTRAN library
LC                          load is complete
$                           ready for next command
```

If an error occurs in the Loader itself during an operation, the Loader prints a two-letter error code, then the $ prompt character. Loader error messages and suggested handling techniques are discussed immediately following the section on commonly-used Loader commands and in Appendix A. Most of the errors encountered are caused by large programs where the user is not making full use of the Loader capabilities.

When a system error (FILE NOT FOUND, NO SUCH UFD, NO ACCESS RIGHTS, etc.) is encountered, the loader prints this system error and returns its prompt symbol ($).

Command Files

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. No spaces may appear in a comment line other than the one directly after the asterisk. (See Example) A comment line is not processed by the Loader.

Example of a Command File:

```
* COMMAND.FILE.TO.LOAD.THE.LOADER
FILMEM
* INVOKE.OLD.LOADER.
LO B↑LOAD 174000
SA HILOAD
* NOW.USE.NEW.TO.CREATE.NEW.LOAD
EX
* NOW.WE.ARE.IN.HILOAD
LO B↑LOAD 60000
SA LOAD
QU
```

Command Formats

Each Loader command consists of a command name followed by a series of arguments:

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

where COMMAND is the command name, each name is a text string which may be a PRIMOS filename or UFD name, and each arg is an octal argument (numeric only) 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. 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). The Kill (?) and Erase (") character functions are supported in the command line.

A complete list of the LOAD commands is given below. (Underlines indicate minimum required abbreviation.)

Command	Function
ATTACH	Attach to different UFD
AUTOMATIC	Automatic generation of base areas
COMMON	Relocate COMMON address
EXECUTE	Initiate direct program execution in Loader
FORCELOAD	Forceload first module in object file
F/	Forceload all modules in an object file
HARDWARE	Hardware definition
INITIALIZE	Reinitialization
LIBRARY	Load library object file (i.e., object files in UFD=LIB)
LOAD	Load object file
MAP	Generate Load state map
MODE	Select addressing mode
P/	Begin loading at next page boundary
QUIT	Return command to PRIMOS
SAVE	Save loaded memory image
SETBASE	Define a new linkage area
VIRTUALBASE	Relocate base sector
XPUNGE	Controls the deletion of symbols

Loader Command Categories

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

1. Commands the programmer uses quite often:

   LOAD
   LIBRARY
   MAP
   SAVE
   EXECUTE
   QUIT

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

   FORCELOAD
   F/
   AUTOMATIC
   SETBASE
   COMMON
   MODE
   HARDWARE
   INITIALIZE
   ATTACH

3. Commands designed for the use of the systems programmer. These are normally of very little use to the applications programmer.

   P/
   VIRTUALBASE
   XPUNGE

FREQUENTLY USED LOADER COMMANDS

See Table 5-1 for references to Load State Parameters as displayed in the load map.

LOAD filename [loadpoint] [base-start] [base-range] (Format 1)

LOAD filename * prebase (Format 2)

Format 1: Loads the specified object file into memory.

filename	is the object file to be loaded.
loadpoint	is the starting address at which the file is to be loaded; default is the current *PBRK.
base-start	defines a base area starting address; default is ´200.
base-range	number of locations in the base area; default is ´600.

Format 2: Loads the specified object file and defines base areas before and/or after the file (see Figure 5-1). The current *PBRK is used as the first location for this operation.

filename	is the object file to be loaded.
prebase	is the length of the base area to precede the object file; it may be zero.

Do not specify more than 2 numeric parameters in Format 2 . The results are unpredictable.

Notes

1. If all symbols in the load module have been previously defined, the loader skips the module. A load module is defined to terminate with an "END" statement. For example, the user defines a new SQRT which is more accurate in the specific range of application. This module is loaded, the library´s SQRT function is not loaded. It is possible to forceload modules in which all symbols have been previously defined. This is discussed under the less frequently used commands.

2. The compiler converts the program to binary format, creating a new file with a new name (e.g., B←MUX). This binary version must be specified in the LOAD command.

Table 5-1. Load State Definition

Parameter	Definition	Value at Start of Load (octal)
*LOW	The lowest location in memory loaded	177777
*HIGH	The highest location in memory loaded	Ø
*START	The location at which execution will begin	Ø
*PBRK	The next location in memory to be loaded	1ØØØ
*CMLOW	The lowest location in COMMON	XX777
*CMHGH	The highest location in COMMON	XX777
*SYM	The lowest location used by the symbol table	YYØØØ
*UII	The net hardware/UII package requirement (see HARDWARE command for meaning)	Ø

NOTE:

XX = Last Sector Occupied by Loader

YY = First Sector Occupied by Loader

Example:

A FORTRAN program called MUX when compiled would generate a binary file with a default name of B<-MUX. The programmer loads this program as follows:

```
OK, LOAD
$ LO B<-MUX
$ LI
LC
$
```

LIBRARY [filename] [loadpoint]

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

filename — is the library file to be loaded; if omitted the FORTRAN library FTNLIB is loaded.

loadpoint — is the starting address for loading; if omitted the current *PBRK is used.

MAP [filename] [option]

filename — is the map file to be opened; if omitted the map will be printed at the terminal.

option — specifies type of map to be generated.

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
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:

- The lowest and the highest storage memory locations
- The location at which the program execution begins
- The next location available for loading
- The high and low COMMON area
- The lowest location used by the symbol table
- The net hardware UII package requirement

These eight parameters are printed in the load state map with a corresponding storage address. (See Table 5-1.)

Example:

```
OK, LOAD
GO
$ LO B←SIMP
$ LI
LC
```

MAP Option 1 - Load State Map

```
$ MA 1
*START  001000  *LOW    000200  *HIGH   006512  *PBRK   006513
*CMLOW  063777  *CMHGH  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:

```
$ MA 2
*START  001000  *LOW    000200  *HIGH   006512  *PBRK   006513
*CMLOW  063777  *CMHGH  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.

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 the special FORTRAN COMMON block LIST, defined as starting at location 1.

Example:

```
$ MA
*START  001000  *LOW    000200  *HIGH   006512  *PBRK   006513
*CMLOW  063777  *CMHGH  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

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$AT1   005174  WATBL   005237
LUTBL   005256  PUTBL   005313  RSTBL   005350  O$AD07  005405
```

Writing Map to File

Load maps may be sent to a file instead of the user's terminal. The following 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                  load FORTRAN library
LC
$MA MAP1 1            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  *CMHGH  063777  *SYM    057374  *UII    000001
```

SAVE filename [aregister] [bregister] [xregister]

Saves the loaded memory image from *LOW to *HIGH, including all initialized COMMON areas, in the current UFD. Also saved with the program are the low, high, start, and keys parameters obtained from the Loader (there is no option to set them).

filename — is the name of the file in which the memory image is to be saved.

a-register — initial value of A register.

b-register — initial value of B register.

x-register — initial value of X register.

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 however.

EXECUTE [aregister] [bregister] [xregister]

Enables the user to start execution of the loaded program with optional values preset into the A, B, and X registers. Execution starts at the location specified by the *START entry of the load map.

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. If the Loader has opened a MAP file, it is closed at this time.

Loader Error Messages

The commonly occurring Loader error messages with suggested remedies are listed below. A complete list is found in Appendix A.

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. Make sure that object module 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 op-code with an address of zero. Loading continues. Object code may be in 64V mode; recompile and then restart load.
MO	Memory Overflow Errors. As users' programs become larger, MO (memory overflow) errors become more common. This section contains a description of the several typical causes of these errors and suggested solutions to these cause. When MO error occurs, the user should do a 'MA 2' and examine the map for any of the following possible situations: 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. The sector zero base area is full - the next free location is '1000. The size of the sector zero base may be increased by a SETB 100 command at the beginning of the load - if locations '100 to '200 are free - or an AU command may be used to insert base areas throughout the load. Alternatively, recompile using the DEBASE option. (See AUTOMATIC, SETBASE). c. *CMLOW is near *PBRK. COMMON should be moved to higher memory using the COmmon command. Re-initialize using the FILMEM command. If COMMON must be moved above

´100000, it may be necessary to recompile the program in 64R mode and the program load must begin with a MO D64R command. (See COMMON, MODE).

d. 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 6).

e. None of the above. The user´s program requires initialized COMMON. COMMON is usually defaulted to overwrite the space used by the Loader. Those locations between the bottom of the symbol table and the top of the Loader cannot be initialized as this would destroy the Loader. The solution is to use a COmmon command to move COMMON out of the way of the Loader. Possibly the user will want to use HILOAD to permit COMMON to use the locations normally used by the Loader. (See COMMON.)

OR Out of Reach. An attempt has been made to reference a COMMON area that is out of reach of the load mode.

Begin the load with an MO D64R command, or move COMMON to ´100000 or lower with the CO command. (See COMMON, MODE)

LESS FREQUENTLY USED LOADER COMMANDS

These commands are generally used for one or more of these reasons:

- solving a specific problem in loading a program (see Loader error messages)
- optimizing the loading of a program
- portability between different levels of Prime Computers
- added convenience to the programmer

Forceloading

Forceloading causes a module to be loaded even if all the symbols in that module have been previously defined. This operation is useful in building systems or program templates. There are two commands for forceloading.

FORCELOAD filename [load-options]

Forceloads the first module in filename.

filename	is the file to be forceloaded
load-options	specify where the module is to be forceloaded; these options are the same as the LOAD command options.

F/xx filename [load-options]

Forceloads all modules in the file specified.

xx	is one of the load commands: LO, LI, or FO. If the LI command is used and filename is omitted, then all of the FORTRAN library file, FTNLIB, will be forceloaded.
filename	is the file to be forceloaded. It may be omitted with the LI command to forceload the FORTRAN library.
load-options	specify where the file is to be forceloaded. These are the same as the LOAD command options.

AUTOMATIC base-length

Causes the Loader to insert a base area of length base-length words 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 base-length may be changed between load files. This automatic feature is turned off with an AU 0 command.

AUTOMATIC helps to reduce the number of loads which are addressed directly from the Sector zero base area by instructing the Loader to insert base areas automatically.

SETBASE base-start [base-range]

or

SETBASE * base-range

Defines a base area that begins at base-start and includes the number of locations specified by base-range. If the range is not specified, the end of the area is location '777 of the sector containing the Base-start location. Multiple Base areas are allowed. A command to create a base linkage area overlapping a previously defined area is ignored.

The command SE * creates a base area of the specified length to be inserted at the current location. Thus, if *PBRK (base-start) is 1765, the command SE * 20 creates a setbase area of length 20 at 1765 and the *PBRK set at 2005 after the command has been executed.

The user may increase the size of the sector zero base area by the command:

SE base-start

at the beginning of the load session.

Example:

SE 125 lowers the start of the sector zero base area to location '125.

CAUTION

The start of the sector zero base area must not be set lower than '100.

Default values for the sector zero base area are:

base-start '200
base-range '600

COMMON address

Moves the top or starting location of COMMON to the address specified. Space for COMMON items is allocated downward from but not including the

specified address. If COMMON is to be moved, this should be done before loading any object modules.

The top of COMMON is the highest location used for COMMON by the Loader. The default COMMON load address is the last location in the last Loader sector. This means, for example, that the top of COMMON for LOAD is ´63777 (for HILOAD, it is ´177777).

Note

To specify a COMMON load point, (top or starting location) give the location desired + 1. For example, CO 40000 specifies ´37777 as the top location in COMMON. This is for compatibility with previous releases of the Loader.

MODE parameter

Directs the Loader to desector the FORTRAN load module in one of the CPU addressing modes:

Parameter	Addressing Mode
D32R	32K Relative (default value)
D64R	64K Relative

The mode command is used when an addressing mode other than 32K relative is required.

Programs compiled in 32R mode must be loaded in D32R mode; programs compiled in 64R mode may be loaded in D32R or D64R mode. The D64R mode would be used if 32K words of user address space are not adequate.

The MODE command, when used, should precede any other command.

HARDWARE definition

Specifies the level of instruction execution hardware existing on the CPU whereon the program is intended to be run. For increased speed, some operations (as floating point and high-speed arithmetic) are implemented by hardware on the higher-level Prime computers. If such operations are attempted on lower-level Prime computers, a UII (Unimplemented Instruction Interrupt) occurs and control is transfered to the appropriate UII routine. This routine simulates the missing hardware via software routines.

The FORTRAN compiler (FTN) outputs an object group informing the Loader of the need, if any, for such hardware-implemented operations in any given program module. The user may determine if such operations will be needed by examining the value of *UII in a Load Map. If the value is 0 (for a complete load), then it is not necessary to load the UII

library. If UII requirements exist, all or part of the UII library must be loaded. The user minimizes the portion of the library loaded by specifying the execution hardware of the machine to the Loader by the HARDWARE command. The definition parameters are:

CPU	definition
P400	57
P300/FP	17
P300	3
P200/HSA	1
P100/HSA	1
P200	0
P100	0

FP: with optional floating-point. HSA: with optional high-speed arithmetic.

The hardware must be specified prior to loading the UII library. The UII library must be the last module loaded before the program memory image is SAVEd.

The UII library is loaded by the command LI UII.

INITIALIZE [filename] [load-options]

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 5-1) return to their initial values. If no filename is provided, the Loader repeats its prompt character ($).

load-options Refer to the loadpoint, base-start and base-range options available under the LOAD command. This allows the programmer to restart a LOAD session without the necessity of returning to the PRIMOS level and re-invoking the Loader.

ATTACH [ufd-name] [password] [ldisk] [key]

Attaches to different UFD's. This command is converted into a CALL to the PRIMOS subroutine ATCH$$ and has exactly the same effect.

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

password The user gets owner or nonowner status according to the password given. The password parameter is necessary only when the UFD is password-protected.

ldisk — If the ldisk parameter is omitted, the Loader searches only device Ø for the specified UFD. If an Ldisk value of ´100000 is specified, the file system searches all started devices in logical unit order.

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

- Ø Do not change home UFD.
- 1 Adopt named UFD as home UFD.
- 2 Attach to a sub-UFD in the current UFD; do not set as home.
- 3 Attach to sub-UFD in the current UFD; set as home.

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.

SYSTEMS LEVEL COMMANDS

P/xx filename Ø [base-start base-range]

Begins loading of the first module in the file at the next page boundary. A page boundary is a location whose address is a multiple of ´1000.

xx — is one of the load commands: LO, LI, or FO.

filename — is the file to be loaded on a page boundary. If the load command is LI, then omitting filename causes the FORTRAN library, FTNLIB, to be loaded at the page boundary.

base-start — defines the base area starting address. The default value is ´200.

base-range — is the number of locations (octal) in the base area. Default is ´600.

VIRTUALBASE base-start to-sector

Copies the base sector into the corresponding locations of the to-sector. This command is used in building RTOS modules.

base-start	first location in the base sector to be copied. The base sector is copied from base-start to the base sector end address.
to-sector	the sector into which the base sector is to be copied.

XPUNGE dsymbols dbase

Deletes COMMON symbols, other defined symbols, and base areas.

dsymbols controls symbol deletion

dsymbols	operation
0	delete all symbols except undefined symbols
1	delete all symbols except undefined symbols and COMMON areas

dbase controls base area deletion

dbase	operation
0	delete all defined base areas
1	delete all defined base areas except sector zero
2	retain all defined base areas

SECTION 6

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 is a 64K word block of user's virtual address space. Segment '4000 is the segment that SEG and other external commands occupy when invoked. Segment '4000 is the lowest-valued non-shared segment in the PRIMOS system. SEG creates a run file 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 re-assigned until logout. Since only a fixed number of segments are available for all users, extra 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 done to relative-addressed runfiles at the PRIMOS command level or by the Linking Loader. Since the nature of SEG runfiles differs from that of relative-addressed runfiles, separate commands are needed.

SEGMENTED RUNFILES

A segmented runfile consists of segment subfiles in a segment directory. For this reason, you 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 to memory) which requires the name of the runfile before anything is loaded. The runfile may be new or may be a previously used SEG runfile, and may be in any UFD. A relative-addressed runfile may not 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

The 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 loadmap, 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 on their command line. These modifiers are never used in the basic SEG load sessions. In fact, usually 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 that 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

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

Code and data are loaded in separate segments to support re-entrant procedures. Data consists of all COMMON blocks and link frames. 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. (See Section 12.)

The Stack

The Loader assigns a stack (which is a dynamic work area) when SAVE or EXECUTE 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; locations 0 to 3 are used for internal SEG information. 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 filename-1 filename-2 par-1 par-2 par-3 ....
```

Where filename-1 is the filename (or treename) of the file to be accessed. Treenames enable 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 tree filename. 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 the SEG commands follows. Commands may be abbreviated to the underlined characters.

The code column specifies where the command is described in detail:

•	This section
E	Extended Functionality (Section 11)
S	Shared Procedure (Section 12)
blank	Intended for use at assembly language level or for in-house use (see The PMA Programmer´s Guide, PDR3059)

Code	Command	Function
•	DELETE	delete a SEG runfile
•	HELP	print a list of SEG commands at user's terminal
E	MAP	generate a loadmap
E	MODIFY(SAVE)	invoke modification sub-processor
E	NEW	write new copy of SEG runfile to disk
	PATCH	modify save range of existing segment
E	RETURN	return to SEG command level
E	SK	alter stack size and/or location
E	START	change program execution start address
	WRITE	rewrite all segments to disk (to preserve patches)
	PSD	invoke VPSD debugging utility
•	QUIT	return to PRIMOS command level
	RESTORE	bring SEG runfile into user memory
E	RESUME or RESUME	restore SEG runfile and begin execution
S	SHARE	create R-mode runfiles for segments below '4001
S	SINGLE	create R-mode file image of single segment
E	TIME	print time and date of last runfile modification
•	VLOAD (LOAD)	define runfile and invoke loader for creation
•	VLOAD * (LOAD *)	define runfile and invoke loader for appending
E	ATTACH	attach to another UFD
S	A/SYMBOL	define a symbol in memory and reserve space for it using absolute segment numbers
S	COMMON ABS	relocate Common using absolute segment numbers
E	COMMON REL	relocate Common using relative segment assignment
E	D/xx	perform load using previous parameters
•	EXECUTE	save load to disk and execute program
S	F/xx	forceload all routines in object file
S	IL	load the impure FORTRAN library
E	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
S	PL	load the pure FORTRAN library
	P/xx	load on a page boundary
•	QUIT	return to PRIMOS command level
S	RETURN	return to SEG command level
E	RL	reload a routine
E	R/SYMBOL	define a symbol in memory and reserve space for it using relative segment assignment
•	SAVE	save load to disk
S	SPLIT	break segment into data and procedure portions
E	STACK	change stack size
E	SYMBOL	define a symbol at a specific location in memory
S	S/xx	load a specific absolute segment
S	XP	expunge symbols from symbol table; delete base information

The addition of letters or using the command names in full will not affect SEG's operation and may be used if this aids the programmer.

Vestigial Commands

A number of commands exist whose functionality has been superseded, 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 the same functions 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.

Error Messages

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 A, along with probable causes of the errors and suggestions for correcting or eliminating them.

USING SEG

SEG is a command under CMDNCO; the FORTRAN 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 7 - 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, Section 11 and Section 12.

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., lines 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 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 TREE NAME:

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

The first time a SEG runfile name 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

HELP (SEG level)

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

VLOAD [filename] or LOAD [filename] (SEG level)

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 filename is the name of an existing SEG runfile, that runfile will be reinitialized before control is passed to the Loader.

To access existing runfiles, see SEG's VLOAD * command in Section 11.

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

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

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 otherwise.

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

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

Loader Subcommands

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. Other Loader commands, including extended forms of the commands below, are discussed later in Sections 11 and 12.

LOAD filename (Loader Subcommand)

Processes 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.

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 FORTRAN compiler with the 64V option. If filename is not given or is an incorrect type (not an object file), an error will be generated.

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 [filename] (Loader Subcommand)

Processes the library file in the same manner as LOAD processed object files. In most cases, any libraries needed are loaded after other object files.

filename is the name of the file in UFD=LIB which is to be loaded into the runfile.

The File Filename must be a File 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 (in that order) will be used.

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 in Sections 11 and 12.

MAP 3 (Loader Subcommand)

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 (Loader Subcommand)

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 (Loader Subcommand)

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

QUIT (Loader Subcommand)

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.

DELETE filename (format 1) (SEG level)

or

DELETE (format 2) (SEG level)

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

Note

Do not attempt to delete a SEG runfile with the PRIMOS level DELETE command. It will only 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.

QUIT (SEG level)

Returns the user to the PRIMOS command level.

EXAMPLE OF A LOAD

Assume that the user has compiled a main program, MAIN and a subroutine in a separate source file named SUBR. Both have been compiled in 64V mode using the default object file names. They could be loaded as follows:

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 the separately compiled subroutine
$ LI	load the FORTRAN libraries
LC	indicates all references are satisfied
$ SAVE	user saves the runfile (this is not strictly necessary as the EXECUTE command will SAVE the program)
$ EXECUTE	first attempt to execute the program
OK,	control returns to PRIMOS

SECTION 7

EXECUTING PROGRAMS

INTRODUCTION

This section treats the following topics:

- Execution of program memory images saved by the Linking Loader
- Execution of segmented runfiles saved by SEG's Loader
- Run-time error messages
- Installation of programs in the Command UFD (CMDNCØ)

PROGRAM MEMORY IMAGES SAVED BY THE LINKING LOADER (32R AND 64R MODES)

Execution is performed at the PRIMOS level using the RESUME command:

```
OK, R filename
```

where filename is the program in the current UFD to be executed.

Programs which have been made resident in the user's memory may be executed by a START command:

```
OK, S
```

Programs which have halted or been interrupted by the user may be restarted at the beginning by the START command as:

```
OK, S 1ØØØ
```

These two commands are discussed in detail below.

RESUME filename

RESUME brings the memory-image program filename from the disk into the user's memory, loads the initial register settings, and begins execution of the program.

Example:

```
OK, R *TEST          User requests program
GO                   execution begins
THIS IS A TEST       output of program

OK,                  PRIMOS requests next command
```

Note

RESUME should not be used for segmented (64V mode) programs; use the SEG command (discussed later) instead.

START [start-address]

If a program has been made resident in memory (e.g., by a previous RESUME command) START may be used to initialize the registers and begin execution.

START can also restart a program that has returned control to PRIMOS (for example, because of an error, a FORTRAN PAUSE or CALL EXIT statement). If START is typed without a value for start-address, the program resumes at the address value at which execution was interrupted. To restart the program at a different point, specify an octal starting location as the start-address value; the usual default value for the beginning of FORTRAN programs is 1000.

Example:

```
OK, R TEST1          Begin
GO                   execution starts
INPUT NEW KEY: 5     program asks for input
QUIT                 user hit CTRL/P to stop
OK, S 1000           restart program from beginning
GO                   execution restarted
INPUT NEW KEY:
```

The FORTRAN programmer will almost always use the default forms of the RESUME and START commands (the form discussed here). For a complete treatment of these commands see the PRIMOS INTERACTIVE USER GUIDE, MAN2602.

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

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

Execution is performed at the PRIMOS command 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 starts execution. 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 #TEST        user request program
GO                   execution begins
THIS IS A TEST       output of program

OK,                  PRIMOS requests next command
```

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

A SEG runfile may be restarted by the command:

S 1000

provided both the SEG runfile and the copy of SEG used to invoke it are in memory.

RUN-TIME ERROR MESSAGES

During program execution, error conditions may be generated and detected by the FORTRAN mathematical functions, file system subroutine calls, or the operating system. A list of run-time errors is included in Appendix A.

R-Mode FORTRAN Functions

FORTRAN functions (COS, SIN, etc.) used for programs compiled in the 32R and 64R mode generate error messges in this format:

****cc <n>

where cc is a two-letter code and <n> is the FORTRAN logical unit number; <n> is printed out only for I/O errors. When an error is encountered, the error message is printed at the user's terminal. Most errors return command to PRIMOS level.

V-Mode FORTRAN Functions

FORTRAN functions (COS, SIN, etc.) used for segmented (64V mode) programs generate error messages in this format:

**** error-message

Errors detected are generally of the same type as those in the R-mode functions; due to less restrictive program size constraints, error messages have been made clearer. Most errors return control to the PRIMOS level.

New File System Calls

In the new file system, subroutines return an integer error code as part of their argument list. A non-zero value indicates the type of error which has occurred. The error code value may be used to transfer control in the program. The error message can be printed to the terminal using the ERRPR$ subroutine. The error message format is:

standard-text user´s-text-if-any (name-if-any)

where:

standard text	is the file system standard error message (listed in Appendix A, Table A-4).
user´s-text-if-any	is an optional message which the user may elect to have printed
(name-if-any)	is the program/subsystem detecting or reporting the error. Again, the user selects this text.

Example:

Following a call to PRWF$$, CODE was returned as CODE=E$UNOP; the call:

```
CALL ERRPR$ (K$SRTN,CODE,´DO A STATUS´,11,´PRWF$$´, 6)
```

results in the message:

```
UNIT NOT OPEN. DO A STATUS (PRWF$$)
```

The new file system contains many improvements and enhancements of the old; the use of the new system is strongly urged. The new system is described in REFERENCE GUIDE, FILE MANAGEMENT SYSTEM (FMS), PDR 3110.

Note

The error code should always be checked for zero/non-zero value to ensure that errors do not go unnoticed.

In the list of standard error messages for new file calls, parentheses enclose a list of subroutines most likely to generate that error; brackets enclose the name of the error code corresponding to its numeric value. (See Appendix A.)

Old File System Calls

The old file system calls are included for completeness. It is suggested that users convert to the new calls. In the old file call, if the user did not trap the error with an ALTRTN in the subroutine call, the following would occur:

1. Error message printed at terminal

2. Control returned to PRIMOS

The error message would have appended to it, in parentheses, the name of the file system subroutine in which the error occurred as:

BAD DAM FILE (PRWFIL)

Trapping the error with an ALTRTN allowed the user to obtain the error code with the GETERR subroutine and print it at the terminal with the PRERR subroutine. See PRIMOS FILE SYSTEM USER GUIDE, REV. A, MAN26Ø4 for details. The two-letter error codes generated by these errors are given in brackets following the old file call error messages.

Others

Error messages may be printed by other subroutines or by the operating system. Error messages specific to execution of segmented programs are labelled 64V mode. Some error messages indicate system problems beyond the scope of the applications programmer; they are so indicated in the explanation of the error message.

INSTALLATION IN THE COMMAND UFD (CMDNCØ)

Run-time programs in the Command UFD (CMDNCØ) can be invoked by keying in the program name alone. This feature of PRIMOS is useful if a number of users invoke this program. Only one copy of the program need reside on the disk in UFD=CMDNCØ.

Even more space is saved during execution by multiple users if the program uses shared code (64V mode only). (See Section 12.)

Program Memory Images Saved by the Linking Loader

Installation in the command UFD is extremely simple. The runtime version of the program is copied into UFD=CMDNCØ using PRIMOS' FUTIL file handling utility.

Example:

Assume you have written a utility program called FARLEY. This utility acts as a "tickler" for dates. Using FARLEY, each user builds a file with important dates. The FARLEY utility program, upon request, prints

out upcoming events or occasions of interest to the user.

Note

This utility does not necessarily actually exist; it is used as a plausible example.

First, compile the program

```
OK, FTN FARLEY -64R                 Compile in 64R mode
GO
0000 ERRORS [<.MAIN.>FTN-REV14.0]   Compiler message

OK, LOAD                            invoke the Linking Loader
GO
$LO B←FARLEY                        load the object file; the default
                                      name is used
$                                   load other required modules
 .
 .
 .
$LI                                 load the FORTRAN library
LC                                  load is complete
$SA *FARLEY                         save the memory image
$QU                                 return to PRIMOS

OK, FUTIL                           invoke the file utility
GO
>TO CMDNC0 ORDER                    defines the TO UFD as CMDNC0;
                                    password is ORDER
>COPY *FARLEY FARLEY                copies the runtime program
                                    *FARLEY into UFD=CMDNC0
                                    under the name of FARLEY
>QUIT                               return to PRIMOS Command level

OK,
```

It was not necessary to define a FROM UFD; the default (home) was used.

Any user can now invoke this program:

```
OK, FARLEY                          invoke program
GO                                  execution begins

HOW FAR:                            asks for future time period

etc.
```

Segmented Runfiles Saved by SEG's Loader

A segmented program cannot be run directly from UFD=CMDNC0 because PRIMOS' command processor cannot directly handle the SEG runfiles. The

segmented program may be invoked by means of a non-segmented interlude program in CMDNC0.

The procedure for creating an interlude is:

a. Create the desired SEG runfile.

b. Attach to UFD=SEG.

c. Run the command file CMDSEG; it will ask for a runfile name - this name is the new SEG runfile name used in step d. This command file will create the interlude program under the name *TEST.

d. Make a copy of the SEG runfile in UFD=SEG using FUTIL's TRECPY command. The name of the new SEG runfile should be the name by which it will be invoked.

e. A copy of *TEST should be placed in UFD=CMDN0 using FUTIL's COPY command. The file name should be that by which the program will be invoked.

Example:

a. Extensions to the FARLEY utility described above make it desirable to compile and load it as a segmented program.

```
OK, FTN FARLEY -64V          compile in 64V mode
GO
0000 ERRORS [<.MAIN.>FTN-REV14.0]

OK, SEG                      invoke SEG utility
GO
# VLOAD #FARLEY              establish runfile name
$ LO B←FARLEY                load object file
$ .
  .
  .
$ LI                         load 64V mode FORTRAN library
$ SA                         Save the file
$ QU                         return to PRIMOS
OK,
```

b. Attach to UFD=SEG

```
OK, A SEG
OK,
```

c. The command file CMDSEG creates the interlude program.

```
OK, CO CMDSEG

R  *CMDMA
GO
RUN FILE NAME:  FARLEY
FTN S$$SEG 1/5707
GO
0000 ERRORS (FTN .......
FILMEM
LOAD
MO D64R
CO 171400
LO B←S$$S 171400
SETB * 5
AU 10
LO CMDLIB
AU 1
LI
QU
SAVE *TEST 171400 177777 171400 0 0 0 4000
DELETE S$$SEG
DELETE B←S$$S
* FUTIL.YOUR.SEGMENTED.PROGRAM..INTO.UFD.'SEG'
* WITH.THE.RUN.FILE.NAME.GIVEN.TO.CMDSEG
* FUTIL.*TEST.INTO.CMDNC0.WITH.THE.RUN.FILE.NAME.GIVEN.TO.CMDSEG
* YOU.NOW.HAVE.A.SEGMENTED.PROGRAM.WHICH.WILL.RUN.FROM.CMDNC0!

CO'TTY
OK,
```

d. UFD=SEG contains the SEG runfiles which are actually executed by the interlude programs. The SEG runfile is copied here from the UFD in which it was SAVEd.

```
OK, FUTIL                       invoke FUTIL
GO
>FROM MYUFD                     FROM UFD is user's old home UFD
>TRECPY #FARLEY FARLEY          make a copy under the invocation
>                               name
```

There is no TO UFD defined as the default (home) is being used.

e. The interlude program *TEST is copied into the Command UFD under the name by which it will be invoked.

```
>FROM *                      new FROM UFD - the current home
>TO CMDNCØ ORDER             TO UFD=CMDNCØ; password here
                               is assumed to be ORDER
>COPY *TEST FARLEY           copy the interlude
>QUIT                        return to PRIMOS command level

OK,
```

When FARLEY is entered at the user terminal, the FARLEY interlude program in CMDNCØ is executed. This program attaches to the SEG UFD, restores the segmented runfile FARLEY, re-attaches to the user's home UFD and begins execution of the SEG runfile.

If the SEG runfile requires only one segment of loaded information (procedure, link frames, and initialized common) in user space (segment '4ØØØ and above) it is possible to include the interlude in the SEG runfile. This is discussed in Section 12.

SECTION 8

DEBUGGING

The following information will be discussed in the final version of this document. The TRACE statement format is given in Section 16 of this version.

CODING STRATEGY

- Modular programs
- Use of comments
- Indentation and spacing
- Inserting TRACE statements

COMPILER USAGE

- Syntax checking
- Compiler global TRACE option

PROGRAM EXECUTION

- Use of TRACE output
- Run-time errors

THE PM COMMAND

- R-identity use
- V-identity use
    - Using the SEG loadmaps

PROGRAM VALIDATION

- Test cases
- Intermediate value checking

PART III

ADVANCED PROGRAMMING TECHNIQUES

SECTION 9

OPERATING SYSTEM FEATURES

The following information will be discussed in the final version of this document. For detailed information, refer to PRIMOS INTERACTIVE USER GUIDE, REVISION A, MAN2602, PTU 31 and PTU 42.

COMMAND FILES

- Construction of command files
- Comment lines in command file
- The COMINPUT command
- The COMOUTPUT command

PHANTOM USERS

- Command files for phantom users
- the PHANTOM command
- Monitoring job status
- Halting a phantom user

SEQUENTIAL JOB PROCESSOR

- Command files for sequential job processing
- The CX command
- Monitoring job status
- Removing job from queue

SORT

- Input file requirements
- The SORT command

MAGNETIC TAPE

- The MAGRST command
- The MAGSAV command
- The MAGNET command

SECTION 10

FILE SYSTEM FEATURES

The following information will be discussed in the final version of this document. For detailed information, refer to: REFERENCE GUIDE, FILE MANAGEMENT SYSTEM (FMS) PDR 3110 and PRIMOS INTERACTIVE USER GUIDE, REVISION A, MAN2602, PTU 31 and PTU 42.

FILE NAME CONVENTIONS

Long names
Tree names

FILE ACCESS METHODS

Sequential-access method (SAM)
Direct-access method (DAM)

FILE PROTECTION

Owner and non-owner access - the PROTEC command

EXAMINING FILES

File length - the SIZE command
Comparing files:
- The FILVER command
- The PUSS command
- The SAVER command

FILE UTILITY (FUTIL)

Invoking FUTIL
Use of FUTIL
List of FUTIL commands

SECTION 11

EXTENDED FEATURES OF SEG

THE SEG LOADMAP

Invoking SEG's Loadmap

MAP filename-1 [filename-2] map-option Note 1.

or

MAP * [filename-2] map-option Note 2.

Prints a specified load map of the currently established runfile or any other SEG runfile to the user's terminal or to a file.

filename-1 is the filename (or treename) of a SEG runfile for which a map is to be generated.

filename-2 is the filename (or treename) of the file into which the map is to be written.

Map-Option	Map Sections	Map
Ø (or omitted)	all	Full map
1	I, II	Extent map only
2	I, II, III	Extent map and base areas
3	VII (part)	Undefined symbols only
4	all	Full map (identical to Ø)
5	not applicable	System programmer's map
6	VII (part)	Undefined symbols, alphabetical order
7	all	Full map, sorted alphabetically

Note 1

This format should usually be used to get a loadmap of a runfile other than the established runfile.

If filename-2 is not supplied, the map is printed at the user's terminal.

Examples:

MA #TEST 3	print unsatisfied references in SEG runfile #TEST at the terminal
MA #TEST ATLAS 7	write a full map of SEG runfile #TEST into map file ATLAS
MA 1	print an extent map of the established runfile at the terminal.

Note 2

If the runfile name has been established, this format will write the map into the specified mapfile filename-2. If filename-2 is omitted, the loadmap will be printed at the terminal.

Example:

MA * ATLAS	write a full map of the currently established SEG file into mapfile ATLAS.

Contents of SEG Loadmap

The loadmap is particularly useful for:

- location where program halted (LB address after a crash)
- modules not loaded (MA 3 or MA 6)
- reason for stack overflow (SB address after a crash)

(Use of the PM command to obtain LB, SB, etc., addresses and their use is discussed in Section 8 - Debugging.)

The full SEG load map consists of seven sections (see Figure 11-1 for an example of MA 7), not all of which may be present in any load. In particular, Section III may not be present in small SEG loads.

```
I    *START  004002  000002  *STACK  177777  000000  *SYM    171470

II   SEG. #    TYPE      LOW       HIGH      TOP
     004001    PROC##    001000    001441    001441
     004002    DATA      000000    000266    000266

III  *BASE   000000  000176  000242  000275  000275

IV   ROUTINE        ECB          PROCEDURE     ST. SIZE  LINK FR.
     GETNUM     4002  000043    4001  001036    000016    177437
     TlIB       4002  000227    4001  001426    000012    177627
     TlOB       4002  000247    4001  001434    000012    177627
     TlOU       4002  000205    4001  001372    000016    177605
     TBUFIN     4002  000117    4001  001174    000020    177517
     TIDEC      4002  000073    4001  001075    000032    177473
     TODEC      4002  000141    4001  001277    000020    177541
     TONL       4002  000163    4001  001364    000012    177563

V    DIRECT ENTRY LINKS
       ClIN      4001  001412    EXIT      4001  001416    TNOUA     4001  001422

VI   COMMON BLOCKS
       DRW       4002  000036

VII  OTHER SYMBOLS
     **IIIABS    4002  000032
```

Figure 11-1. Full SEG Loadmap (MAP 7).

Section I - Extent: Section I is always present and contains the currently assigned Start address (*START), Stack address (*STACK), and the current bottom location of the symbol table (*SYM).

*START at the beginning of a load, the start address is initialized to 000000 000000. (The first word is the the segment number; the second word is the the address in the segment - both are octal). SEG fills in *START for the first segmented procedure encountered (usually the main program).

*STACK Current address of the start of the stack; initialized to 177777 000000 at the start of a load. This value is not changed until a Loader SAVE or EXECUTE command is invoked. (The first word is the segment; the second is the address -- both octal.) The default stack is in the first procedure segment with 6000 (octal) free locations at the top of memory.

*SYM address of the bottom of the symbol table; one word only as it is a 64R mode address. Indicates to the user how much space is left for the symbol table. To determine the location of the top of the symbol table, generate a map prior to loading; at this time the top and bottom of the symbol table will be identical and *SYM will also be the location of the top.

Section II - Segment Assignment Descriptions: Each segment is labeled as procedure (PROC) or data (DATA); the segment chosen for the stack is identified by ## following the segment type.

LOW lowest loaded location in the segment. (Not necessarily the lowest assigned location.) Initialized to '177777 (-1) at segment creation; if the segment is used only for uninitialized COMMON areas, LOW is not changed.

HIGH highest loaded location in the segment. (Not necessarily the highest assigned location.) Initialized to '000000 at segment creation; if the segment is used only for uninitialized COMMON areas, HIGH is not changed.

TOP highest assigned location in the segment. TOP should not be lower than HIGH. If it is, the user may have specified incorrect load addresses. When not using default values, the user is responsible for loading into correct areas. TOP is initialized to '177777 (-1) at segment creation. When space is reserved for large COMMON blocks, the Loader will only set TOP to a maximum of '177776 even though the entire segment to '177777 is reserved.

The reason for this is: a LOW, HIGH, and TOP of

177777 000000 177777 labels an empty segment.

Section III - Lists Base Areas (All values are octal):

*BASE VVVVVV WWWWWW XXXXXX YYYYYY ZZZZZZ

VVVVVV	segment number
WWWWWW	lowest location for base area
XXXXXX	next available location if starting up from lowest location
YYYYYY	next available location if starting down from highest location
ZZZZZZ	highest location for base area

The lowest default location for the sector zero base area is '100.

There may be a sector zero base area in each procedure segment; there must be none in data segments. Base areas other than sector zero ones are generated by PMA modules.

Section IV - Describes all Symbols Related to Procedures: When a routine (main program or subroutine) is compiled in 64V mode, the compiled result is called a procedure. A procedure is composed of a procedure frame (the executable code), an ECB (the entry control block), a link frame (local storage, constants, transfer vectors) and a stack frame (dynamically allocated storage which is created when the routine is called and released upon return from the routine). This section of the map describes these items. For FORTRAN procedures, the ECB is part of the link frame. The procedure frame will be located in a segment reserved for procedure frames. Link frames and COMMON Blocks will be located in segments reserved for data.

The first pair of numbers in this section of the map is the segment and word address for the ECB, the second pair is the segment and word address for the procedure start.

Column 6 (ST. SIZE) is the size of the stack frame (working area) created whenever the routine is called. Its segment (and location therein) are assigned at execution time.

Column 7 (LINK FR.) gives the link 'offset' - this is '400 locations lower than the actual position for compatibility with the information printed by the PRIMOS PM command. The segment number is usually that of the ECB.

The stack size and link frame information are useful in debugging and are discussed in that section.

Procedures with no names, specifically a FORTRAN main program, are identified by #### in the name field.

Section V - Direct Entry Links: At Rev. 14, PRIMOS IV supports direct entry calls to the supervisor for certain routines. These are created as fault pointers in the SEG runfile. Where references are satisfied by these fault pointers, they will appear in the DIRECT ENTRY LINKS section of the map. The FORTRAN programmer will not be concerned with this map section.

Section VI - COMMON Blocks: Lists a COMMON Block, its segment number, and word address in the segment. To save space, three COMMON blocks are given on each line.

Section VII - Other Symbols (including Undefined Symbols): Lists the symbol, its segment, and word address in that segment. As in Section VI, the format is three symbols per line. Unsatisfied references are preceded by **; for example

```
**COMO$$  0006  064152
```

The numbers for unsatisfied references (segment and word address) locate the last request for the routine (here COMO$$) processed by the Loader. This allows the routines calling missing routines to be identified. (See Section 8 - Debugging.)

Ordering Within Map Section

Section II - Segment table is sorted in order of creation.

Section IV - Procedures are normally sorted by address of executable code (not ECB address) (MA, MA 4) or alphabetically by symbol name (MA 7).

Section V, VI, VII - Sorted by Address (MA, MA 4), or, alphabetically by symbol name (MA 7).

MA 3 lists unsatisfied references (part of Section VI).

MA 6 lists unsatisfied references (part of Section VI) alphabetically by symbol name.

EXTENDED FUNCTIONALITY OF THE LOADER SUB-PROCESSOR

The Loader sub-processor is accessed from the SEG command level by the VLOAD or the VLOAD * command.

VLOAD * [filename] (or LOAD * [filename])

This command accesses the SEG Loader, preserving the contents of the specified runfile. VLOAD * should be used for:

- adding to an existing runfile (LOAD, LIBRARY)
- replacing a module in an existing runfile (RL)
- loading program modules to shared procedure templates (Section 12)

filename is the filename or treename of a SEG runfile; if omitted the established runfile will be used.

This section describes features in SEG's Loader useful for: more advanced programming, solution of specific problems, user convenience, and debugging.

The commands are:

ATTACH
INITIALIZE
RL (Reload)
STACK
MAP

COMMON REL
LOAD, LIBRARY and RL with arguments and the D/ prefix modifier
SYMBOL
R/SYMBOL

The last four commands give the user additional limited control over program and COMMON placement within SEG's default segments. They are described following a discussion of the Loader's relative segment assignment feature.

The SYMBOL and R/SYMBOL commands override some of SEG's testing: correct segment, enough room in segment, etc. The user must perform these functions if SYMBOL or R/SYMBOL are used.

ATTACH [ufd-name] [password] [ldisk] [key]

Attaches to another UFD.

ufd-name is the UFD; default is home UFD

password is the password of the UFD to be attached to, if password-protected.

ldisk is the logical disk on which MFD is to be checked for UFD specified.

Ø or omitted	search logical disk Ø
1ØØØØØ	search all logical disks
177777	search logical disk on which current UFD is located

key key giving attach/set information

Ø	attach to UFD in MFD; do not set home
1	attach to UFD in MFD; set home to new current UFD
2	attach to sub-UFD in current UFD; do not set home
3	attach to sub-UFD in current UFD; set home to new current UFD

Since SEG supports treenames throughout, this command is a convenience rather than a necessity. However, since the treenames involved may be long, the user might find it easier and less subject to typing error to attach to the UFD containing the object files to be loaded rather than using the treenames of these files.

Note

After the completion of the ATTACH command (unlike library), the user remains attached to the new UFD and must specifically re-attach to the original UFD if desired.

INITIALIZE [filename]

Initializes SEG's Loader and restarts it. The SEG runfile will also be initialized if it is an old file.

filename is the filename (or treename) of a SEG runfile; if omitted the current runfile name is used.

This command may be used to abort a bad load or to begin a new load after a SAVE command.

IN	Initialize currently established runfile (bad load)
IN filename	Open new SEG runfile filename (treename is allowed)

RL filename

'Replaces' a routine or routines in a SEG runfile, making it possible to replace a defective subroutine without having to completely rebuild the runfile.

filename is the filename (or treename) of the file to be reloaded.

The new module logically and functionally replaces the old module of the same name by patching the entry point. The new module need not be the same length as the old since it is not physically reloaded on top of the old module.

Example:

RL B←MODULE places MODULE in SEG's default segments and logically replaces the old B←MODULE subroutine with the new one.

Redefinition of COMMON blocks is not allowed; however, new COMMON blocks may be added.

Note

The new module replaces the old module functionally but not physically. Thus, the old module still occupies space in the runfile. Overuse of the RL command may significantly increase runfile size and restore and execution times.

CAUTION

To access an existing runfile for reloading, use SEG's VL * (or LO *) Load command. It is advisable to use a copy of the runfile for reloading as a mistake may destroy the runfile's integrity. The NEW subcommand of MODIFY (SAVE) may be used for this.

STACK size

Changes the amount of space required for the stack.

size is the minimum required stack size in words (octal).

Example: ST 100000

requires at least '100000 free locations in the segment used for the stack. To force use of a whole segment, set size to '177774.

Note

This command can only change stack size; changes of stack location must be done with the SK command in the MODIFY (SAVE) sub-processor.

MAP [filename] [map-option]

Prints a loadmap of the currently established runfile to the user's terminal or to a file.

filename is the filename (or treename) of the map file to be opened by the loader; if filename is omitted, the map will be printed at the user's terminal.

When a map file is specified, it is opened on PRIMOS Unit 13 and remains open until the Load session is completed. Any additional MAP commands specifying output to a file will use the one already opened; exiting from the Loader (via EXECUTE, QUIT, or RETURN) closes the map file. If the user has opened a file on PRIMOS Unit 13 prior to invoking SEG's Loader, then this file will be used for the map. In this case, leaving the Loader does not close the file.

map-option specifies the type of map.

The loadmaps and options are identical to those generated by SEG's MAP command and are discussed in detail there. Omitting the map-option gives a full map; inclusion generates partial and/or alphabetically sorted maps.

The Loader's Relative Assignment Feature

User-controlled placement of modules with a load can be desirable for reasons including:

- more efficient runfile
- aid in debugging
- isolation of potential trouble spots

Two mechanisms are provided in the Loader for this purpose: relative

segment assignment (discussed below) and absolute segment assignment (discussed in Section 12).

Relative segment assignment assigns reference numbers to SEG's default segments; these reference number remain associated with their assigned segments during a Load session. Since the Loader assigns and keeps track of those segment numbers, the user retains the benefits of the Loader's internal checking functions (except as specifically noted). Assignments are made by the COMMON REL command or in conjunction with the Loader's family of Load commands (LOAD, LIBRARY, RL, etc.). Reference numbers should be small positive values.

For example:

```
COMMON REL 3

    or

LOAD B←MAIN Ø 1 2
```

(These commands are described in detail later in this section.)

The numbers 1, 2, and 3 are relative segment reference numbers. The Ø where segment reference number is expected, tells the Loader to use the default segments without reference numbers. For example, the sequence of load commands:

```
LO B←MAIN
LO B←SUBR Ø Ø 1
LI
```

can be used to separate SUBR's link frame from the link frames of the rest of the program. This might be done if it were thought that SUBR had a local array with incorrectly specified dimensions. (See the LOAD command below for a fuller explanation).

COMMON REL segno

Allows the user to establish a reference number for segment(s) into which COMMON will be loaded.

segno	is the segment number into which COMMON will be loaded. segno is always a small octal number.

Example:

```
CO REL 1
```

If data segment was assigned a relative value of 1 (see below) then COMMON will be loaded into a segment with this relative segment assignment number. If no such segment has been assigned, then this

command will declare one of SEG's default segments to be data segment (relative) 1 and use it for loading COMMON.

When using SEG's default segment assignments, the COMMON RELATIVE command will cause SEG to load the COMMON blocks into a different segment than that used for the link frames. This often decreases the size of the runfile which has to be restored. The user may also desire to reserve space for certain COMMON blocks in a selected segment with specific link frames. (See SYMBOL, R/SYMBOL.)

The Load Family

LOAD filename [addr psegno lsegno]
LIBRARY [filename] [addr psegno lsegno]
RL filename [addr psegno lsegno]

The Loader's family of Load commands (of which LOAD, LIBRARY, and RL have been discussed) has optional numeric arguments for load placement control in conjunction with command modifiers (for example D/ as in D/LO filename). Use of the arguments here will be as applicable to relative segment assignment numbers. Reasons for their use was considered above. The optional filename for the LI command has been discussed previously under LIBRARY. Discussion here is confined to the arguments.

addr specifies the starting point for procedure in the segment with relative segment assignment number specified by psegno. If specified to be Ø, the current PBRK for that segment will be used (TOP+1). Users will usually specify Ø for this parameter.

psegno a relative segment assignment number to be used in loading procedure (the code).

lsegno a relative segment assignment number to be used in loading link frames. COMMON will not be loaded with the link frames unless a CO REL command specifying this same relative segment reference number has been given prior to loading this module.

If psegno and/or lsegno are specified as Ø, the ordinary SEG default segments without relative segment assignment numbers are used. IN ALL CASES, the Loader creates the original (and additional) segments with appropriate relative segment reference numbers as needed.

The reference numbers are incremented by the Loader as necessary; thus, it is possible that some COMMON and link frame information will appear in the same segments if suitable (possibly not the same) relative segment assignment numbers are chosen.

Example:

For a specific program, it is known (from the loadmap) that the link frames occupy 2-1/2 segments and COMMON will occupy about 1/2 segment. The following commands will permit the last half segment of link frames to occupy the top of the COMMON segment:

```
CO REL  3
LO B←MAIN Ø 1 1
LO B←SUB1 Ø 1 1
   .
   .
   .
LO B←SUBLAST Ø 1 1
LI Ø 1 1
```

The use of 1 for both psegno and lsegno is non-conflicting, as the Loader keeps track of which are procedure and which are link segments.

D/

The D/ modifier tells the Loader to use the same numeric parameters as were used for the preceding Load family command. The example above is equivalent to:

```
CO REL 3
LO B←MAIN Ø 1 1
D/LO B←SUB1
   .
   .
   .
D/LO B←SUBLAST
D/LI
```

The commands:	long form
LO B←MAIN	LO B←MAIN
LO B←SUB1 Ø 1 1	LO B←SUB1 Ø 1 1
D/LO B←SUB2	LO B←SUB2 Ø 1 1
LI	LI

cause MAIN and the FORTRAN libraries to be loaded in the same pair (procedure and link) of segments. SUB1 and SUB2 will be loaded in the same pair of segments but these will be a different pair from those used for MAIN and the FORTRAN libraries.

The D/ modifier is especially useful for large loads and in command files. Use of D/ decreases input typing and time and minimizes errors; editing command files is made simpler (fewer changes) with less chance of error.

SYMBOL [sname] segno addr

sname is the symbol name

segno is the absolute segment (octal) in which the symbol is to be located.

addr is the address (octal) in the specified segment for the symbol.

SYMBOL may be used to define a symbol at a specific location in memory; it is not necessary that the segment ever be defined by the loader.

SYMBOL does not actually assign a segment in SEG's segment table for the symbol sname, but only an entry in the symbol table. Hence, the command is useful for defining COMMON blocks which will not be restored to memory (uninitialized) when a program is invoked; this will decrease restore time prior to execution. SYMBOL may only be used to define a symbol before it is referenced.

Note

SYMBOL cannot be used to define a COMMON block which will be initialized by a DATA statement or BLOCK DATA subprogram. Symbol names defined by this command cannot be used to satisfy unsatisfied references in a partial load.

Examples:

```
SY CYMBAL 4001  12000
```

locates symbol CYMBAL at segment '4001, location '12000.

```
SY  4015   1000
```

defines blank COMMON as beginning in segment '4015 at location '1000. Here the user has located blank COMMON above the other program procedure and data segments so that overflow of blank COMMON (indexes out of range) will not overwrite other code. The user must determine which segments and locations are to be used by examining SEG's loadmaps. (See R/SYMBOL, COMMON.)

Example of Use: A program BENCH has 3 large (over 33K) COMMON blocks. It is desired to reduce time to restore the runfile to memory and also reduce the number of segments used. It has been determined that segment '4000 (SEG's segment) is available above location '60000. (This is actually the case at Rev. 14. This location may change, mostly upward, from Revision to Revision. The exact location can be obtained by RESTORing SEG and checking the top loaded location using the PM command. See PRIMOS INTERACTION USERS GUIDE, MAN2602 for

details.) A previous load of BENCH determined that the procedure loaded in segment ´4001 ended well below ´60000. Finally, the link frames in segment ´4002 would end well below ´60000 if some of them did not get loaded after the large COMMON blocks were declared.

The COMMON blocks are AA, BB, and AABB; none are initialized. They will fit in the ´120000 locations above ´60000. The following load sequence will reduce the number of segments used from 5 (including SEG´s) to 3.

```
SY  AA 4000  60000
SY  BB 4001  60000
SY  AABB 4002  60000
LO  B←BENCH
```

The user is responsible for placing the COMMON blocks and afterwards must examine the loadmap to be sure that it conforms to expectations.

R/SYMBOL sname [segtype] segno size

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 relative segment reference number. If 0 is specified, the first avaible segment of the current type is used.

size is the number of locations to be reserved for the symbol. If omitted, it is assumed to be 0.

This command places a symbol and reserves 0 or more locations in memory for it. This is especially useful in controlling the placement and size of COMMON blocks during a load. If the segment specified does not exist, or does not contain enough room, an appropriate new segment will be created to locate the symbol.

CAUTION

> The user must check that the number of locations reserved for the symbol is sufficient if it is to be used as a COMMON block (or for any other purpose).

These commands may not be used to satisfy unsatisfied references already existing in the load.

Examples:

(TOP+1) is the next available location in a given segment - see MAP.)

R/SY COUSIN Ø 1ØØØ	places symbol COUSIN at the current TOP+1 in a data segment with no reference number, reserving 1ØØØ (octal) locations for it.
R/SY COUSIN PR Ø 1ØØØ	place symbol COUSIN at current TOP+1 in a procedure segment with no reference number, reserving 1ØØØ (octal) locations for it. This is a way of placing a COMMON block in a procedure segment.
R/SY COUSIN DA 1 Ø	place symbol COUSIN at current TOP+1 in a data segment with reference number 1, reserving Ø locations for it.

In the above case, if a segment with reference number 1 did not exist, it would be created and the address of COUSIN would be Ø (a special case of TOP+1).

THE MODIFICATION SUB-PROCESSOR

SEG's modification sub-processor is accessed by the SEG level command MODIFY.

MODIFY [filename] or (SAVE [filename])

filename is the filename (or treename) of the SEG runfile; if omitted, the established runfile name is used.

The command invokes the modification sub-processor. This sub-processor allows the user to create a new runfile or modify and rewrite to the disk an old runfile. Modifications permitted are:

- Change starting ECB address (not of consequence in FORTRAN)
- Change stack size and/or location
- Save a copy of a runfile modified with VPSD to the same or to a new runfile
- Create a new copy of a shared procedure template file for creation of a program using the template.

SK ssize Note 1.

or

SK segno addr Note 2.

Specifies either a new stack size (1) or absolute stack location (2).

Notes

1. ssize is the stack size (octal) in words. Changes the size required for the stack (default is '6000). To reserve an entire segment, set ssize to '177774. This form of the command would be used to handle stack overflow problems (a run-time error).

   Example:

   SK 60000 set up a stack of minimum length 60000 octal words in a segment of SEG's choice. If 0 is specified for ssize, the default value of '6000 is used.

2. segno is the specific (octal) absolute segment number; addr is the starting (octal) address for the stack in the specified

segment.

This form of the command is used when the user has previously reserved, during the load process, 4 locations in a data segment (segno) for the stack header.

Note

addr must be 4 or greater, as locations 0 to 3 in the segment where the stack is located are used for stack hardware. When locating the stack in a specific segment, be sure that locations 0 to 3 have not been allocated; reserve them using the R/SY command.

Example:

SK 4000 122000

locate the start of the stack in segment '4000 at location '122000.

If extremely large stacks are required, extension stack segments may be created. This is discussed in Section 12.

START segno addr

segno is the absolute (octal) segment number

addr is the new ECB address word (octal) in the segment for start of execution.

One possible application of this command is the creation of template programs with multiple entry points (i.e., programs alike except for the start of execution location). If reset to 000000 000000 as part of template creation, SEG's Loader will reset *START to the starting address of the program using the template.

NEW filename

filename is the filename (or treename) of the new SEG runfile which is to be created.

Duplicates all portions of a SEG runfile resident above segment '4000 under the specified new name. The full map and all references to segments below '4000 are preserved. It may be used to create a template for further additions. If there was a previous file named filename, it is overwritten.

CAUTION

If there is a segment ´4000 in the runfile containing loaded information below the symbol MAP in SEG´s load map, it will overwrite and crash SEG.

RETURN

Writes the entire runfile to the disk and then transfers control back to the SEG command level.

SEG LEVEL COMMANDS

RESUME [filename]

or

RESUME [filename]

The runfile will be restored to memory, if necessary, and then executed. At the PRIMOS level, the SEG filename command is preferred to RESUME.

filename is the filename (or treename) of a SEG runfile. If omitted, the currently established runfile is used.

TIME [filename]

Prints, at the user's terminal, the time of creation or last saved modification of the file. Modification means any changes to the load or starting parameters.

filename is the filename (or treename) of the SEG runfile. If omitted, default is to the established runfile.

This command allows the user to know when the runfile was last modified by anyone.

Example:

```
OK, SEG
GO
# TI #TEST
07-21-77  14:13:14
#
```

SECTION 12

SHARED CODE AND OTHER ADVANCED SEGMENTED PROGRAM TECHNIQUES

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 as CALLable with name MAIN. (FORTRAN header SUBROUTINE MAIN)
- Compile in 64V mode.
- Load to the runfile using the SEG Loader's defaults to determine size and placement of COMMON, procedure, etc.
- With this information, initialize and load to the runfile, splitting procedure and data portions of programs. Debug the program.
- Load for shared procedure and return to SEG command level.
- Separate out segments below '4001 into separate R-mode 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.

WARNING

> 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 as a subroutine named MAIN; i.e. the first statement of the program should be:

SUBROUTINE MAIN

This header will work for either shared or unshared loading. In unshared operations SEG will call the main program as a subroutine; in shared operations the interlude program RUNIT will call the main program. A loadmap will show the main routine as MAIN rather than #### as would be the case if the main program had no header. It is not necessary to include a RETURN statement as the CALL EXIT statement at the end of the main program insures an orderly exit to PRIMOS command level.

Note

> Since the main program is labelled as a subroutine named MAIN no other subroutine may have that name. There is no subroutine or function of that name in any of the Prime-supplied libraries; be sure that no user subroutines involved in the load have the name MAIN.

COMPILING

The source program is compiled with the 64V mode option; this produces code to be loaded with SEG. If an array or COMMON block will exceed 64K words in length the program must be compiled with the BIG option. If recursive subprograms (ones that call themselves) are used the program must be compiled with the DYNM option. Both BIG and DYNM may be used in the same compilation; either one forces compilation in the 64V mode. Use of and constraints on over 64K COMMON are treated later in this section. Extension stacks, which may be necessary in certain cases of recursive subprograms or if programs are chained are also discussed later in this section.

LOADING

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

As in the SEG Extended Use (Section 11) 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:

```
OK, SEG                 invoke SEG
#VL #BENCH              establish runfile and access Loader
$LO B←BENCH             load main program
$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. As previously discussed (Section 11) the number of segments used can be decreased by moving the location of COMMON blocks and the Stack. The load would be:

```
OK, SEG                 invoke SEG
#VL #BENCH              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←BENCH             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 '4000
```

```
                        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, the user 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.

WARNING

> Relative assignment numbers (see Section 11) 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 R-mode Interlude program RUNIT. RUNIT allows the segmented program to be invoked as an R-mode 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.

or

SPLIT addr ssegno saddr esegno — Note 3.

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 the 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.

3. Splits segment ´4000 and supports extension stacks.

   segno is the segment in which the stack will begin

   saddr is the beginning stack address in Ssegno

   esegno is the first segment available for stack extensions

Example

SP 10000 4001 177720 4005

Splits segment ´4000 is a procedure portion below ´10000 and a data portion above. During execution the stack will begin at location ´177720 in segment ´4001 and, as needed, an extension stack will be created in segment ´4005, etc.

At least 12 (´15) words must be available in the primary stack segment.

The non-zero value of saddr distinguishes this form of the SPLIT command from its other forms.

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 sname [segtype] segno size

Places a symbol and reserves Ø or more locations in memory for it. If the segment specified does not exist it will be created.

sname is the name of the symbol.

segtype is the type of segment, etither 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; Ø is assumed.

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 4ØØ2 1ØØØ — place symbol KELVIN at the current TOP+1 in data segment '4ØØ2 reserving 1ØØØ (octal) locations for it.

A/SY KELVIN PR 4ØØ1 1ØØØ — place symbol KELVIN at current TOP+1 in procedure segment '4ØØ1 reserving 1ØØØ (octal) locations for it.

The above is a way of placing a COMMON block in a procedure segment.

A/SY KELVIN DA 4ØØ1 1ØØØ — place symbol KELVIN at current TOP+1 in data segment '4ØØ1, reserving 1ØØØ (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 Ø. (a special case of TOP+1).

COMMON ABS segno

Loads COMMON into the specified segment.

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 4Ø15

will cause the loader to load all COMMON into segment ´4Ø15 so long as it will fit, then into segment ´4Ø16, ´4Ø17, 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)

The first three commands have been discussed in Section 6
PL and IL load the 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

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 Ø 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 supplied. 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 procede 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 Ø 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 Ø 4000 4000 load the impure portion of the FORTRAN library into the split segment ´4000.

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

Example:

```
S/LO B_BENCH Ø 2000 4000
D/PL
```

is equivalent to

```
S/LO B←BENCH Ø 2ØØØ 4ØØØ
S/PL Ø 2ØØØ 4ØØØ
```

CAUTION

When using this modifier (S/) some of SEG's checking mechanisms are overidden. 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

xx	is one of the loading commands, LO, LI, RL, PL, or IL.
filename	is the file name (or tree name) 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 linkages 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 Ø 2001 4002
F/S/PL Ø 2001 4002
```

is equivalent to

```
S/LO B=BENCH Ø 2001 4002
F/D/PL
```

XPUNGE dsymbol dbase

Expunges some or all defined symbols from the symbol table. Undefined symbols may not be removed.

dsymbol	Action
Ø	delete only entry points, leaving COMMON areas
1	delete all defined symbols, including COMMON areas

dbase	Action
Ø	retain all base information
1	retain only sector zero information
2	delete all base area information

XP dsymbol is equivalent to XP dsymbol Ø

XP is equivalent to XP Ø Ø

RETURN

Returns the user to the SEG command level. This command does not SAVE the runfile; the user should perform the SEG SAVE sub-command 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 R-mode 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:

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

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

SINGLE [filename] segno

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:

```
#SI 4001
TWO CHARACTER FILE ID:  IX
CREATING IX4001
#
```

The SINGLE command only works for segments loaded with the S/xx command.

Including the R-mode 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 and 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. Uninitialized 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 R-mode 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, SEG
# VL #BENCH
$ SP 4000
$ SY AA 4000 5000
$ SY BB 4002
$ SY AABB 4001
$ S/LO B←BENCH Ø 4000 4000    difference
$ D/LI                        difference
$ SAVE
$ RE
$ SH
TWO CHARACTER FILE ID:  BE
CREATING BE4000
# QU
OK,
```

would load the program as non-shared procedure. The resulting R-mode 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 CMDNCØ or by a PRIMOS RESUME command.

Example:

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

```
OK, SEG                          invoke SEG
# VL #BENCH                      establish runfile and access Loader
$ SP 4000                        split segment ´4000 at location ´4000;
                                 for impure FORTRAN library and data
$ SY AA 4000 5000                locate AA in segment ´4000 at location
                                 ´5000
$ SY BB 4002                     locate BB in segment ´4002
$ SY AABB 4001                   locate AABB in segment ´4001
$ S/LO B↑BENCH Ø 2000 4000       load the procedure portion of the user
                                 program into segment ´2000; load link
                                 frames into ´4000.
$ D/PL                           load the pure FORTRAN library with the
                                 same parameters
$ S/IL Ø 4000 4000               load impure FORTRAN library
$ SAVE                           save the runfile
$ RE                             return to SEG command level
$ SH                             ask SEG to split out segments below ´4001
TWO CHARACTER FILE ID:  BE       SHARE asks for ID
CREATING BE4000
CREATING BE2000
# QU                             return to PRIMOS command level
OK,
```

INCORPORATING FILES INTO SHARED SEGMENTS

Using SEG's SHARE command creates one R-mode runfile for each segment of the SEG runfile below segment '4001. The R-mode 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

filename is the name of the R-mode 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
Ø	none
200	read
600	read and execute
700	read, write, and execute

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

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

WARNING

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

The PRIMOS command OPR 1 must preceed SHARE commands; OPR Ø must follow the last SHARE command.

Example:

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

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

```
CNAME BE4000 BENCH
```

The R-mode image of segment '4000 may also be put into the command UFD

and invoked as a command.

```
OK, FUTIL                    invoke FUTIL
>TO CMDNCO                   define TO UFD
>COPY BE4ØØØ BENCH           copy BE4ØØØ into UFD=CMDNCØ
                             under the name BENCH
>QU                          return to PRIMOS
OK,
```

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

COMMON BLOCKS OVER 64K WORDS LONG

The size of COMMON blocks and the arrays within them are limited only by the operating system. Either 15 or 31 segments of 64K words are available to the user depending upon the version of SEG implemented on the specific system. The size of a 64V mode program includes COMMON blocks and the procedure, linkage and stack frames of the main program, subprograms and required library routines.

Usage

No special syntax is needed to create a COMMON block over 64K; any named COMMON or blank COMMON may be over 64K. The only indication that a COMMON is over 64K is in the concordance. The concordance address field for all items in an over 64K COMMON block contains two 6-digit octal numbers rather than one. The first number corresponds to a segment offset; the second number is the word offset.

Any array in a COMMON block over 64K is treated as an array that spans a segment boundary regardless of size of the array. Code normally generated for array references will not work for these areas. The program (and subprograms) must be compiled with the BIG option. (This also forces compilation in 64V mode).

A COMMON block over 64K must be explicitly declared over 64K in every program that references the COMMON. Otherwise, the compiler will not generate special code for arrays within that COMMON block.

Dummy Argument Arrays

If a dummy argument array may become associated with an array that spans a segment boundary (through a CALL statement or function reference), the compiler must be made aware of this when the subroutine or function is compiled.

Example:

```
COMMON IBUF (1000,200)
CALL SUB (IBUF, 1000, 200)
.
.
.
END
SUBROUTINE SUB (IDUM, N, M)
DIMENSION IDUM (N, M)
.
.
.
END
```

When subroutine SUB is being compiled, the compiler must be notified that dummy argument array IDUM becomes associated with an array that spans a segment boundary (IBUF).

Note that code generated for an array that spans a segment boundary will work whether or not the array actually spans a segment boundary. There are two methods to notify the compiler that a dummy argument array may become associated with an array that spans a segment boundary:

1. Within the subroutine or function, dimension the dummy argument array over 64K words.

   Example:

   ```
   SUBROUTINE S(IARRAY)
   DIMENSION IARRAY (100000)
   ```

Note that this method cannot be used when there are dummy arguments or COMMON dimensions.

2. Compile the subprogram with the BIG option. All dummy argument arrays will be treated as arrays spanning segment boundaries.

   Example: FTN SUB -BIG

BIG also forces compilation in 64V mode.

The above discussion relates only to dummy argument arrays. A dummy argument variable may become associated with an element of an over segment boundary array, and the code normally generated by the compiler will work correctly.

System and Library routines that require arrays as arguments must not be called with arrays that span segment boundaries, unless these routines are recompiled with the BIG option. This includes the matrix

manipulation routines in MATHLB.

Restrictions

There are a number of restrictions on over 64K COMMON blocks and segment boundary spanning arrays. The compiler will issue an error message if any of these restrictions are violated.

Although an array may span segment boundaries, no array element or variable may cross a segment boundary. If the first word of a real number is in one segment, the second word must be in the same segment. For this reason, the compiler must enforce the following restriction:

> Any multiword variable or array of multiword elements must be offset a multiple of its element length from the start of the COMMON block.

Thus, a double-precision variable or array (regardless of its dimension) must be offset Ø or 4 or 8 words, etc. from the start of an over 64K COMMON block. This restriction also applies to items EQUIVALENCEd to elements in an over 64K COMMON block.

Items in COMMON blocks over 64K cannot be initialized by a DATA statement. Any initialization of COMMON blocks over 64K must be done by assignment statements. This restriction applies even if the item is in the first segment of an over 64K COMMON block.

A segment boundary spanning array must not appear unsubscripted in the list of an I/O or ENCODE/DECODE statement. The equivalent functionality can be achieved by using implied DO Loops.

Implementation Notes and Programming Considerations

The code generated for a subscripted array reference normally consists of instructions to load an index register with the subscript followed by an indexed instruction that references the array element. This code sequence cannot be used for a segment boundary spanning array reference because the index registers are only 16 bits wide and indexing never affects the segment number. A segment boundary spanning array subscript is computed using 32 bit integer arithmetic and then added to the array base address. This resultant address is stored in a temporary location and the array element is referenced indirectly through the temporary location. Thus, on every reference to an over segment boundary array, an execution speed and program size penalty is paid relative to a normal array. For efficiency, all arrays under 64K words should be placed in COMMON blocks under 64K.

The compiler requires that any COMMON block over 64K be allocated in contiguous segments. It also requires that starting address to be a multiple of 4, the largest data type size (complex and double precision floating point).

Calculating Array Size in Words

The size of an array is the product of its dimensions multiplied by the number of words per element. The number of words per element is determined by the type of the arrays as follows:

Type	Number of Words Per Item
INTEGER*2	1
LOGICAL	1
INTEGER*4	2
REAL (REAL*4)	2
COMPLEX	4
DOUBLE PRECISION (REAL*8)	4

Example: REAL A(1ØØØ,44)

Number of Words = 1ØØØx44x2=88ØØØ

EXTENSION STACK SEGMENTS

FORTRAN programs using the DYNM parameter for automatic storage of local arrays in the stack may require extension stack segments to prevent overflow. Extension stacks are supported by the SK command in the Modification sub-processor and by the SPLIT command in the SEG Loader. If no extension parameters are supplied SK and SPLIT will operate as previously described.

In specifying extension stack segments the user supplies the first available free segment; SEG then allocates additional extension stack segments sequentially as needed. If an allocated segment is not needed for an extension it is not assigned to the runfile.

SK ssize Ø segno

ssize is the size of the stack to be allocated.

segno is the first segment available for the extension stack.

This form of the command is used to establish a larger stack size in a segment with the extension stack in a segment of the user's choice.

Example:

```
SK 100000 Ø 4005
```

allocates a primary stack segment in the first convenient segment with ´100000 free locations and the extension stack to begin in segment ´4005. If the ssize parameter is Ø, then the default stack size of ´6000 will be used.

SK ssegno addr segno

ssegno is the segment in which the stack begins

addr is the beginning stack address in ssegno

segno is the first segment available for the extension stack

Example:

```
SK 4001 170000 4005
```

Sets the initial stack frame to location ´170000 in segment ´4001. The extension stack frame (if needed) will begin in ´4005 followed by ´4006, etc. (if needed).

Note

At least 12 words (´15) must be available in the primary stack segment.

SECTION 13

INTERFACE TO OTHER SYSTEMS AND LANGUAGES

INTRODUCTION

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

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

MULTIPLE INDEX DATA ACCESS SYSTEM (MIDAS)

Introduction

MIDAS is a system of interactive utilities and high-level subroutines enabling the use of index-sequential and direct-access data files at the application level. Handling of indices, keys, pointers, and the rest of the file infra-structure is performed automatically for the user by MIDAS. Major advantages of MIDAS are:

- Large data files may be constructed
- Efficient search techniques
- Rapid data access
- Compatibility with existing Prime file structures
- Ease of building files
- Primary key and up to 19 secondary keys possible
- Multiple user access to files
- Data entry lockout protection
- Restructuring utility (REMAKE) for file changes and optimization
- File repair utility (REPAIR)
- Partial/full file deletion utility (KIDDEL)

This section introduces the programmer to the major concepts and usage of MIDAS. Sufficient information is presented to allow the programmer to determine if MIDAS would be applicable to specific situations.

Note

> This section does not contain all the information necessary to implement a MIDAS application. The extensive features of MIDAS and the actual implementation and usage are described in detail in REFERENCE GUIDE, MULTIPLE INDEX DATA ACCESS SYSTEM (MIDAS), PDR3061.

Requirements

The MIDAS system requires that UFD=LIB contain the KIDAFM library, the KIDALB library (for non-segmented addressing use) and the VKDALB library (for segmented-addressing use). The library is loaded just prior to loading the FORTRAN library when loading programs. The files PARM.K and OFFCOM, which contain mnemonics for flags and keys used in MIDAS subroutines, must be located in UFD=SYSCOM.

Using MIDAS

MIDAS usage is implemented in four major steps through PRIME-supplied interactive utilities (see Fig. 13-1).

- Creating/modifying the template - the user defines the data sub-file, indices, etc. (CREATK)
- Building the data sub-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 at the application program level, using MIDAS data access subroutines.
- 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 done 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.

Creating and Modifying Template

The interactive program CREATK allows the user to build, examine, and modify a MIDAS template file. This template contains the information the MIDAS programs and subroutines require to build and maintain the data sub-file and its associated index sub-file(s) and directories.

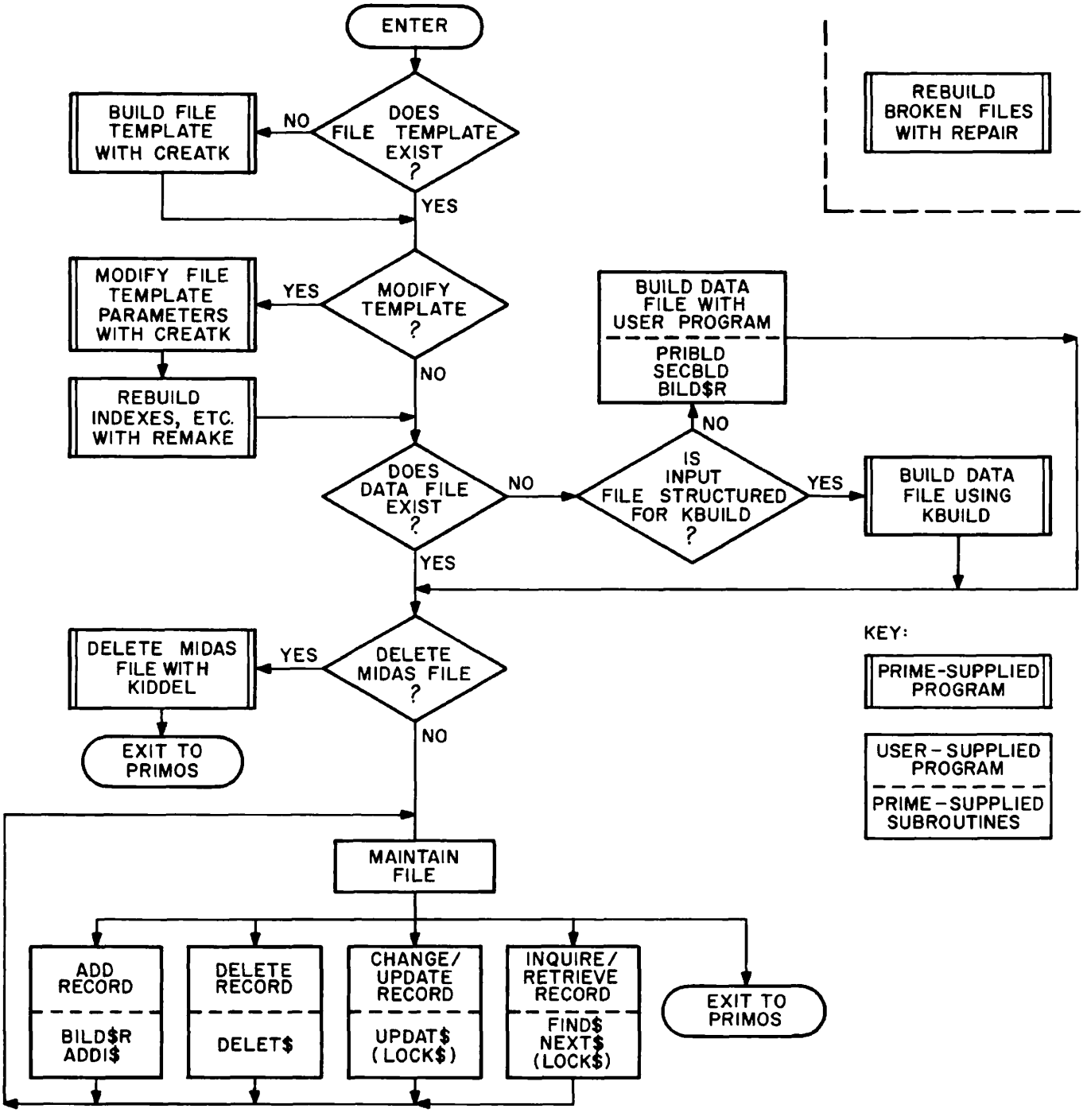


Figure 13-1. User's Functional Overview of the MIDAS File System

When constructing the template, the user specifies filename, direct access support (if supplied), block length, and index requirements (both primary index and secondary indices, if any). For many parameters, the system will supply default values in lieu of the user's specifications if so desired. Secondary indices allow duplicate keys; the primary index key-data record association must be unique. A long index subfile requires the space for two default secondary index subfiles.

If there are no data files to be converted to the MIDAS format, the user may begin file maintenance (addition, updating, deletions) at this point.

The CREATK program can also be used to examine and reset the template parameters for an existing file. The REMAKE program is then invoked to modify the index sub-files and directories. Certain restrictions exist in modifying parameters, especially in converting to long indices.

An example of the template creation dialogue is shown in Figure 13-2.

Building the Data Sub-file

The MIDAS data file may be constructed with the Prime-supplied program KBUILD, or the user may write a file creation program (with the appropriate Prime-supplied subroutines BILD$R, PRIBLD, SECBLD). The use of KBUILD is simpler but it places certain restrictions on the input data files and the resulting output MIDAS data sub-file.

KBUILD Program: KBUILD may be used to generate or add data to MIDAS files; it cannot alter data in existing files. KBUILD expects the input data files to be sequential, fixed-record-length disk files.

Input data files may be text (created by FORTRAN WRITE statements or the text editor) or binary (created by disk I/O subroutines).

During its processing KBUILD prints (to the user's terminal and to a file) non-fatal error messages and milestones. The rate at which milestones are printed is user-specified; milestone information is: records processed, run time, CPU time, disk time, total time, and time used since last the milestone repart. Milestone reports are also generated at the start and end of file processing.

The MIDAS file created by KBUILD has fixed-length records. The user may alter these to variable-length data records by the user of CREATK and REMAKE. The MIDAS file created by KBUILD has completely sorted indexes with no entries in the index overflow areas.

Sample KBUILD dialog:

Suppose the file is sorted on the primary key only, that there is one input file containing 10100 entries called FILE01 in the current UFD, and that the output file is a MIDAS template file called CUSTFIL.KIDA which is on a new partition UFD called NEWPAR. The error file ERRFIL.KIDA will also be written to this UFD.

```
OK, CREATK                                  invoke CREATK
GO
MINIMUM OPTIONS?  YES

FILE NAME?  POLITC                          creating a new file
NEW FILE?  YES
DIRECT ACCESS?  NO

DATA SUBFILE QUESTIONS

KEY TYPE:  A                                ASCII key
KEY SIZE = :  2                             2-word key length
DATA SIZE = :  40                           40 words (80 characters)

SECONDARY INDEX

INDEX NO.?  1

DUPLICATE KEYS PERMITTED?  YES
KEY TYPE:  A
KEY SIZE = :  1
USER DATA SIZE = :  20

INDEX NO.?  2

DUPLICATE KEYS PERMITTED?  YES
KEY TYPE:  A
KEY SIZE = :  2
USER DATA SIZE = :  40

INDEX NO.?  .RETURN.                        RETURN indicates no more
                                             indices
OK,
```

Figure 13-2. Sample of CREATK Dialogue

```
SECONDARIES ONLY? NO
ENTER INPUT FILE NAME: FILE01
ENTER INPUT RECORD LENGTH(WORDS): 63
INPUT FILE TYPE: B
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILE NAME: NEWPAR>CUSTFIL.KIDA
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 51
SECONDARY KEY NUMBER: 1
 ENTER STARTING CHARACTER POSITION: 61
SECONDARY KEY NUMBER: 3
 ENTER STARTING CHARACTER POSITION: 1
IS FILE SORTED? (CR)
IS THE PRIMARY KEY SORTED? (CR)
ENTER INDEX NUMBER OF SECONDARY SORT KEY: (CR)
NUMBER OF RECORDS IN INPUT FILE: 10100
ENTER LOG/ERROR FILE NAME: NEWPAR>ERRFIL.KIDA
ENTER MILESTONE COUNT: (CR=0)
```

User File-Building Program: If the input data file(s) is not in the format expected by KBUILD, the user must write a program to create the MIDAS file. Before building the data file the user must first create a template using CREATK. Three major subroutines (BILD$R, PRIBLD, and SECBLD) are supplied to assist the programmer.

If the input file is unsorted or if the user wishes to add data to an existing file, the subroutine BILD$R should be used. BILD$R adds all entries in the index overflow area and periodically merges and reorganizes the index files. It passes through the file once. It may be used with PRIBLD and SECBLD concurrently.

PRIBLD assumes that the input file data is sorted on the primary key: it is much faster than BILD$R when the input file is about 2000 records or greater.

If the input file is sorted on any secondary keys SECBLD may be used to create those secondary indices files.

Maintaining and Using the File

A number of subroutines are supplied to enable the programmer to make effective use of the MIDAS file. These subroutines are designed to allow more than one user to access the data file simultaneously. All the subroutines require the file PARM.K be inserted in the user program with:

```
$INSERT SYSCOM>PARM.K
```

ADD1$ - adds a data entry to the file and modifies the index sub-files appropriately. Insertion is by primary key only; the file is locked during insertion.

DELET$ - deletes a data entry and modifies the index sub-file(s) accordingly. Deletion may not occur if the data entry is locked.

FIND$ - locates a data entry and reads its contents into a buffer. Look-up is by primary and secondary key(s). If there exist data entries with the same secondary key (synonyms) the oldest data entry (i.e., first one in the file) is retrieved.

NEXT$ - retrieves the data entry with the next higher key. Search may be on primary or secondary keys. This subroutine allows synonyms which are not oldest to be accessed.

LOCK$ - locates a data entry and, if not locked, then locks the data entry. The data entry is unlocked by a successful call to UPDAT$, FIND$, or NEXT$.

UPDAT$ - re-writes a data entry. This subroutine should not be called before a successful call to LOCK$.

An example of a subroutine using NEXT$, LOCK$, and UPDAT$ is shown in Figure 13-3. The $INSERT file KIDINS is one the applications programmer has created to facilitate communication between the main program and various subroutines. In the example, the user would probably check the error return from LOCK$ to see if the record was already locked. If this is the case, it would be appropriate to recycle a few times until the record is unlocked and then proceed with the update.

Performing Housekeeping

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 are searched more slowly than the ordered indices) and frees for use the space occupied by data entries flagged as deleted.

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

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

Updating a Data Record - NEXT$, LOCK$ and UPDAT$

The principle here is that NEXT$ is called until the correct record is found, then this is LOCK$ed and UPDAT$ed.

```
          SUBROUTINE UPDATH(....,ALTRTN,.......
C
          LOGICAL VERIFY, AYENAY
          INTEGER FLAGS, ALTRTN
          DOUBLE PRECISION PAY
$INSERT KIDINS
C
          .
          .
          .
C
C         SET FLAGS TO USE AND RETURN ARRAY, RETURN PRIMARY
C         KEY IN BUFFER AND STOP SEARCHING ONLY WHEN
C         TERMINATED BY USER.
C
             FLAGS=FL$RET+FL$USE+FL$PLW+FL$KEY
             ARRAY(1)=-1                    /*FLAG NEXT$ TO IGNORE ARRAY
C
C         DATA WILL BE READ INTO DBUFFR
C         SEARCH WILL BE DONE ON SKEY1
C         USING INDEX 1
C         FILE# IS 0, AS USUAL - FIRST 0
C         PLNGTH IS 0 - RETURN FULL RECORD - 2ND 0
C         KEYLNT IS 0 - USE FULL KEY - 3RD 0
C
100          CALL NEXT$(1,DBUFFR,SKEY1,ARRAY,FLAGS,$9000,1,0,0,0
             IF(.VERIFY(.......).) GO TO 1000
C
C         AYENAY IS A FUNCTION REQUESTING A YES/NO RESPONSE FOR THE
          MESSAGE INDICATED
C
             IF(.AYENAY('NEXT? ',6).) GO TO 100
             GO TO ALTRTN
C
C         CALL LOCK$ USING SAME PARAMS
C
1000         CALL LOCK$(1,DBUFFR,SKEY1,ARRAY, FLAGS,$9000,1,0,0,0)
             BALANC=BALANC+PAY
C
C         CALL UPDAT$ USING SAME PARAMS TOO
C
             CALL UPDAT$(1,DBUFFR,SKEY1,ARRAY,FLAGS,$9100,1,0,0,0)
             RETURN
```

Figure 13-3. Example of Data Maintenance Program.

REPAIR Program: If a file is damaged by a system or program crash, most of the usable data may be recovered with the REPAIR program. Data entries which are still valid but which cannot be accessed by MIDAS are recovered; data that were overwritten, deleted, or truncated obviously cannot be retrieved. The user first builds a new template identical to the original (as the old one may have been damaged). REPAIR then builds a new MIDAS file using the new template and what exists of the old file. Sufficient space must be available to write the new file.

Messages are printed at the user terminal and in a file. Problems encountered in rebuilding the file (bad data, no index entry, etc.) are also printed with the context in which they were found as an aid to the user.

DATABASE MANAGEMENT SYSTEM (DBMS)

FORTRAN/DBMS Interface

The FORTRAN interface to the DBMS includes two major processors and their respective languages: the FORTRAN Subschema Data Definition Language (DDL) Compiler and the FORTRAN Data Manipulation Language (DML) Preprocessor.

The application programmer's 'view' of a schema is written in the FORTRAN Subschema DDL. The Subschema Compiler translates the DDL into an internal, tabular form called the subschema table which is used by the DML Preprocessor.

Commands for locating, retrieving, deleting, and modifying the contents of a database are written in the FORTRAN DML. These commands are interspersed with FORTRAN statements in the application source program and translated into ~~COBOL~~ FORTRAN declarations and statements by the FORTRAN DML Preprocessor. The output of the Preprocessor is the source input for the ~~COBOL~~ FORTRAN Compiler.

See: PRIME COMPUTER REFERENCE GUIDE FOR DBMS SCHEMA DATA DEFINITION LANGUAGE (DDL), IDR3044.

FORTRAN REFERENCE MANUAL FOR DBMS, IDR3045.

FORMS MANAGEMENT SYSTEM (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 FORTRAN. Applications programs communicate with FORMS through input/output statements native to the host language. Programs that currently run in an interactive mode can easily be converted to use FORMS.

FORMS allows cataloging and maintenance of form definitions available within the computer system. To facilitate use within an applications program, all form definitions reside within a centralized directory in the system. This directory, under control of the system administrator, may be easily changed, allowing the addition, modification, or deletion of form definitions.

FORMS is device independent. If certain basic criteria are met, any mix of terminals attached to the Prime computer may be used with the FORMS system. Terminal configuration is governed by a control file in the centralized FORMS directory. This file is read by FORMS at run-time to determine which device driver to use, depending on this user's terminal type. This means that multiple terminal types may be driven by the same applications program without change. Certain terminal types are supported by FORMS as released by Prime. Should the user have another terminal capable of supporting FORMS, all that need be done is to write a low-level device driver for the terminal and incorporate it into the FORMS run-time library.

See: PRIME FORMS MANAGEMENT SYSTEM (FORMS), IDR3040

OTHER LANGUAGES

COBOL Programs

FORTRAN subroutines may be called by COBOL programs; the responsibility for proper coding is at the COBOL program level.

See: THE COBOL PROGRAMMER'S GUIDE, PDR3056

PMA Programs

FORTRAN subroutines may be called by PMA programs; proper instructions must be placed in the calling program by the PMA programmer. FORTRAN programs may call subroutines written in PMA. The FORTRAN programmer must ascertain the subroutine name, the calling sequence and the data modes of the subroutine arguments.

See: THE PMA PROGRAMMER'S GUIDE, PDR3059

SECTION 14

OPTIMIZATION AND OTHER HELPFUL HINTS

This section is reserved for specific techniques which users have found to be useful. It is intended that this section will serve as feedback for users of this manual.

PART IV

FORTRAN LANGUAGE REFERENCE

SECTION 15

FORTRAN LANGUAGE ELEMENTS

LEGAL CHARACTER SET

The characters allowed in Prime FORTRAN IV are:

a. The 26 letters: A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S, T,U,V,W,X,Y,Z

b. The 10 digits: 0,1,2,3,4,5,6,7,8,9

   Letters and digits together are called alphanumeric characters.

c. These 12 special characters:

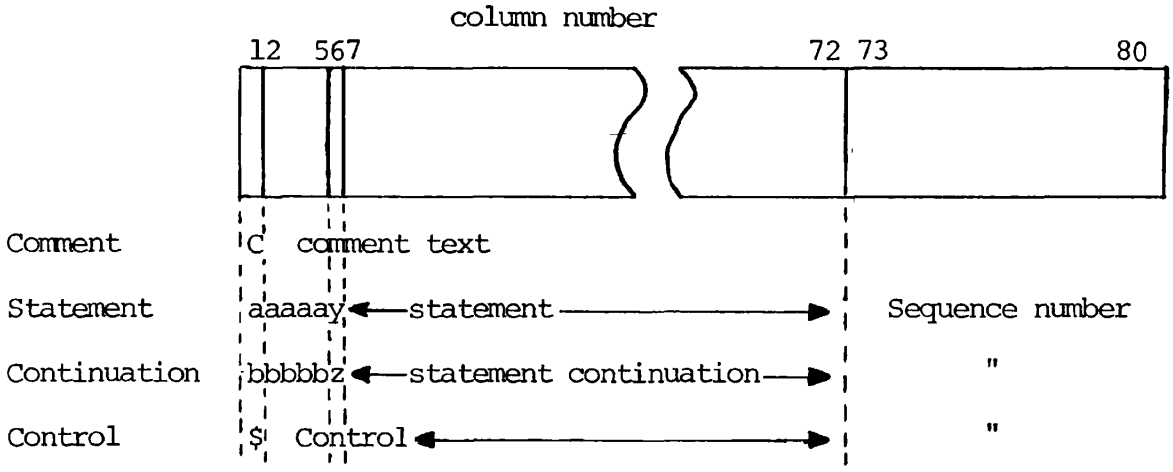
=	equals
'	single quote (apostrophe)
:	colon
+	plus
-	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
$	dollar sign

d. Blanks or spaces (which will be represented as b in text, when necessary).

Blanks in Hollerith constants (character strings) or in formatted input/output statements are treated as character positions. Elsewhere in Prime FORTRAN, blanks have no meaning and should be used as desired to improve program legibility.

LINE FORMAT

Each program line is a string of 1 to 72 characters. Each character position in the line is called a column, numbered from left to right starting with 1. There are three types of lines: Comments, statements (and their continuations), and control statements. (See Figure 15-1.)



aaaaa - Statement label (optional)
bbbbb - Blanks
y - Blank or zero
z - Any character except blank or zero

NOTE: Comments may be extended past column 72 to column 80.

Figure 15-1. Program Line Format

Comments

Comment lines are identified by the letter C in column 1. The remainder of the line may contain anything. A comment line is ignored by the compiler, except that it is printed in the program listing. A comment may be placed on a statement line (except inside a Hollerith constant) using the format:

```
/* comment */
```

Statements

Columns 1-5 are reserved for the numerical statement label, if any. (Blanks and leading zeros are ignored.) Column 6 must be a blank or a zero. Columns 7-72 contain the statement. The statement may begin with leading blanks; this is often done to make the program easier to read, as for indention of nested DO loops or nested IF statements. In the continuation of a statement, columns 1-5 must be blank, column 6 may be any character EXCEPT 0 (zero) or a blank, and the statement continuation is in columns 7-72.

Control

Column 1 must contain the special character $. Other columns are specified by the individual control operation. (See, for example, $INSERT in Section 16.)

Columns 73 to 80 are available for line order sequence numbers or other identification (usage is optional). These columns, like comments, are ignored by the compiler except that they are printed in the program listing.

OPERANDS

Operands are those elements which are manipulated by the program. They are constants, parameters, variables, arrays, and address constants.

Constants

Constants may be any of the following types:

Mode	Memory Words	Range
INTEGER (short)	1	-32768 to +32767
INTEGER (long)	2	-2147483648 to +2147483647 (-2**31 to +2**31-1)
REAL	2	±(10**-38 to 10**38)
DOUBLE PRECISION	4	±(10**-9902 to 10**9825)
COMPLEX	2x2	as for real
LOGICAL	1	0 or 1 (i.e., .FALSE. or .TRUE.)

Integers: May be decimal or octal numbers. In either case, no decimal point appears in the representation. Short integers may have up to 5 decimal digits or 6 octal digits, plus a sign, within the magnitude range.

decimal 12345 or -23579

octal: :13752 or -:156

or

5013752 or -30156

(obsolete notation - supported for compatibility; use is not recommended)

Short integers range in magnitude from 0 to 32767 (decimal); i.e., :0 to :177777 (octal).

Long integers may have up to 10 decimal digits or 11 octal digits plus a sign. 0 to 2147483647

The representation is the same as short integers. Long integers range in magnitude from 0 to 2147483647 (2**31-1); :000000 to :17777777777 (octal) and From -2147483648 (:20000000000) to -1 (:37777777777) = -(2**31)

Integer constants are treated as short integers unless:

a. Their magnitude exceeds 32767 or :177777 (octal).

b. Their representation exceeds 5 decimal digits or 6 octal digits; leading zeros are counted in determining the number of digits in the constant.

Example:

30 short integer

000030 long integer

If the program is compiled with INTL then all integer constants are treated as long integers. (See Sections 4 and 18 for details.)

Long integers may be used in the FORTRAN program anywhere that short integers are used. This included subscripts, ASSIGNed GOTOs, computed GOTOs, FORTRAN I/O unit numbers, DO-loop index values, and character counts.

CAUTION

Some subroutines expect short integers as arguments. In these cases, convert any long integers to short integers via the INTS function (see Section 20 for details).

Real Numbers: may be written as

1357.924

or

0.3579 E 02

The decimal point is mandatory in the first case. In the exponential form the decimal point is optional; the exponent ranges from -38 to +38. The position following the E must contain a blank, a plus sign, or a minus sign. The blank is interpreted as a plus sign.

Only the seven most significant digits are retained.

Double Precision Numbers: are similar to real numbers except that fourteen significant digits are retained, and the exponential (or floating point) representation uses D in place of E, e.g.,

12345.9253 D-11

The exponent (following D) may take on values from -9902 to +9825.
Only two digits can be printed For the exponents.
Complex Numbers: are an ordered pair of two real numbers enclosed in parentheses and separated by a comma:

(REAL1, REAL2) e.g., (1.345, 0.59 E-2)

The rules for real number representation apply to each element of the complex number.

Logical Constants: Logical constants have only two possible values:

0 (zero) corresponding to .FALSE.

1 (one) corresponding to .TRUE.

ASCII: ASCII constants are character strings. They are stored as follows:

Mode	Maximum Number of ASCII Characters Stored
Integer, short	2
Integer, long	4
Real	4
Double Precision	8
Complex	8

When character strings are compared, bit-by-bit checking is only done for those stored in integers; hence storage in modes other than integer (long or short) should be avoided.

Characters are left justified and the remainder of the word(s) are packed with blanks.

ASCII constants are represented in either of two ways:

1. A character count followed by the letter H and the string:

   23HTHISbISbANbASCIIbSTRING

2. The string enclosed in single quotes:

   'THISbISbANbASCIIbSTRING'

A single quote may be represented in a string by using two single quotes ('') (NOT a double quote). This will count as one character.

Example:

```
    WRITE (1,1)
  1 FORMAT ('AB''C')
```

will print AB'C at the terminal.

Parameters

Parameters are named constants and may be of any data mode. They may be used in the program anywhere a constant can be used, except in FORMAT statements; they may also appear in DATA and DIMENSION statements. Parameters are loaded at compile time, and the code generated for them is identical to that generated for constants (see the PARAMETER statement in Section 16).

Variables

Variable names have from 1 to 6 characters. Character 1 must be alphabetic; characters 2-5 (if any) must be alphanumeric.

If no modes are specifically declared, then all variables whose names begin with the letters I,J,K,L,M,N, are integer mode, and variables whose names begin with A-H, or O-Z are real mode. Check Section 16, Specification Statements, for instructions on how to override this implicit convention and also specify double precision, complex and logical modes.

Arrays

Arrays are ordered multidimensional sets of data represented as:

ANAME $(I_1,I_2,\ldots,I_n)$.

The I's are the indexes (subscripts) of the array, and must be positive integers (constants, parameters, or variables). All elements of the array must be of the same mode--integer (short or long), real, double precision, complex, or logical.

CAUTION

If a variable expression is used for an index, its form may be no more complicated than:

(constant)*(integer variable)±(constant).

No more than seven subscripts may be used to index an array.

Address Constants

Address constants consist of a statement label prefixed by a dollar sign ($). They contain the memory address of the first line of code generated by the statement label whose value is that of the address constant.

Example:

100 A=B*C is a statement in the program. Then $100 is the address of the code generated by that statement. The address constant is an integer value. It is usually used in conjunction with the ALTRTN from external subroutines (these are alternate returns generated by encountering errors in executing the subroutines).

OPERATORS

Operators modify an operand or concatenate two operands.

Logical Operators

FORTRAN's logical operators are: .NOT., .AND., .OR. (in this section, P and Q have been specified as logical variables.

.NOT.: .NOT.Q negates the value of Q

Q	.NOT. Q
.TRUE.	.FALSE.
.FALSE.	.TRUE.

.AND.: P .AND. Q is the logical ANDing of the bits of P and Q (set intersection)

Q \ P	.TRUE.	.FALSE.
.TRUE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.FALSE.

.OR.: P .OR. Q is the logical non-exclusive ORing of P and Q. (set union)

Q \ P	.TRUE	.FALSE.
.TRUE.	.TRUE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.

Arithmetic Operators

**	Exponentiation
-	Unary minus
*	Multiplication
/	Division
+	Addition
-	Subtraction
=	Equality or replacement

Relational Operators

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

Operator Priority

FORTRAN evaluates operators within expressions in the following order:

**	Exponentiation
-	Unary Minus
* or /	Multiplication or division
+ or -	Addition or subtraction
.LT.,.LE.,.EQ., .NE.,.GT.,.GE.	Relational operators
.NOT.	Logical negation
.AND.	Logical intersection
.OR.	Logical union

At equal level of operators, priority evaluation proceeds from left to right. Expressions within parentheses are evaluated before operations outside the parentheses are performed.

PROGRAM COMPOSITION

Each program (or subroutine or external function) consists of a number of program lines. Program lines are grouped and ordered by type of statement as shown in Figure 15-2. Comments and TRACE and LIST control statements can be used anywhere in the program. The END statement must be the last statement of a program; nothing may follow END except FUNCTION or SUBROUTINE of another subprogram. The types of statements are discussed in Section 16.

Header statement, if required:

FUNCTION, SUBROUTINE, BLOCK DATA

Storage and Specification Statements:

COMMON, DIMENSION, EQUIVALENCE, SAVE, EXTERNAL, COMPLEX, DOUBLE PRECISION, INTEGER, INTEGER*2, INTEGER*4, LOGICAL, REAL, REAL*4, REAL*8, IMPLICIT, PARAMETER

DATA Statements

STATEMENT
~~Internal~~ Function Definitions

Executable Statements

Arithmetic and logical assignments

Control Statements:	GOTO, ASSIGN, IF, DO, CONTINUE, PAUSE, STOP, RETURN
Input/Output Statements:	READ, WRITE, PRINT, FORMAT, REWIND, BACKSPACE, END FILE
Subroutines:	CALL subrname [(arg-1,...,arg-n)]

END Statement

Figure 15-2. Source Program Composition

SECTION 16

FORTRAN STATEMENTS

IMPLEMENTED STATEMENTS

Legal statements for Prime FORTRAN IV are listed below with their functional category.

Statement	Category
ASSIGN	Control
BACKSPACE	Device Control
BLOCK DATA	Header
CALL	External Procedure
COMMON	Storage
COMPLEX	Specification
CONTINUE	Control
DATA	Data initialization
DECODE	Coding
DIMENSION	Storage
DO	Control
DOUBLE PRECISION	Specification
ENCODE	Coding
END	Control
ENDFILE	Device Control
EQUIVALENCE	Storage
EXTERNAL	External Procedure
FORMAT	Format
FULL LIST	Compilation/Run-Time Control
FUNCTION	Header
GO TO	Control
IF	Control
IMPLICIT	Specification
INTEGER	Specification
INTEGER*2	Specification
INTEGER*4	Specification
LIST	Compilation/Run-Time Control
LOGICAL	Specification
mode FUNCTION	Header
NO LIST	Compilation/Run-Time Control
PARAMETER	Specification
PAUSE	Control
PRINT	Input/Output
PROTECTED FUNCTION	Header
PROTECTED mode FUNCTION	Header
PROTECTED SUBROUTINE	Header
READ	Input/Output
REAL	Specification
REAL*4	Specification

Statement (cont)	Category (Cont)
REAL*8	Specification
RETURN	Control
REWIND	Device Control
SAVE	Storage
STOP	Control
SUBROUTINE	Header
TRACE	Compilation/Run-Time Control
WRITE	Input/Output
$INSERT	Compilation/Run-Time Control

In this reference, section statements are grouped in functional order to clarify and simplify discussion, as follows:

Header Statements:

BLOCK DATA
FUNCTION
SUBROUTINE

Specification Statements:

IMPLICIT

mode:		
	COMPLEX	LOGICAL
	DOUBLE PRECISION	REAL
	INTEGER	REAL*4
	INTEGER*2	REAL*8
	INTEGER*4	

PARAMETER

Storage Statements:

COMMON
DIMENSION
EQUIVALENCE
SAVE

External Statements:

CALL
EXTERNAL

Data Definition Statements:

DATA

Compilation and Run-Time Control Statements:

FULL LIST
LIST
NO LIST
TRACE
$INSERT

Assignment Statements:

Control Statements:

ASSIGN
CONTINUE
DO
END
GOTO
IF
PAUSE
RETURN
STOP

Input/Output Statements:

PRINT
READ
WRITE

Coding Statements:

DECODE
ENCODE

Format Statements:

FORMAT

Device Control Statements:

BACKSPACE
ENDFILE
REWIND

Functions

Subroutines

HEADER STATEMENTS FOR SUBPROGRAMS

BLOCK DATA Statement

BLOCK DATA

The BLOCK DATA statement labels a block data subprogram. This type of subprogram labels COMMON areas and then initializes data values within these areas via DATA statements. Block data subprograms are compiled separately and linked to the main program by the Loader.

FUNCTION Statements

[PROTECTED] [mode] FUNCTION name (argument-1, argument-2, . . argument-n)

The arguments are a non-empty list of the arguments passed by the calling program. There is no syntactical upper limit to the number of arguments. However, long lists will slow execution. The name - is both the name of the function in the calling program and the variable that returns the value calculated by the function. The mode - is an optional specification of one of the data types, selected from the following list:

COMPLEX	LOGICAL
INTEGER	REAL*4 (REAL)
INTEGER*2	REAL*8 (double precision)
INTEGER*4	

If no mode is specified, FORTRAN will assign one implicitly based upon the first letter of the function name (i.e., I-N=Integer, A-H or O-Z=Real).

CAUTION

Labelling a function or subroutine as PROTECTED prevents it from being interrupted before it has completed its calculations and returned to the calling program. This is a system-level command enhancement and is not available at the time-sharing user level. Inclusion of PROTECTED in a program designed to run in the restricted (time sharing) mode will generate a run-time error.

SUBROUTINE Statements

[PROTECTED] SUBROUTINE Name [argument-1,argument-2...argument-n]

The arguments are a list of arguments, some of which are passed by the calling program. Others are dummy arguments whose values are calculated by the subroutine and returned to the calling program. There is no syntactical upper limit to the number of arguments. However, long lists will slow execution.

CAUTION

Under PRIMOS, subroutines are called by address (location) rather than by name. Thus, it is extremely important not to place constants or parameters in the argument list as arguments which will be returned, since this will alter their value. Also, returned arguments may not be expressions.

Example:

```
      I=5                          prints on user terminal
      PRINT 10,I                   5
      CALL SUB1(I,5)
      I=5
      PRINT 10,I                   25
   10 FORMAT (I2)
      .
      .
      .
      SUBROUTINE SUB1 (J,K)
      K=J**2
      RETURN
      END
```

See also the note for PROTECTED FUNCTION above.

SPECIFICATION STATEMENTS

FORTRAN automatically assigns modes to all variables, parameters, arrays, and functions (except intrinsics) that do not appear in mode specification statements. The FORTRAN language default is as follows: if the symbol's first character is I through N (inclusive), the symbol is typed as integer; all others (A-H, O-Z) are typed as real. (The default integers are short integers unless the program is compiled with the long integer default - see Section 4.

IMPLICIT Statements

IMPLICIT mode-1 (list-1), mode-2 (list-2),..., mode-n (list-n)

The IMPLICIT statement allows the programmer to override the language convention for default data typing. Each mode is a data mode such as REAL*4, COMPLEX, etc. Each list lists the letters to be typed as the mode specification. Letters may be separated by a comma or an inclusive group of letters may be indicated with a dash.

Symbols not typed in this statement and not specified in mode specification statements will revert to the FORTRAN language default.

Example:

```
IMPLICIT DOUBLE PRECISION (A,M-Z), LOGICAL (B)
```

First letter of symbol	Type
A, or M through Z	Double Precision
B	Logical
C through H	Real
I through L	Integer

If used, the IMPLICIT statement must be the first statement of a main program, or the second statement of a subprogram. IMPLICIT typing does not affect intrinsic or basic external functions. IMPLICIT affects all symbols not otherwise typed. This includes dummy variables in the first statement of a subroutine or function. The user should take care to make sure that these dummy variable symbols will be of the proper data type.

Mode Specification Statements

```
mode [V1,V2,...,Vn]
```

The mode specification statement allows override of the implicit mode assignments of symbol names which was done either by IMPLICIT or language default.

The word mode is replaced by one of the nine data mode specifications:

```
COMPLEX
DOUBLE PRECISION (same as REAL*8)
INTEGER
INTEGER*2
INTEGER*4
LOGICAL
REAL (same as REAL*4)
REAL*4 (same as REAL)
REAL*8 (same as DOUBLE PRECISION)
```

The V's are a list of variable names, parameter names, array names, function names, or array declarers.

Recognition of synonymous specifications is designed to ease conversion of extant programs to the Prime FORTRAN system. INTEGER will normally default to INTEGER*2 (short integer) unless the program is compiled including the INTL option. In this case, INTEGER will default to INTEGER*4 (long integer). It is recommended in new programs that the programmer explicitly use INTEGER*2 and INTEGER*4 specifications. (See Section 4 for compiler information.)

Global mode definition occurs if a mode specification does not include a symbol list. In this case, all symbols which do not appear in specification statements and whose first appearance follows this global mode statement are declared to be of this globally-specified mode.

CAUTION

The use of global mode and the IMPLICIT statement in the same program unit is prohibited. The global mode is functionally replaced by the IMPLICIT statement. The use of the IMPLICIT statement is strongly recommended as a superior programming technique. The global mode is still supported by the FORTRAN system to allow the use of existing programs utilizing it.

PARAMETER Statement

PARAMETER (V1=C1, V2=C2.,,,. Vn=Cn)

Where the V's are variables (arrays are not allowed) and the C's are constants or constant expressions of the same mode as the corresponding variables. The operands in the constant expressions may be constants or previously defined parameters. Allowed operations include +, -, *, and / on Integer*2, Real*8, and Real*4 operands. Integer *2 XOR, OR, AND, MOD, shift, and truncate function references are also allowed. An error message, ILL. CONSTANT EXPR., is generated if these restrictions are violated. The variable names must be typed explicitly prior to the PARAMETER statement or default-typed implicitly. All other uses of the PARAMETER names must follow the PARAMETER statement. PARAMETER names may be used wherever a constant would be used (including DATA and DIMENSION statements) except in FORMAT statements. Since the parameters are named constants, PARAMETER names may not be used in COMMON or EQUIVALENCE statements.

Enclosing the parameter list in parentheses is required by the proposed FORTRAN 77 standard. At Rev. 14, Prime's FORTRAN will accept a PARAMETER statement with or without the parentheses.

STORAGE STATEMENTS

COMMON Statement

COMMON /X1/A1/.../Xn/An

Where each A is a non-empty list of variable names, array names, or array characters (no dummy arguments) and each X is a COMMON block name or is empty (blank COMMON). The COMMON block names must not be identical with names of subprograms called or FORTRAN library subroutines. Data items are assigned sequentially within a COMMON block in the order of appearance. The loader program assigns all COMMON blocks with the same name to the same area, regardless of the program or subprogram in which they are defined. Blank COMMON data are assigned in such a way that they overlap the loader program, thereby making the memory area occupied by the loader program, available for data storage.

Note

The form // (with no characters except blanks between slashes) may be used to denote blank COMMON.

The number of words that a COMMON block occupies depends on the number of elements, the mode of the elements, and the interrelations between the elements specified by an EQUIVALENCE statement. COMMON blocks that appear with the same block name (or no name) in various programs or subprograms of the same job are not required to have elements within the block agree in name, mode, or order, but the blocks must agree in total words.

As an aid to system-level programming, the compiler defines absolute memory location '00001 as the origin of a COMMON block named 'LIST'.

It is customary to assign an array called LIST into the labeled COMMON area called LIST, such that the first word in this array is location '00001, the sixth word location '00006, etc., as in:

```
COMMON/LIST/LIST(1)
```

Effectively, the subscript of array LIST is the actual memory address. This feature is not required when compiling in 64V mode.

Note

Techniques for handling COMMON areas larger than 64K words (64V mode only) are discussed in Section 12.

DIMENSION Statement

```
DIMENSION V1(I1), V2(I2),... VN(In)
```

Declares the name of the array, the number of subscripts (IJ=J1, J2,... Jn n=1 to 7), and the maximum value for the subscripts. This allocates the maximum storage requirement for the array. In a subroutine, the subscript(s) in a dimension statement may be a variable, provided this value is passed to the subroutine from the calling program.

EQUIVALENCE Statement

```
EQUIVALENCE (k11, k12, k13...), (k21, k22, k23...)
```

Where each k is a variable, subscripted variable or array name. Each element in the list is assigned the same memory storage by the compiler. An EQUIVALENCE statement equates single variables to each other, entire arrays to each other, elements of an array to single variables and vice-versa. If equivalences are established between variables of different modes, the shorter mode is stored in the first words of the longer mode.

SAVE Statement

```
SAVE V1, V2,..., Vn
```

Where the V's are local variables or array names. Arrays cannot be dimensioned in a SAVE statement. Any symbol name appearing in a SAVE statement cannot appear in a COMMON statement or be EQUIVALENCEd to a COMMON element. A labeled COMMON block (not blank COMMON) may appear in the list if it is enclosed in slashes.

Note

In the current revision, inclusion of a COMMON block name has no effect. This feature is included to allow compatibility with the new proposed FORTRAN standard.

Variables listed in the SAVE statement are assigned local storage in the linkage frame (static) rather than the stack frame (dynamic). Thus, the SAVE command has meaning only when the program is compiled including the DYNM command (64V mode only). Symbol names in DATA statements, SAVE statements or EQUIVALENCEd to names in these statements are stored in the linkage frame. Only variables in the linkage frame can be initialized. Variables allocated to the stack frame are not preserved from one subroutine CALL to the next.

If the SAVE statement appears without a list of symbol names then all local storage is allocated to the linkage frame.

A further discussion of local storage allocation will be found in Section 18.

EXTERNAL PROCEDURE STATEMENTS

CALL Statement

CALL subroutine [argument-1, argument-2,..., argument-n]

Where subroutine is a subroutine name and the arguments are a list (possibly empty) of the arguments passed and to be returned. Subroutines may not CALL themselves unless the program units are all compiled with the DYNM parameter (64V mode on Prime 400 or higher computers).

EXTERNAL Statement

EXTERNAL V1, V2,..., Vn

Where each V is declared to be an external procedure name. This permits the name of an external function (such as COS) to be passed as an argument in a subroutine call or function reference.

DATA DEFINITION STATEMENT

DATA Statement

DATA k1/d1/,k2/d2,...kn/dn

Allows initialization of variables or array element at load time. Each k is a list of variables or array elements (with constant subscripts) separated by commas; each d is a corresponding list of constants of the same data mode as the variables and array elements in the list.

COMPILATION AND RUN-TIME CONTROL STATEMENTS

The following statements provide diagnostic tools for the programmer and are discussed in more detail in the Debugging section and the Compiler Section (4).

FULL LIST Statement

Causes a listing of subsequent source code with a symbolic listing. Overridden by compiler parameters.

LIST Statement

Causes a listing of subsequent source code with no symbolic listing. Overridden by compiler parameters.

INSERT Statement

See $INSERT.

NO LIST Statement

Causes a cessation of subsequent source code listing and of symbolic listing. Overriden by compiler parameters.

FULL LIST, LIST, AND NO LIST may be used anywhere in the source program.

TRACE Statements

```
TRACE V1, V2,...Vn
```

Item trace: Each V is a variable or array name. Prints the value of the variable at each point in the program where the variable is modified. Printout of a variable may be altered by another TRACE command with that variable name. Trace coding is inserted into the program at compilation; TRACE takes effect in source program physical order, not logical execution order.

```
TRACE n
```

Area Trace: Causes values of the variables used in statement label n to be printed out during execution of the code between the area TRACE statement and statement label n.

Note

Do not place an area trace statement in the range of another area trace statement, unless both refer to the same statement label.

TRACE is overridden by the compiler global trace parameter (see Section 4). It is possible to have the TRACE output written into a file instead of at the user's terminal. Prior to executing the program, switch the output to a file by the PRIMOS-level command.

```
COMO filename
```

where filename is the file into which terminal output is to be written. After the program has halted, output to filename is stopped and the file closed by:

```
COMO -END
```

The form of the command given here does not turn off output to the terminal. A complete description of this command is given in the PRIMOS INTERACTIVE documentation.

$INSERT Statement

```
$INSERT treename
```

Insert into the program, at compilation time, the file with specified treename. The $INSERT command should not be nested; do not include a $INSERT command in a file which will be inserted into a program by a $INSERT command.

$INSERT is used for:

Insertion of COMMON specification into programs.

Commonly used one-line functions.

Data initialization statements.

ASSIGNMENT STATEMENTS

Assign a value to a variable

1. arithmetic A=B**2
2. logical (P, Q, R are logical variables)

```
P=Q.OR.R
P=A.GT.B
```

Mixed Mode

Data of different modes may be combined with one another with the following restrictions:

1. Logical data should not be combined with any other mode.

2. No operator can combine Double Precisions and Complex data.

3. Subscripts and Control statement indexes must be integers (short or long).

4. Arguments of functions and subroutines must be of the mode expected by the called subprogram.

It is convenient to think of the arithmetic data modes as forming a hierarchy:

COMPLEX or DOUBLE PRECISIONS

REAL

LONG INTEGER

SHORT INTEGER

Whenever two data of differing modes are concatenated by an operator, the resulting mode is that of the higher in the list, as in:

REAL + SHORT INTEGER is a REAL

CAUTION

If LONG INTEGERS are converted to REALs, there may be a loss of precision. The rules for data mode conversion via assignments (i.e., A=B) are given in Table 16-1. Conversion of long (short) to short (long) integers by assignment is not recommended as good practice; use the INTL and INTS functions instead.

CONTROL STATEMENTS

ASSIGN Statement

```
ASSIGN k TO i
```

Where k and i are integer variables whose values are statement label numbers. An ASSIGN statement must be executed prior to an assigned GO TO.

CONTINUE Statement

```
[statement-number] CONTINUE
```

Transfers control to the next executable statement. With the optional statement-number, it is usually used to indicate the end of the range of a DO loop.

DO Statement

```
DO n i=m1, m2 [,m3]
```

Table 16-1. Data Mode Rules for Assignment Statements (A=B)

FROM B (right-hand-side)	TO A (left-hand-side)				
	Integer, Short	Integer, Long	Real	Double Precision	Complex
Integer, Short	Assign	Sign-Extend and Assign	Float and Assign	DP Float and Assign	Float and Assign to Real Part (Imaginary Part is Zero)
Integer, Long	Truncate and Assign	Assign	Float and Real Assign	DP Float and Assign	Float and Assign to Real Part (Imaginary Part is Zero)
Real	Fix and Assign	Fix and Assign	Assign	DP Evaluate and Assign	Assign to Real Part (Imaginary Part is Zero)
Double Precision	Fix and Assign	Fix and Assign	DP Evaluate and Real Assign	Assign	NOT ALLOWED
Complex	Fix and Assign Real Part	Fix and Assign Real Part	Assign Real Part	NOT ALLOWED	Assign

Assign: Transmit resulting value without change
Real Assign: Transmit as much precision of the most significant part of the resulting value as Real datum can obtain.
DP Evaluate: Evaluate, then DP float.
Float: Transform value to Real datum form.
DP Float: Transform value to Double Precision form.
Fix: Truncate fractional part and transform integral part to integer.
Truncate: Take 16 low-order bits and store in short integer datum.
Sign-Extend: Pad 16 high-order bits with 0's or 1's if short integer is positive or negative, respectively.

Executes statements until and including the statement with label n; m1, m2, m3 are positive integers (constants, parameters, or variables only - no expression or array elements) with m2>m1; i is an integer variable which assumes the values m1, m1+m3, m1+2*m3, etc. m1 is the initial value, m2 the limit value, and m3 the increment. If m3 is not specified, the increment is defaulted to 1.

DO loops may be nested; there is no syntactical limit to the nesting of DO loops.

It is an undesirable programming technique to have the index variable appear as the initial, limit, or increment values in the DO statement.

After the last execution of the loop, control passes to the next executable statement following the terminal statement of the DO loop. This is called a normal exit.

CAUTION

ANSI standard FORTRAN specifies that the value of the index variable is undefined after a normal exit from a DO loop. The value of the index variable at this point is completely dependent upon the specific compiler and how it performs its limit tests; hence, the terminal value of the index variable will differ at different installations. It is extremely bad programming to use the terminal value of this variable as implicitly set. If the user needs the value of this variable after a normal exit, its value should be explicitly set by an assignment statement.

Note

The DO loop in Prime FORTRAN is a one-trip DO loop. That is, the loop commands will be executed at least once even if the initial value is not less than the limit value. If it is desired to skip the loop under certain conditions, an IF statement preceding the DO statement should be used. Control should be transferred to a statement subsequent to the terminal statement of the DO loop, NOT to the terminal statement.

END Statement

The final statement of program, subroutine, or external function. Tells the compiler that it has reached the end of the source program.

GO TO Statements

GO TO k

Unconditional. Transfers control to statement labelled k.

GO TO (k1, k2,...,kn), i

Computed. Transfers control to statement labelled kj when integer expression i = j. If the value of i lies outside the range 1 to n, then control passes to the next executable statement after the computed GO TO.

GO TO i, [(k1, k2..., kn)]

Assigned. Transfers control to statement labelled i. Prior to executing, the assigned GO TO a value must be assigned to i using the ASSIGN command.

There is no syntactical limit to the number of labels in a computed or assigned GO TO.

IF Statement

IF (arithmetic expression-e) k1, k2, k3

Arithmetic. Where e is an arithmetic expression with an integer, real, or double precision value. If e<0 (negative) control is transferred to statement labelled k1, if e = 0 (exactly), control is transferred to statement labelled 2, and if e>0 (positive) control is transferred to statement labelled k3.

IF (logical expression-e) statement

Logical. Where e is a logical expression which may be .TRUE. or .FALSE.; statement is any valid executable statement except a DO or a logical IF statement. If e is true, the statement is executed; if e is false, control passes to the next executable statement.

Note

An arithmetic IF may be the statement in a logical IF but this is not recommended as a good programming practice.

PAUSE Statement

PAUSE [n]

Where [n] is an optional decimal number of up to five digits. Halts the program, transfers control to subroutine F$HT and prints ****PA n (R-identity) or ****PAUSE n (V-identity) at the keyboard. The value of n is printed in octal representation. Keying in START continues operation of the program at the next executable statement following PAUSE.

RETURN Statement

Returns to the main program from a subroutine or external function. It must be the last logical statement in the subroutine or external function.

STOP Statement

STOP [n]

Where [n] is an optional decimal number of up to five digits. Halts the program, transfers control to subroutine F$HT, prints ****ST n (R-identity) or ****STOP n (V-identity) at the keyboard and returns control to the PRIMOS level. The value of n is printed in octal representation.

INPUT/OUTPUT (I/O) STATEMENTS

(See Table 16-2 for list of FORTRAN device units.)

PRINT Statement

PRINT f {,list}

~~List~~ prints the list of elements on the user's terminal according to the format specified in statement f. Equivalent to WRITE (1,f) {list}.

READ Statements

For all READ statements: if END = a is included, then control is transferred to statement number a if an end-of-file condition is encountered during the read. If ERR = b is included, then control is transferred to statement number b if a device or format error is encountered during the READ statement.

list - is a list of variables and array names (separated by commas) into which data are read.

READ (u, f, [, END = a] [, ERR = b]) list

Formatted READ. Causes data on FORTRAN unit u to be read into the variables/array names specification list according to the format of statement f. If no list is given, one record is read and ignored.

CAUTION

Hollerith formats should be avoided in FORMAT statements associated with READ statements. The A format should be used for strings.

READ (u [, END = a] [, ERR = b]) list

Binary READ. Causes data on FORTRAN unit u to be read into the variables/array names specification list. Enough records are read to satisfy all the list items. If more items are on the record than are required by the list, the excess items are ignored. If no list is given, one record is read and ignored.

DEFAULT

Table 16-2. Devices and Their^FORTRAN Unit Numbers

FORTRAN Number (Unit No.)	Device
1	user terminal
2	paper tape reader or punch
3	MPC card reader
4	serial line printer
5	Funit 1
6	Funit 2
7	Funit 3
8	Funit 4
9	Funit 5
10	Funit 6
11	Funit 7
12	Funit 8
13	Funit 9
14	Funit 10
15	Funit 11
16	Funit 12
17	Funit 13
18	Funit 14
19	Funit 15
20	Funit 16
21	9-track magnetic tape unit 0
22	9-track magnetic tape unit 1
23	9-track magnetic tape unit 2
24	9-track magnetic tape unit 3
25	7-track magnetic tape unit 0
26	7-track magnetic tape unit 1
27	7-track magnetic tape unit 2
28	7-track magnetic tape unit 3

CAUTION

If the list requires more data than are in the current record, then the next record(s) are read until the list is satisfied. This is not a clean programming technique and should be avoided.

READ (u,* [, END = a] [, ERR = b]) list

List-directed READ. List-directed I/O frees the programmer from including format statements for READs from free-format input devices such as the user terminal. The input data is converted according to the data type of items in the I/O list. Additionally, this feature provides a method to indicate in the input data that an item in the I/O list is to remain unchanged by the READ statement.

Delimiters: Values in list-directed input are separated by a blank, comma, or slash. A slash or comma may be preceded and followed by any number of blanks. An end of record is treated as a blank. A slash terminates a READ and leaves the values of the remaining items in the I/O list unchanged. Two adjacent commas with no intervening characters except blanks will leave the corresponding item in the I/O list unchanged. A list-directed READ will read any number of records until a slash is encountered or until all items in the I/O list have been satisfied.

Examples:

1. Source line: READ(1,*)A,B,C
   Input Data: 151,,2E2
   Result: A=151.
   B is unchanged
   C=2.E2

2. Source line: READ(1,*)I,J,K
   Input Data: 5 -3 /
   Result: I=5
   J=-3
   K is unchanged

Numerical Input: If an item in the I/O list is a long or short integer variable or array element, the corresponding input field must contain a string of decimal digits optionally preceded by a + or - sign, as in:

-357 100514 +12387

If a real or double precision item is in the I/O list, the corresponding input field must contain a string of decimal digits with an optionally embedded decimal point. An exponent field may follow in either E or D format, as in:

51 -27.68 7.65E-14 863D2
.503 +265.

The input field corresponding to a complex item must contain two real numbers (as described above), separated by comma and enclosed in parentheses, as in

(1E2,-2.) (5.67E-6,8.09)

Character String Input: A variable or array of any type can be set equal to a character string using list-directed READ. A character string must be enclosed in single quotation marks in the input data. Within a character string, a quotation mark is represented by two consecutive quotation marks. A character string, regardless of length, matches a single item in the I/O list whether it is a variable, array element, or whole array (represented by including the unsubscripted array name in the I/O list). If the character string is shorter than the list item, the rightmost characters of the list item are blank-filled. If the character string is longer than the list item, the rightmost characters of the character string are ignored. Characters are packed two per word, as in:

1. Source: INTEGER*2 IBUF(2)
   READ(1,*) IBUF
   Input Data: 'ABC'
   Result: IBUF(1)=AB
   IBUF(2)=C

2. Source: READ(1,*) (IBUF(I), I=1,2),J
   Input Data: 'GHIJ', 5 /
   Result: IBUF(1)='GH'
   IBUF(2)=5
   J is unchanged

Note

If the I/O list has been satisfied, a slash in the input data is optional. A carriage return is the end of a record on a READ from a user terminal and is treated as a blank on list-directed READs.

WRITE Statements:

For all WRITE statements: If ERR=b is present, control is transferred to statement b if a device error is encountered during the WRITE statement.

list - is a list of variables and array names (separated by commas) from which data are printed.

WRITE (u,f [,ERR=b]) list.

Formatted WRITE. Causes data in the list to be written out on FORTRAN unit u according to the format statement f.

WRITE (u [,ERR=b]) list

Binary WRITE. All words in the list are written into a record in binary format. If there are insufficient data to fill the record, it is padded out with zeroes; if there are more items than a record can hold, multiple records are written automatically. If necessary, the last record is padded with zeroes.

Both READ and WRITE statements allow implied DO loops for transferred data between arrays and devices. In this case, the list could have a form such as:

```
(NAME1 (INDEX1), INDEX1 = 1, 5, 2)

                OR

(NAME1 (INDEX1), NAME2 (3, INDEX1), INDEX1 = 1, 5)

                OR

(NAME1 (INDEX1, INDEX2), INDEX 1 = 1, m), INDEX2 = 1, n, p)
```

where m, n, and p are constant positive integers (constants, parameters, or variables).

CODING STATEMENTS

c - number of ASCII characters to be transferred
f - format statement label
a - array name
list - I/O list of elements (same as in a READ or WRITE statement)

DECODE Statements:

```
DECODE (c,f,a[, ERR=sn])list
```

Formatted DECODE. Converts the first c characters in the array a from ASCII data into the I/O list elements according to the specified format f. If the optional error branch is inserted, a FORMAT/DATA mismatch will cause a transfer to the statement labelled sn.

```
DECODE (c, *, a [, ERR=sn])list
```

List-directed DECODE. Allows the user to input/decode data from free-format input devices such as the user terminal. The requirements on input and delimiters are the same as for the list-directed READ statement (see READ).

ENCODE Statement:

```
ENCODE (c,f,a) list
```

Converts the elements of the I/O list into ASCII data according to format f and stores the first c characters of the resultant string into array a.

FORMAT STATEMENTS

FORMAT Statement:

sn FORMAT (dF1 dF2 dF3...Fn)

sn - mandatory statement number
Fl - etc. - is a format field description
d - is a format delimiter (, or /). (The first d may be null)

The right parentheses marks the end of a record.

Delimiters:

/ (slash) - proceed to next record
, (comma) - remain within current record

The maximum record length is determined by the type of device or storage unit.

Format Field Descriptor: Tables 16-3 and 16-4 summarize the field descriptors available in Prime FORTRAN IV, where n (positive integer constant) is the number of times the basic field descriptor to be repeated, w (positive integer constant) is the total width of the field in columns (or characters).

d (non-negative integer constant) is the number of digits to the right of the decimal point. (See format G output for an exception to this).

Repetition: All field descriptors except those marked by an * in Tables 16-3 and 16-4 (X,H,B) can be assigned a repeat count causing the descriptor to be used that number of times in succession.

```
FORMAT (3E10.5)
```

and

```
FORMAT (E10.5, E10.5, E10.5)
```

are equivalent.

Groups of descriptors (including X,H,B) may be enclosed in parenthesis and the entire group assigned a repeat count.

```
FORMAT (2(3G11.6,5X))
```

and

```
FORMAT (3G11.6,5X,3G11.6,5X)
```

are equivalent.

Repeat groups have a maximum nesting of two levels.

```
FORMAT (3(2(10F.7,3X),I2,5X))
```

is permissable.

Table 16-3. Results of Formats in Output Statements

FORMAT	OUTPUT
snFw.d Floating	Prints Real or Double Precision Numbers as mixed output (no exponent) with as many significant figures as the data type allows. w is the total field width and must allow one position for a decimal point and one for a minus sign (if negative numbers are to be printed). d is the number of decimal places (right of decimal point). Numbers are right justified. Leading zeroes are inserted for numbers less than 1; trailing zeroes are used to fill the decimal places if necessary. Only minus signs are printed. If total field width is too small, the number is truncated and a $ printed if positive, a = if negative. If the decimal section is too small, the number is rounded.
snEw.d Exponential	Prints Real or Double Precision numbers as a number with a magnitude between 0.1 and 0.9999999 times an exponent. The field width w must allow for a minus sign (if one is to be printed), a decimal point, E (the exponent), a blank or a minus sign, and one or two positions for the exponent value. The number d sets the number of places to the right of the decimal point - the maximum is seven. The representation with magnitude less than 1 may be overridden using scale factors.
snGw.d	Prints Real or Double Precision numbers in F or E format according to the magnitude of the number and the decimal place specifier - d. Magnitude — Effective Format 0.1 to 1.0 — F(w-4).d,4X 1.0 to 10.0 — F(w-4).(d-1), 4X . . . 10**(d-2) to 10**(d-1) — F(w-4).1, 4X 10**(d-1) to 10**d — F(w-4).0, 4X Outside Range — Ew.d
General	Truncation is performed as for E and F formats
snDw.d Double Precision	Prints Double-Precision Numbers only in an exponential format similar to the E format except that the letter D is used instead of E and that d has a maximum value of 14.

Table 16-3. Results of Formats in Output Statements (Cont)

FORMAT	OUTPUT
wX * Space	Writes w spaces into the output record (negative w backspaces for replacing).
Tw Tab	Positions output pointer to column w in the output record. Back tabbing is permitted. Example: (T1,40A2,T15,F9.3)
wHc1c2...cw * Hollerith	Prints the string c1c2...cw 1. Does not require an item in the output list 2. Need not be followed by a delimiter
nAw ASCII	Prints Integer, Real, Complex, or Double Precision variables as ASCII characters. w is number of of characters per variable or array name. Output is left justified and padded with spaces.
nLw Logical	Prints logical variables: +1 prints as T, 0 prints as F. Output is right justified and padded with spaces. If w<1 there is no output.
nIw Integer	Prints contents of integer (short or long) variables or array names as a string of integers (no decimal points). If string is longer than field width w then number is right truncated and preceded by a $ if positive and = if negative. Minus signs are printed but not plus signs.
B'string' * Business	Prints templated numerical output for business purposes. Features include: Fixed and floating signs, trailing signs, plus sign suppression, trailing minus change to 'CR', fixed & floating $, field filling, leading zero suppression, insertion of commas. Length of string determines field width; if number is greater than field width then output is printed as string of asterisks. See text for details on this format

* No repeat count is allowed associated with the format specifier itself, but the format specifier may be included in a group repetition.

Table 16-4. Results of Formats in Input Statements.

FORMAT	INPUT
snFw.d Floating	External numbers may be represented as integers mixed integers, or scaled numbers (with exponents). Leading blanks are treated as zeroes; imbedded and trailing blanks are ignored. The implied decimal point is placed to the left of the first d digits counting from the right (if no decimal point in the external number). A decimal point in the external number overrides the positional decimal point. The decimal exponent (D or E) and the exponent value are a unit; both must be included or omitted. All numbers are assumed positive unless a minus sign is present.
snEw.d Exponential	All numbers are initially converted internally to double-precision numbers; if entered in E,F, or G format they are truncated.
snGw.d General	
snDw.d Double-Precision	
wX * space	Skips w columns in the input data (negative w backspaces to reload record)
Tw Tab	Tabs to column w in the input record.
wHc1c2...cw * Hollerith	NOT USED
nAw ASCII	Stores ASCII characters in Integer, Real, Complex, or Double-Precision variables. If input is greater than storage available in variables, only the leftmost characters are stored.

Table 16-4. Results of Formats in Input Statements (Cont)

FORMAT	INPUT
nLw Logical	Stores true/false in internal representation based upon first non-space characters in the input data (all others ignored). If T-set to +1; if F-set to 0; if anything else-set to 0 and set error flag (use OVERFL to look at error flag).
nIw Integer	Stores external numbers in integers. If no sign is present, a plus sign is assumed. A sign or blank is counted as one character position. No decimal points are allowed. If there are more numbers than the field width, w, only the left-most w characters are stored.
B'string' * Business	NOT USED

* No repeat count is allowed associated with the format specifier itself but the format specifier may be included in a group repitition.

Rescanning Format Lines: If the format list is exhausted before the input/output list, the format list is repeated. Repetition starts at the opening (left) parenthesis that matches the last closing (right) parenthesis in the format list. The parentheses around the format list itself are used only if there are no other parentheses. Any repeat count preceding the rescanned format are in effect.

Output - the current record is padded with blanks and a new record is started.

Input - the remainder of the current record is skipped and the device advanced to the beginning of the next record.

Formats as Variables: It is possible to enter format statements at run time by any method of building it as text string and loading it into an array. The array can later be referenced in lieu of a FORMAT statement, by the READ or WRITE statement that handles the data. Arrays to be used for this purpose must be assigned as integer type and must be dimensioned to accomodate the format description, at two characters per word. The format description is loaded into the array by a READ statement that references a type A format statement:

```
      DIMENSION FORM (6), TEXT (80)
      INTEGER FORM
      READ (1,20) FORM
   20 FORMAT (6A2)
      WRITE (1,FORM) (ARG (I) , I=1,3)
```

These statements provide for an output format specification such as (3(F7.3,I7)) to be entered at run time. Note that the specification must include opening and closing parenthesis but not the word FORMAT.

B-Format: The B-Format is used in printing business reports where it is desirable to fill number fields to prevent unauthorized modifications (as on checks), suppress leading zeroes and plus signs, print trailing minus signs (accounting convention) and convert minus signs to CR (for indicating credit entries on bills). The form of the B-field specifiers is:

```
B 'string'
```

The length of the string determines the field width. If the width is too small for the number, then the output will be a string of asterisks filling the field. Legal characters for the string are:

```
+ - $ , * Z # . CR
```

Plus (+):

If only the first character is +, then the sign of the number (+ or -) is printed in the leftmost portion of the field. (Fixed sign) If the string begins with more than one + sign, then these will be replaced by printing characters and the sign of the number (+ or -) will be printed in the field position immediately to

the left of the first printing character of the number (floating sign). If the rightmost character of the string is +, then the sign of the number (+ or -) will be printed in that field position following the number (Trailing sign).

Minus (-):

Behaves the same as a plus sign except that a space (blank) is printed instead of a + if the number is positive (Plus sign depression).

Dollar Sign ($):

A dollar sign ($) may at most be preceded in the string by an optional fixed sign. A single dollar sign will cause a $ to be printed in the corresponding position in the output field. (Fixed dollar).

Multiple dollar signs will be replaced by printing characters in the number and a single $ will be printed in the position immediately to the left of the leftmost printing character of the number. (Floating dollar)

Asterisk (*):

Asterisks may be preceded only by an optional fixed sign and/or a fixed dollar. Asterisks in positions used by digits of the number will be replaced by those digits; the remainder will be printed as asterisks. (Field filling)

Z:

If the digit corresponding to a Z in the output number is a leading zero, a space (blank) will be printed in that position; otherwise the digit in the number will be printed. (Leading-zero suppression)

Number sign (#):

#'s indicate digit positions not subject to leading-zero suppression; the digit in the number will be printed in its corresponding portion whether zero or not. (Zero non-suppression)

Decimal point (.):

Indicates the position of the decimal point in the output number. Only #'s and/or either trailing signs or credit (CR) may follow the decimal point.

Comma (,):

Commas may be placed after any leading character, but before the decimal points. If a significant character of the number (not a sign

or dollar) precedes the comma, a , will be printed in that position. If not preceded by a significant character, a space will be printed in this position unless the comma is in an asterisk field; then an * will be printed in that position.

Credit (CR)

The characters CR may only be used as the last two (rightmost) of the string. If the number is positive, 2 spaces will be printed following it; if negative, the letters CR will be printed.

See Table 16-5 for examples of B-Format usage.

Scale Factors (D,E,F, and G Formats): A scale factor designator for use with the F,E,G, and D descriptors causes a multiplication by a power of 10. The form is:

nP (represented as s in Tables 16-3 and 16-4)

Where n, the scale factor, is an integer constant with an optional minus sign. Once a scale factor has been specified, it applies to all subsequent F,E,G, and D field descriptors, until another scale factor is encountered. If n=0, an existing scale factor is removed. The scale factor has no effect on type I,A,H,X,L, or B descriptors.

E and D Output Scale Factor: Before output conversion, the fractional part of the internal number is multiplied by 10**n and the exponent is decreased by n.

F Output Scale Factor: The internal number is multiplied by 10**n.

G Output Scale Factor: The scale factor has an effect only if the internal number is in a range that uses effective E conversion for output. In this case, the effect of the scale factor is the same as in the corresponding E conversion.

D,E,F,G, Input Scale Factor: The internal value is formed by dividing the external number by 10**n. However, if the external number contains a D or E exponent, the scale factor has no effect.

Formatted Printer Control: The first character of each ASCII output record controls the number of vertical spaces to be inserted before printing begins on the line printer.

Table 16-5. Examples of B-Format Usage.

Number	Format	Output Field
123	B'####'	0123
12345	B'####'	****
0	B'####'	0000
123	B'ZZZZ'	123
1234	B'ZZZZ'	1234
0	B'ZZZZ'	
0	B'ZZZ#'	0
1.035	B'#.##'	1.04
0	B'#.##'	0.00
1234.56	B'ZZZ,ZZZ,ZZ#.##'	1,234.56
123456.78	B'ZZZ,ZZZ,ZZ#.##'	123,456.78
0	B'ZZZ,ZZZ,ZZ#.##'	0.00
2	B'+###'	+002
-2	B'+###'	-002
2	B'-ZZ#'	2
-2	B'-ZZ#'	- 2
234	B'ZZZZZ+'	234+
-234	B'ZZZZZ+'	234-
234	B'ZZZZZ-'	234
-234	B'ZZZZZ-'	234-
12345	B'ZZZ,ZZ#CR'	12,345
-12345	B'ZZZ.ZZ#CR'	12,345CR
123	B'+++,++#.##'	+123.00
-123	B'+++,++#.##'	-123.00
98	B'$ZZZZZZ#'	$ 98
98	B'$$$$$$$#'	$98
156789	B'$***,***,**#.##'	$****156,789.00

First Character	Effect
Space	one line
0	two lines
1	form feed - first line of next page (effective only on devices with mechanized form feed.)
+	no advance - print over previous line (line printer only)
Other	one line

In the case of space, 0,1, and +, the control character is not printed, on all other cases, the character is printed as well as spacing a line.

DEVICE CONTROL STATEMENTS

For physical positioning of sequential access devices.

BACKSPACE Statement (for magnetic tape unit only)

```
BACKSPACE u
```

Repositions FORTRAN unit u so that the preceding record is now the next record. If the unit is at its initial point, this command nas no effect. Backspace has no effect on disk files.

ENDFILE Statement

```
ENDFILE u
```

Writes an endfile record on FORTRAN unit u indicating the end of a sequential file for magnetic tape. Closes a disk file on FORTRAN unit u.

REWIND Statement

```
REWIND u
```

Repositions FORTRAN unit u to its initial point. Does not close or truncate disk file.

FUNCTION CALLS

Functions are called by means of assignment statements in which the right had side is an expression of the form:

```
name (argument-1,argument-2,...argument-n)
```

Where name is the name of the function called (COS, SIN, etc.) and argument is a non-empty list of arguments to the function separated by commas. The data modes of the arguments must be the same as the data modes in the definition of the function. There is no syntactical limit to the number of arguments.

SUBROUTINE CALLS

Subroutines are called from a program by the statement:

CALL name (argument-1,argument-2,...,argument-n)

Where name is the symbolic name assigned by the SUBROUTINE statement beginning the subroutine subprogram. The argument is a list of arguments, some of which are passed to the subroutine by the calling program, and the remainder are dummy arguments whose values are calculated by the subroutine and returned to the main program. The arguments in the main program must agree in number, order, and mode with the arguments used in the subroutine subprogram. There is no syntactical limit to the number of arguments.

CAUTION

Do not place constants in the argument list of a subroutine or function in the position where a value is to be returned to the calling program. This will cause the constant to be altered and produce undesirable results.

SECTION 17

FORTRAN FUNCTION AND SUBROUTINE STRUCTURE

FUNCTIONS

There are four types of functions; all are called in the same manner (see Section 16).

Prime FORTRAN Library Functions

These are a collection of library subprograms (see Section 20) which are called during compilation and appended to the main program during loading.

Prime Extended Intrinsic Functions

These are a collection of functions designed to increase the efficiency of Prime FORTRAN IV in logical processing of integers. They are inserted in the program by the compiler.

User-Defined Function Subprograms

FUNCTION subprograms can consist of many statements, coded and compiled separately. This permits them to be used in the same way as library functions.

FUNCTION subprograms must be prepared as separately compiled subprograms that produce a single result, in the following format:

```
mode FUNCTION name (argument-1, argument-2,...argument-n.)
    .
    .
(Any number of FORTRAN statements which perform the required
calculations, using the supplied arguments as values.)
    .
    .
    .
name = Final calculation
    .
    .
    .
RETURN
```

FUNCTION Statement: The FUNCTION statement, which must be the first statement of a FUNCTION subprogram, assigns the name of the function and identifies the dummy arguments. In the preceding example, name is a symbolic name assigned to identify the function, and each argument is a dummy argument. There is no syntactical limit to the number of arguments. The function name must conform to the normal rules for all symbolic names with regard to number of characters, etc. Implicit result mode typing occurs according to the first letter of the name. Implicit mode typing can be overridden by preceding the word FUNCTION with one of the mode specifications. The function name must differ from any variables used in the function subprogram or in any main program which references the function.

Body of Subprogram: The body of the function subprogram can consist of any legal FORTRAN statements except SUBROUTINE, BLOCK DATA, or other FUNCTION statements. The statements that evaluate the function use constants, parameters, variables, and expressions in the normal way. The program must produce a single result for a given set of argument values. The subprogram must equate the assigned symbolic function name to the result, by using name on the left side of an assignment statement. It is the function name itself, used as a variable, that returns the result to the main program.

RETURN Statement: The RETURN statement consists of a single word RETURN. It terminates the subprogram and returns control to the main program. The RETURN statement must be the last statement in the subprogram (logically, not physically; that is, it must be the last statement to which control passes).

Statement Functions

Statement functions are embedded in the coding of the main program and are compiled as part of the main program. Any calculation that can be expressed in a single statement, and produces a single result, may be assigned a function name and referenced in the same way as a library function. A statement function is defined in the form:

name (argument-1, argument-2,...argument-n) = Expression

where name is the symbolic name assigned to the function and each argument is a dummy variable that represents one of the arguments.

The following rules apply to all functions:

1. The name may consist of one to six alphanumeric characters, the first of which is alphabetic. It must differ from all other function names and variable names used in the main program.

2. The argument list follows the name and is enclosed in parentheses. There must be at least one argument. Multiple arguments are separated by commas. Each argument must be a single nonsubscripted

variable. These arguments are only dummy variables, so their names may be the same as names appearing elsewhere in the program. The dummy variable names do indicate argument mode, however, by implicit or explicit mode typing. There is no syntactical limit to the number of arguments.

3. During each call of a function, the values supplied as the argument variables must be in the same mode as the arguments were when the function was defined.

4. Implicit mode typing of the result of a function is determined by the first letter of the function name. Functions that begin with I,J,K,L,M, or N produce INTEGER results; others produce REAL results. Regardless of the first letter, the result mode can be set by an appropriate mode specification preceding the FUNCTION statement.

5. The expression that defines the function may use library functions, previously defined function statements, or FUNCTION subprograms; but not the function itself. Dummy variables cannot be subscripted.

6. Variables in the expression that are not stated as arguments are treated as coefficients - i.e., are assumed to be variables appearing elsewhere in the main program.

7. Statement functions must be defined following specification and DATA statements but before the first executable statement of a program.

SUBROUTINES

Some types of subroutines include:

PRIMOS System Subroutines

These invoke the PRIMOS system to perform the actual work. They allow file transfer, attaching, etc. (See Section 20 and REFERENCE GUIDE, FILE MANAGEMENT SYSTEM (FMS), PDR3110).

Application Library Subroutines

These handle file manipulation (opening and closing, reading, and writing, etc.) and data transfers, greatly enhancing the capability of the FORTRAN language (Section 20 and REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106).

FORTRAN Math Subroutines

These handle mathematical calculations such as matrix multiply and inversion permutations, etc. (See Section 20.)

User-Defined Subroutines

Called in the same manner as those supplied with the system. They are constructed as follows:

```
SUBROUTINE name (argument-1, argument-2, ...argument-n)
    .
    .
(any number of FORTRAN statements which perform the required calcula-
tions, using the supplied arguments, if any, as values).
    .
    .
RETURN
END
```

SUBROUTINE Statement: The SUBROUTINE statement, which must be the first statement of a SUBROUTINE subprogram, assigns the name of the subprogram and identifies the dummy arguments, if any.

The subprogram name must conform to the normal rules for symbolic names with regard to number of characters, but the first letter does not set the data mode of the results. The name must be unique to both the subprogram and a main program which calls it.

The argument list usually consists of a series of dummy variables which are processed by the subroutine and return arguments to the main program. Each argument may be a variable, array, or function name. If an argument is the name of an array, it must be mentioned in a DIMENSION statement following the SUBROUTINE statement.

There is no syntactical limit to the number of arguments. A subroutine with no arguments is allowable. Such a subroutine might obtain arguments from, and return results to, COMMON. Or it might be used to output a message or control function to a peripheral device.

CAUTION

Arguments that return values to the main program must not be constants or expressions in the calling sequence.

Body of a SUBROUTINE: The body of the subroutine can consist of any legal FORTRAN statements except SUBROUTINE, BLOCK DATA, or FUNCTION statements. The results of calculations may be stored in variables used by both the subprogram and main program, or they may be placed in COMMON. Variables may be used freely on either the right or left side of the equal sign in assignment statements. Each variable that represents a result must appear on the left side of at least one assignment statement, in order to present the result to the main program.

The subroutine is terminated by a RETURN statement (described previously). The last physical statement in a subroutine must be an END statement.

PART V

UTILITY REFERENCE

SECTION 18

COMPILER REFERENCE

PRIME FORTRAN COMPILER PARAMETERS

All parameters are preceded by a dash, "-", in the command line. Parameters that are the PRIME-supplied default parameters (i.e., those that need not be included) are indicated. The system manager may have changed the defaults; if so, the programmer should obtain a list of the installation-specific defaults. (See figure 18-1).

BIG

Treats all dummy arrays as arrays that span segment boundaries and also sets the compiler to produce 64V mode object code. If a dummy argument array may become associated with an array spanning a segment boundary (through a subroutine CALL statement or function reference) the compiler must be made aware of this by including BIG in the parameter list. The code generated here will work whether or not the array actually spans a segment boundary. See also NOBIG, 64V. See Section 12 for more information on this requirement.

B[INARY] { treename / YES / NO }

Specifies the binary (object) output file. If treename is given, then 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. Omitting the parameter is equivalent to the inclusion of -BINARY YES. (See Table 18-1.)

DCLVAR

Flags undeclared variables. If included in the parameter list, the compiler will generate an error message when a variable is used in the program, but not included in a specification statement. The message will be generated once per undeclared variable. See NODCLVAR, SPO.

Table 18-1. Compiler File Specifications

Compiler Mnemonics	INPUT or SOURCE	LISTING	BINARY
treename	looks for file named treename as source file	opens file named treename as listing file	opens file named treename as (object) file.
YES	not applicable	uses default filename for listing file. L←PROGRM	used default filename for binary file. B←PROGRM
NO	not applicable	no listing file.	no listing file.
TTY	compiler will compile program as entered from the terminal.	print listing on user terminal.	not applicable
SPOOL	not applicable	spool listing directly to line printer.	not applicable
option not invoked	source filename must be first option after FTN command.	same as NO	same as YES

To use other peripheral devices such as magnetic tape, card reader, or paper tape punch/reader for file location, see Table 18-2 for A- and B-register settings.

DEBASE

Conserves Loader base areas. When enabled, it reduces the sector zero requirements of large programs. The compiler generates double-word memory reference instructions and uses the second word as an indirect link for all references to the same item within the relative reach. Use of this option reduces sector zero usage by 70% to 80%. Programs compiled with this option can be loaded only in the relative addressing modes (32R or 64R) (a loader NS diagnostic is generated if an attempt is made to load in a sectored addressing mode).

DYNM

Enables local storage in Stack Frame (Prime 400 and higher only). Allows dynamic allocation of local storage and also sets the compiler to generate 64V mode object code. The DYNM parameter allows better memory utilization in the 64V mode. It also allows the creation of recursive FORTRAN subroutines (subroutines which call themselves). See SAVE, 64V.

ERRLIST

Prints only error messages in the listing file. See EXPLIST, LIST.

Note

This parameter has no effect unless an output device/file is specified using LISTING.

ERRTTY Default

Prints error messages at the user terminal. The normal system default causes each statement containing an error to be printed at the user terminal. This feature is especially useful when a corrected program is being recompiled, to confirm that the errors have been properly corrected. See NOERRTTY.

EXPLIST

Prints full listing in the listing file. The full listing consists of an assembly language type listing, the source statements (with line numbers), and error messages. See ERRLIST, LIST.

Note

This parameter has no effect unless an output device/file is specified using LISTING.

FP Default

Generate instructions from the floating-point skip set when testing the result of a floating-point operation.

I[NPUT] treename

Specifies the treename of the input source program (see Table 18-1). This parameter must not be used if the source filename immediately follows the FTN command; otherwise, it must be included in the parameter list. See SOURCE.

INTL

Long integer default. Sets the long integer (INTEGER*4) as the default for the INTEGER statement instead of the short integer (INTEGER*2). The normal INTEGER data type in Prime FORTRAN is a 16-bit word. A 32-bit integer data type is available through the use of the INTEGER*4 statement.

The long integer default parameter is used to simplify conversion of extant FORTRAN programs to Prime computers. When this is enabled all variables, arrays, and functions explicitly or implicitly specified as INTEGER will be 32-bit integers. All integer constants will be treated as 32-bit integers. Only names specifically appearing in INTEGER*2 statements will be 16-bit integers. The 32-bit integer has a greater range than the 16-bit integer (-2147483648 to 2147483647 vs. -32768 to 32767). The 32-bit integer has the same storage requirement as the REAL*4 (REAL) data type. See INTS.

CAUTION

FORTRAN requires that the type of actual argument in a function reference of CALL statement must agree with the corresponding dummy argument in the referenced subprogram. Note that a subprogram expecting a long integer must NOT be called with a short integer (and vice versa). Most Prime-supplied subroutines expect short integer arguments. Care should be taken when calling these routines (e.g., SEARCH) in a program compiled with the LONG INTEGER default options.

Example:

```
CALL SEARCH (INTS (1) 'FILENM', INTS (1))
```

INTS (long-integer) is a built-in function that converts its arguments to a short integer. If the INTS conversion functions are omitted, the integer constants

are compiled as long integers, providing INTL is included in the parameter list. Do not confuse the function INTS (long-integer) with the compiler parameter INTS.

INTS Default

Short integer default. Sets the INTEGER default to INTEGER*2 rather than INTEGER*4. See INTL.

LIST Default

Print source listing. Prints a listing of the source statements (with line numbers) and error messages in the listing file. See ERRLIST, EXPLIST.

Note

This parameter has no effect unless an output device/file is specified using LISTING.

L[ISTING] { treename / YES / NO / TTY / SPOOL }

Specifies the listing device/filename:

treename - opens this file for the listing.

YES - uses the default name for the listing file L←PROGRAM (where PROGRAM is the source).

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 NO parameter inclusion (i.e., no listing file is created).

NOBIG Default

Utilizes Relative Addressing. This is the usual memory addressing mode. See BIG.

NODCLVAR Default

Suppresses Undeclared Variable Flagging. Does not generate error messages when undeclared variables are detected. See DCLVAR.

NOERRITY

No Terminal Error Messages. Suppresses the printing of error messages on the users terminal. See ERRITY.

NOFP

Suppresses generation of floating-point skip instructions when testing the result of a floating-point operation. Include NOFP in the parameter list when compiling for machines that do not have the floating-point options. Without NOFP, the programs will still execute on such machines but the UII time will be longer. See FP.

NOTRACE Default

Suppresses Global Trace. Does not enable the global trace. See TRACE.

NOXREF Default

Suppresses Concordance. Do not generate any concordance (cross-reference) listing. See XREFL, XREFS.

SAVE Default

Local Storage Allocation. Performs local storage allocation statically. See DYNM.

S[OURCE]

Same as I[NPUT] (See INPUT).

SPO

Special Library Compilation (System Program Optimization). When the Library Mode parameter is included, certain statements and program formats that would normally be flagged as errors are permitted. It also causes re-interpretation of some statements. Finally, it enables the undeclared variable flag (DCLVAR). This parameter is used for the compilation of some Prime-supplied software and is not recommended for general use. See DCLVAR, NODCLVAR.

TRACE

Enable Global Trace. When this parameter is included, a trace printout is generated at all assignment statements and at every labelled statement in the program unit. The global trace affects only the program unit being compiled; it has no effect on other program units in the same executable program. See NOTRACE.

XREFL

Enable Full Concordance. Appends a full concordance (symbol cross-reference) listing to the end of the program listing. The full concordance includes all symbols in the program unit. See NOXREF, XREFS.

Note

This parameter has no effect unless an output device/ file is specified using LISTING.

XREFS

Enable Partial Concordance. Appends a partial concordance (symbol cross-reference) listing to the end of the program listing. The partial concordance does not include symbols that are referenced only in specification statements. See NOXREF, XREFL.

Note

This parameter has no effect unless an output device/ file is specified using LISTING.

32R Default

32K words (64K bytes) mode. In the 32R (default) mode 64K bytes of user space are available to each FORTRAN user. This space must accomodate the main program, subprograms, all local storage, library routines, and the COMMON blocks. More space is available to the user in the 64R and 64V modes. See 64R, 64V.

64R

64K words (128 bytes) mode. The mode gives the user 128K bytes of user space. All main programs and all subprograms executed must be compiled with the 64R parameter. When using the linking loader utility (LOAD), the MODE command must also be used to change the load mode to 64R. This assures the user of 128K bytes of user space. See 32R, 64V.

Generally, it can be determined if the 64R mode must be selected by looking at the storage areas. Each area requiring space such as the COMMON blocks can be examined. If the COMMON blocks require more than 64K bytes, then the 64R mode decision is obvious. For example, if it is on a segment boundary and a load is attempted resulting in an overflow, it is likely that the addresses for the COMMON are overlapping the program area.

64V

Segmented Memory Mode. Puts the FORTRAN user into the 64V Segmented Memory mode and allows the SEG utility to be used in lieu of the LOAD utility. This is for large 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 BIG, NOBIG, 32R, 64R.

The LOAD utility and load modes are dictated by the options selected at compile time, as shown in the following table:

Utility	Compiler Option	Load Option
LOAD	32R (default)	D32R (default)
	64R	D64R, D32R (default)
SEG	64V	64V (only mode)

Any PRIMOS system can use either the 32R or 64R addressing mode. Only a Prime 400 (and higher) can have 64V addressing mode.

EXPLICIT SETTING OF THE A AND B REGISTERS

Note

If you will not be using the paper tape punch/reader, card punch/reader or magnetic tape for I/O devices at compilation time you need not read this section.

Operation

The FORTRAN compiler is invoked by the FTN command to PRIMOS.

FTN treename [1/a-register] [2/b-register]

where treename is the treename of the FORTRAN source file;

a-register and b-register are the values of the A and B registers.

The default values of the registers are:

```
A '1707 (binary = 0000001111000111)
    Input file is on disk
    No listing file
    Binary file is on disk
    Print error messages at user terminal
    32R mode

B '0 (binary = 0000000000000000)
    Short integers
    No concordance
```

If the default values of a register are used that parameter may be omitted.

```
FTN treename                (default A and B registers)
FTN treename 1/a-reg        (default B register)
FTN treename 2/b-reg        (default A register)
```

For non-default values include both parameters:

```
FTN treename 1/a-reg  2/b-reg
             or
FTN treename 1/a-reg  b-reg
```

Spaces should be used to separate components of the command line. The bit values corresponding to the mnemonic parameters are given in Table 18-2.

Input/Output Specifications

Additional devices are accessible to users explicitly setting the A and B registers. I/O is specified by the A-register setting as:

Type	Bits
Input (source)	8-10
Listing	11-13
Binary (object)	14-16

The settings corresponding to I/O files and devices are given in Table 18-3.

Table 18-2. A-and B-register Bit Correspondences of Parameter Mnemonics (PRIME-supplied defaults are indicated)

A(x,y) = 0(or 1): the mnemonic parameter causes the value of bits x and y in the A register to be 0 (or 1).

B(x,y) = 0(or 1): same as above for the B register.

BIG	B(8,9) = 1
B[INARY]	A(14,15,16) = object file definition (see table 18-3); PRIMOS BINARY command
DCLVAR	B(16) = 1
DEBASE	A(6) = 1
DYNM	B(3,8) = 1
ERRLIST	A(3) = 1
ERRITY	A(7) = 1; default
EXPLIST	A(2) = 1
FP	B(15) = 0; default
I[NPUT]	A(8,9,10) = input file definition (see table 18-3)
INTL	B(10) = 1
INTS	B(10) = 0; default
L[ISTING]	A(11,12,13) = listing file definition (see table 18-3); PRIMOS LISTING command
NOBIG	B(8,9) = 0; default
NODCLVAR	B(16) = 0
NOERRITY	A(7) = 0
NOFP	B(15) = 1
NOTRACE	A(4) = 0; default
NOXREF	B(12,13) = 0; default
SAVE	B(3) = 0; default
S[OURCE]	A(8,9,10) = input file definition (see table 18-3); same as I[NPUT]
SPO	A(1) = B(16) = 1
TRACE	A(4) = 1
XREFL	B(13) = 1
XREFS	B(12,13) = 1
32R	A(5) = B(8) = 0; default
64R	A(5) = 1
64V	B(8) = 1

Table 18-3. Bit/Device Correspondences

Bits	Octal	Device	Mnemonic Parameter
000	0	None	NO
001	1	User terminal	TTY
010	2	Paper tape reader/punch	---
011	3	Reserved for card reader/punch	---
100	4	Reserved for line printer	---
101	5	Reserved for magnetic tape	---
110	6	Reserved	---
111	7	Disk (PRIMOS file system)	---

Disk (PRIMOS file system)

	Defaults	
Source	7	File System
Listing	0	None
Binary	7	File System

Default	A Register Bit	Reset (0)	Set (1)
0 0	1		SPO
0 0 0	2	LIST	EXPLIST
	3	LIST	ERRLIST
	4	NOTRACE	TRACE
1 0 0	5	32R	64R
	6		DEBASE
1	7	NOERRTTY	ERRTTY
7 1 1 1	8		INPUT
	9		SOURCE
	10		
0 0 0 0	11		LISTING
	12		
	13		
7 1 1 1	14		
	15		BINARY
	16		

	B register Bit		
0 0	1		
0 0 0	2		
	3	SAVE	DYNM
	4		
0 0 0	5		
	6		
	7		
0 0 0	8	NOBIG, 32R	BIG, DYNM, 64V
	9	NOBIG	BIG
	10	INTS	INTL
0 0 0	11		
	12	NOXREF	XREFS
	13	NOXREF	XREFL, XREFS
0 0 0	14		
	15	FP	NOFP
	16	NODCLVAR	DCLVAR, SPO

Figure 18-1. Bit-Mnemonic Correspondence

(A and B Registers)

The values of these bits are:

Binary	Octal	Device
000	0	None
001	1	User terminal
010	2	Paper tape reader/punch
011	3	Reserved for card reader/punch
100	4	Reserved for line printer
101	5	Reserved for magnetic tape
110	6	Reserved
111	7	Disk (PRIMOS file system)

File Unit Usage

Three file units may be active during a compilation:

File Type	PRIMOS file unit
Source	1
Listing	2
Object	3

If the disk is specified as the device for the listing and/or object file FTN causes these files to be opened on the disk with default names constructed as follows:

If the source file has the treename

[MFD]>UFD1>. . . .>filename

the listing file and the object file will be opened

as L←filename and B←filename respectively in the UFD currently attached to. Upon completion of the FTN command all files are closed and command returns to PRIMOS.

If the user desires the listing or binary files to have names other than the default names, and/or to be opened in UFD's other than the current one, this must be done prior to invoking the FTN command.

The PRIMOS Commands

LISTING Filename-2 opens a listing file with the specified name filename-2 (in the current UFD) on PRIMOS file unit 2. This inhibits FTN from opening a default listing file.

Note

Unless bits 11-13 of the A-register are set to '7, nothing will be written into this file.

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

For example:

```
LISTING filename
   .
   .
   .
FTN source-1 1/areg  2/breg
   .
   .
   .
FTN source-n 1/areg  2/breg
   .
   .
   .
CLOSE ALL
```

(note: system responses are not printed in this example)

The listing file, filename, 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 filename-3 (in the current UFD) on PRIMOS file unit 3. This inhibits FTN 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 be sure to set bits 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 FTN to establish non-default file, then FTN does not close these files upon completion.

After FTN 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 19

SEG COMMAND REFERENCE

INTRODUCTION

A complete list of SEG commands is given in this section in alphabetical order. The command level for each command is given, i.e. it is a command of PRIMOS, SEG, SEG's Loader, or SEG's Modification sub-processor (Modify). Along with the command level is a reference to the Section of this manual where the command is discussed in detail. Commands marked PMA are not discussed in this manual. See The PMA PROGRAMMER'S GUIDE, PDR3Ø59 for details.

Underlining shows the acceptable command abbreviations. Items in brackets ([ ]) are optional.

SEG COMMANDS

ATTACH [ufd-name] [password] [ldisk] [key] Loader (Section 11)

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.

Ø (or omitted)	search logical disk Ø
100000	search all logical disks
177777	search logical disk on which current UFD is located

key is key for attach/set information.

Ø	attach to UFD; do not set home
1	attach to UFD; set home to new current UFD
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 Loader (Section 12)

Places a symbol and reserves Ø 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 Ø is assumed.

COMMON ABS segno Loader (Section 11)

Specifies segment into which COMMON will be loaded.

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

COMMON REL segno Loader (Section 11)

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] SEG (Section 6)

Deletes saved SEG runfile with name filename. If filename is omitted the established runfile is deleted.

D/xx Loader (Section 11)

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 F/ as either D/F/xx or F/D/xx

EXECUTE [1/a-reg] [2/b-reg] [3/x-reg] Loader (Section 6)

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] Note 1. Loader (Section 12)

F/S/xx [filename] [addr psegno lsegno] Note 2. Loader (Section 12)

Forceloads all routines in a 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) or P/F/xx (or F/P/xx).

HELP SEG (Section 6)

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

IL [addr psegno lsegno] Loader (Section 12)

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 Ø 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 Ø, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

INITIALIZE [filename] Loader (Section 11)

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] Loader (Section 6,11,12)

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 Ø 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 Ø, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

LOAD synonym for VLOAD SEG
LOAD * synonym for VLOAD * SEG

LOAD filename [addr psegno lsegno] Loader (Section 6,11,12)

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 Ø 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 Ø, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

MAP filename-1 [filename-2] map-option Note 1. SEG (Section 11)
MAP * [filename-2] map-option Note 2. SEG (Section 11)

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
Ø(or omitted)	Full map
1	Extent map only
2	Extent map and base areas
3	Undefined symbols
4	Full map (identical to Ø)
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 Loader (Section 6,11)

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 is the type of loadmap to be generated. Map-options are the same as in SEG's MAP command.

MODIFY [filename] SEG (Section 11)

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 — Modify (Section 11)

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 — Loader

Allows 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
Ø	reinstate restrictions
1	relax restrictions

PATCH segno baddr taddr — Modify (PMA Manual)

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] — Loader (Section 12)

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 Ø 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 Ø, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

PSD SEG (PMA Manual)

Invokes the VPSD debugging utility.

P/xx [filename] option [psegno lsegno] Loader

Loads an object file on a page boundary. A page boundary is an address of the form 'yyØØØ 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 as F/P/xx or P/F/xx (See F/xx.)

QUIT SEG (Section 6)

Returns user to PRIMOS command level.

QUIT Loader (Section 6)

Returns user to PRIMOS command level. Does not SAVE runfile.

RESTORE [filename] SEG (PMA Manual)

Restores a SEG runfile to user memory.

filename is the SEG runfile to be restored; if omitted the established runfile is used.

RESUME [filename] SEG (Section 11)

or

RESUME [filename] SEG (Section 11)

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 Loader (Section 12)

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 Modify (Section 11)

Writes entire runfile to disk and then transfers control to the SEG command level.

RL filename [addr psegno lsegno] Loader (Section 11, 12)

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 psegno for the procedure part of the binary file. If Ø 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 Ø, SEG's default segments are used. (See S/xx, F/xx, P/xx.)

R/SYMBOL sname [segtype] segno size Loader (Section 11)

Places a symbol and reserves Ø 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 Ø, 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; Ø is assumed.

SAVE synonym for MODIFY SEG

SAVE [1/a-reg] [2/b-reg] [3/x-reg] Loader (Section 6)

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 Note 1. PRIMOS (Section 7)
SEG Note 2. PRIMOS (Section 6)
SEG filename 1/1 Note 3. PRIMOS (PMA Manual)
SEG 1/1 Note 4. PRIMOS (PMA Manual)

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 VPSD 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] SEG (Section 12)

Converts portions of the SEG runfile corresponding to segments below '4001 into R-mode-like runfiles.

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 SEG (Section 12)

Creates an R-mode-like runfile for specified segment number.

filename is the name of the SEG runfile from which an R-mode runfile is to be split. If omitted, the established runfile is used.

segno is the absolute octal number of the segment for which the R-mode runfile is to be created.

SEG responds to the SINGLE command by asking for a two-character ID as:

TWO CHARACTER FILE ID:

The R-mode runfile is created with a filename composed of the two-character ID followed by the (octal) segmemt number specified.

SK ssize	Note 1. Modify	(Section 11)
SK segno addr	Note 2. Modify	(Section 11)
SK ssize Ø segno	Note 3. Modify	(Section 12)
SK ssegno addr segno	Note 4. Modify	(Section 12)

Notes

1. Specifies stack size

ssize is minimum required stack size in octal words; if Ø 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 Ø 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	Note 1. Loader	(Section 12)
SPLIT addr	Note 2. Loader	(Section 12)
SPLIT addr ssegno saddr esegno	Note 3. Loader	(Section 12)

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 R-mode 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 ssegno

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 Loader (Section 11)

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 Modify (Section 11)

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 Loader (Section 11)

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 segno.

S/xx [filename] addr psegno lsegno Loader (Section 12)

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 Ø 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] SEG (Section 11)

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] SEG (Section 6)

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] SEG (Section 11)

Accesses 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 Modify (PMA Manual)

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 Loader (Section 12)

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 areas

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

SECTION 20

LIBRARIES REFERENCE

FORTRAN FUNCTION LIBRARY

The following functions are available to perform mathematical and logical operations. These functions are part of the FTNLIB library file for the R-identity and the PFTNLB and IFTNLB library files for the V-identity. The data mode(s) expected in the argument list and the data mode of the value returned are shown for each function in the list. The following abbreviations are used:

CP	Complex number
DP	Double-precision floating-point number
I	Integer (short or long)
J	Integer (long)
SP	Single-precision floating-point number

Additional detail on the functions themselves (rather than their operations) will be found in REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106.

Mixing Long and Short Integers

Short integers occupy one word of memory, long integers two words. When long integers are converted to short integers, the 16 low order bits of the long integer are stored in the short integer. When a short integer is converted to a long integer, the low order word is set equal to the short integer; the high order word is sign-extended (padded with 0's or 1's according to the sign of the short integer, + or -). If it is necessary, in a program, to convert between integer modes, it is strongly recommended that this be done with the intrinsic functions: INTL, INTS. (In the following, it is assumed that all variable names beginning with I have been declared to be short integers and all variable names beginning with J as long integers.)

To convert between integer modes, use:

```
J = INTL (I)

I = INTS (J)
```

If a long-(or short) integer is assigned the value of a short (or long) integer, mode conversion will also occur. This is not considered to be good programming practice and is discouraged. (See Assignment Statements in Section 16.)

In functions which accept mixtures of short and long integers in the argument list, the short integers will be internally converted to long integers (with sign-extension) and the value determined. The value will be calculated as a long integer. For these functions it is recommended that the left-hand side of the assignment statement be a long integer. Conversion to a short integer should be explicit, not implicit.

```
JX = AND (JA, JB, IC)
```

is less desirable than

```
JX = AND (JA, JB, INTL (IC))
```

and

```
IY = AND (JA, JB, IC)
```

is less desirable than

```
IY = INTS (AND (JA, JB, INTL (IC)))
```

In general, the logical functions AND, OR, and XOR and the minimum/maximum functions will return a long integer if any of the arguments are long integers. The NOT function returns an integer of the same mode as its argument. The shifting and truncating functions LS, LT, RS, RT, and SHFT return an integer of the same mode as their first argument, that is, the integer on which shifting and/or truncation is to take place.

FORTRAN Functions

ABS — Calculates the absolute value of the argument.
SP = ABS (SP)

AIMAG — Converts the imaginary part of a complex number to a single-precision floating-point number.
SP = AIMAG (CP)

AINT — Truncates a single-precision floating-point number to a single-precision floating-point number whose value is integral.
SP = AINT (SP)

ALOG — Computes the natural logarithm (base e) of the argument. If the argument is not positive, the error LG is generated.
SP = ALOG (SP)

ALOG10 — Computes the base-10 logarithm of the argument. If the argument is not positive, the error LG is generated.
SP = ALOG10 (SP)

AMAX0 Finds the maximum value in a variable list of integers. The list may be a mixture of long and short integers.
SP = AMAX0 (I1,I2,...,In) ~~(In the V-identity, 2<n<4)~~

AMAX1 Finds the maximum value in a variable list of single-precision floating-point numbers.
SP = AMAX1 (SP1,SP2,...,SPn) ~~(In the V-identity, 2<n<4)~~

AMIN0 Finds the minimum value in a variable list of integers. The list may be a mixture of long and short integers.
SP = AMIN0 (I1,I2,...,In) ~~(In the V-identity, 2<n<4)~~

AMIN1 Finds the minimum value in a variable list of single-precision floating-point numbers.
SP = AMIN1 (SP1,SP2,...,SPn) ~~(In the V-identity, 2<n<4)~~

AMOD Computes the remainder when one single-precision floating-point number (SP1) is divided by another (SP2).
SP = AMOD (SP1,SP2)

AND Performs a logical AND operation, bit by bit, on a variable list of integers, long and/or short.
I = AND (I1,I2,...,In)

ATAN Calculates the principal value, in radians, of the arctangent of the argument.
SP = ATAN (SP)

ATAN2 Calculates the principal value, in radians, of the arctangent of one single-precision floating-point number (SP1) divided by another (SP2). If both arguments are zero, the error message AT is generated.
SP = ATAN2 (SP1,SP2)

CABS Computes the absolute value of a complex number, returning a single-precision floating-point number as the result.
SP = CABS (CP)

CCOS Computes the cosine of a complex number.
CP = CCOS (CP)

CEXP Calculates the exponential of a complex number.
CP = CEXP (CP)

CLOG Calculates the natural logarithm (base e) of the argument.
CP = CLOG (CP)

CMPLX Converts two single-precision floating-point numbers into a complex number. The first argument becomes the real part of the complex number; the second argument becomes the imaginary part.
CP = CMPLX (SP1,SP2)

CONJG Computes the conjugate of a complex number.
CP = CONJG (CP)

COS Computes the cosine of a single-precision floating-point number.
SP = COS (SP)

CSIN Computes the sine of complex number.
CP = CSIN (CP)

CSQRT Calculates the square root of a complex number.
CP = CSQRT (CP)

DABS Computes the absolute value of a double-precision floating-point number.
DP = DABS (DP)

DATAN Computes, in radians, the principal value of the arctangent of the argument.
DP = DATAN (DP)

DATAN2 Calculates the principal value, in radians, of the arctangent of one double-precision floating-point (DP1) divided by another (DP2). If both arguments are zero, the error message DT is generated.
DP = DATAN2 (DP1,DP2)

DBLE Converts a single-precision floating-point number to a double-precision floating-point number.
DP = DBLE (SP)

DCOS Computes the cosine of a double-precision floating-point number.
DP = DCOS (DP)

DEXP Computes the exponential of a double-precision floating-point number.
DP = DEXP (DP)

DIM Computes the positive difference between two single-precision floating-point numbers.
SP = DIM (SP1,SP2)

DINT Truncates the fractional part of a double-precision floating-point number.
DP = DINT (DP)

DLOG Computes the natural logarithm (base e) of a double-precision floating-point number. If the argument is not positive, the error message DL is generated.
DP = DLOG (DP)

DLOG2 Computes the base-2 logarithm of a double-precision floating-point number. If the argument is not positive, the error message DL is generated.
DP = DLOG2 (DP)

DLOG10 Computes the base-10 logarithm of a double-precision floating-point number. If the argument is not positive, the error message DL is generated.
DP = DLOG10 (DP)

DMAX1 Finds the maximum value among a variable list of double-precision floating point numbers.
DP = DMAX1 (DP1,DP2,...,DPn) ~~(In V-identity, 2≤n≤4)~~

DMIN1 Finds the minimum value among a variable list of double-precision floating-point numbers.
DP = DMIN1 (DP1,DP2,...,DPn) ~~(In V-identity, 2≤n≤4)~~

DMOD Computes the remainder when one double-precision floating-point number (DP1) is divided by another (DP2). If DP2 is zero, the error message DZ is printed.
DP = DMOD (DP1,DP2)

DSIGN Combines the magnitude of one double-precision floating-point number (DP1) with the sign of a second (DP2).
DP = DSIGN (DP1,DP2)

DSIN Computes the sign of a double-precision floating-point number.
DP = DSIN (DP)

DSQRT Computes the square root of a double-precision floating-point number. If the argument is negative, the error message SQ is generated.
DP = DSQRT (DP)

EXP Computes the exponential of a single-precision floating-point number. If there is an exponent underflow or overflow, the error message EX is generated. SP = EXP (SP)

FLOAT Converts an integer to a single-precision floating-point number. The function will accept either a short or a long integer as the argument
SP = FLOAT (I)

IABS Computes the absolute value of an integer. The argument may be either a long or short integer.
I = IABS (I)

IDIM Computes the positive difference between two integers. The function will accept any mixture of short and long integers.
I = IDIM (I1,I2)

IDINT Converts a double-precision floating-point to an integer.
I = IDINT (DP)

IFIX
INT — Converts a single-precision floating-point number to an integer. Both functions are included in the library to ease conversion from other systems.

```
I = IFIX (SP)
I = INT (SP)
```

INTL — Converts its argument to a long integer.

```
J = INTL (I)
```

INTS — Converts its argument to a short integer.

```
I = INTS (J)
```

IRND — Invokes the random number generator

```
I2 = IRND (I1)
```

I1	Operation	I2
>0	Initializes the random number generator	I2 = I1
=0	Generates a random number	0≤I2≤32767
<0	Initializes the random number generator and returns the first random number	0≤I2≤32767

ISIGN — Combines the magnitude of one integer (I1) with the sign of a second (I2).

```
I = ISIGN (I1,I2)
```

LOC — Generates an integer value representing the memory address where the argument of LOC is located. The argument may be a constant, variable or array name, or a subscripted array element.

```
I = LOC ( { constant
          { variable name
          { array name
          { array element )
```

Note

In the 64V mode, LOC may be passed as an argument in functions or subroutines, e.g., I = AND(LOC(A),LOC(B)). In this mode, LOC returns a two-word value: the first word represents the segment number; the second is the word number in the segment.

LS — Shifts an integer variable left by a specified number of bits; vacated bits are filled with zeroes.

```
I2 = LS (I1, IP)
```

where IP is the number of bits to be shifted to the left. If IP<0, this is equivalent to the RS function with IP = -IP. If IP = 0, no change is made to the integer.

LT — Preserves a specified number of left-most bits and sets the rest to zero. (left truncation). Saves the first IP from the left and sets the rest of the bits to zero. If IP≤0, the entire integer is set to zero.
I2 = LT (I1,IP)

MAX0 — Finds the maximum value among a variable list of integers. (see AMAX0) ~~(In V-identity, 2≤n≤4)~~
I = MAX0 (I1,I2,...,In)

MAX1 — Finds the maximum value among a variable list of single-precision floating-point numbers and converts it to an integer. ~~(In V-identity, 2≤n≤4)~~
I = MAX1 (SP1,SP2,...,SPn)

MIN0 — Finds the minimum value among a variable list of integers. (see AMIN0). ~~(In V-identity, 2≤n≤4)~~
I = MIN0 (I1,I2,...,In)

MIN1 — Finds the minimum value among a variable list of single-precision floating-point numbers and converts it to an integer (see AMIN1) ~~(In V-identity, 2≤n≤4)~~
I = MIN1 (SP1,SP2,...,SPn)

MOD — Computes the remainder when one integer (I1) is divided by another (I2).
I = MOD (I1,I2)

NOT — Performs a logical NOT operation 1's complement) on its argument.
I = NOT (I)

OR — Performs a logical (inclusive) OR operation on two integers.
I = OR (I1,I2)

REAL — Converts the real part of a complex number to a single-precision floating-point number.
SP = REAL (CP)

RND — Invokes the random number generator.
SP = RND (I)

I	Operation	SP
>0	Initializes the random number generator	SP = FLOAT (I)
=0	Generates a random number	0.0≤SP≤1.0
<0	Initializes the random number generator and returns the first random number	0.0≤SP≤1.0

RS — Shifts an integer variable right by a specified number of bits; vacated bits are filled with zeroes.
I2 = RS (I1,IP)
where IP is the number of bits to be shifted to the right. If IP<0, this is equivalent to the LS function with IP = -IP. If IP = 0, no change is made to the integer.

RT — Preserves a specified number of right-most bits and sets the rest to zero (right truncation). Saves the first IP bits from the right and sets the rest of the bits to zero. If IP≤0, the entire integer is set to zero.
I2 = RT (I1,IP)

SHFT — Performs logical shift operations on integer variables.

1. IS = SHFT (I)
   In this form, the variable is unchanged and the value is the variable itself; this form has no real use.
2. IS = SHFT (I,IP1)
   performs a shift operation on the variable. If IP1>0, the shift is to the right; if IP1<0, the shift is to the left; if IP1 = 0, no shift occurs. This form is equivalent to the RS and LS functions.

Operation	Function	Equivalent SHFT function
Right shift	RS (I,IP)	SHFT (I,IP)
Left shift	LS (I,IP)	SHFT (I,-IP)
Right truncate	RT (I,IP)	SHFT (I,IP-16,16-IP)
Left truncate	LT (I,IP)	SHFT (I,16-IP,IP-16)

3. IS = SHFT (I,IP1, IP2)
   performs two shift operations, first by IP1 (setting zeroes in vacated bits), then by IP2 (setting zeroes in vacated bits). The sign of IP1 and IP2 determine the direction of the shift while their magnitude determine the number of bits to be shifted. As seen above, the RT and LT functions are equivalent to special forms of SHFT with three arguments.

SIGN — Combines the magnitude of one single-precision floating-point number (SP1) with the sign of a second (SP2).
SP = SIGN (SP1,SP2)

SIN — Computes the sine of a single-precision floating-point number.
SP = SIN (SP)

SNGL — Converts a double-precision floating-point number to a single-precision floating-point number.
SP = SNGL (DP)

SQRT — Computes the square root of a single-precision floating-point number.
SP = SQRT (SP)

TANH	Computes the hyperbolic tangent of a single-precision floating-point number. SP = TANH (SP)
XOR	Performs a logical exclusive OR on a variable list of integers. I = XOR (I1,I2,...,In)

FORTRAN MATRIX (MATH) LIBRARY

The following subroutines are available to the user for matrix manipulation, solution of sets of linear equations and generation of combinations and permutations. In the subroutines whenever the mode of an argument is explicitly specified as integer it is taken to be a short integer (indexes, error flags, etc.). However, the mode of the matrix elements for integer matrices may be either long or short integers. This library exists only in the R-mode version; the library file name is MATHLIB.

Matrix Operations Subroutines

COMB

CALL COMB (icomb,n,nr,iw1,iw2,iw3,last[,restrt])

COMB computes the next combination of nr out of n elements with a single interchange each time it is called. The first call to COMB returns the combination 1,2,3,...,nr. This subroutine is self-initializing and proceeds through all n!/(nr!*(n-nr)!) combinations. At the last combination, it returns a value of last = 1 and resets itself. The COMB subroutine may be re-initialized by the user by passing a new value of n and/or nr or by passing the restrt parameter with a value of 1. (The restrt parameter is optional; if re-initialization is not desired either omit this parameter from the calling sequence or set it to a value of 0.)

Argument	Mode	Subscript(s)	Dimension(s)	Comments
icomb	Integer	1	nr	return
n	Integer			pass
nr	Integer			pass
iw1	Integer	1	n	work
iw2	Integer	1	n	work
iw3	Integer	1	n	work
last	Integer			return
restrt	Integer			pass (optional)

The calling program should not attempt to modify icomb, iw1, iw2, or iw3. For further details see:

"Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," Gideon Ehrlich, Journal of the ACM, 20, No3 (July 1973) pp. 5000-5113.

Note

COMB is not loopless

LINEQ, CLINEQ, DLINEQ

CALL LINEQ (xvect, yvect, cmat, work, n, npl, ierr)

(This is the form for single precision numbers; for complex and double-precision numbers, use CLINEQ and DLINEQ respectively.) Solves the set of n linear equations in n unknowns represented by

(cmat) (xvect) = (yvect)

where CMAT is the nxn square matrix of coefficients, yvect is the nx1 column vector of constants, and xvect is the nx1 column vector of unknowns in which the solution is stored. The user is required to provide as a work area, a nplxnpl matrix work (npl = n+1). The integer error flag ierr returns one of three possible values.

ierr	
0	solution found satisfactorily
1	Coefficient matrix singular
2	npl ≠ n+1

If ierr ≠ 0 no modifications are made to xvect.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
xvect	*	1	n	returned
yvect	*	1	n	passed
cmat	*	2	n,n	passed
n	Integer			passed
work	*	2	npl,npl	work
npl	Integer			passed (=n+1)
ierr	Integer			returned

* all of the same mode which determine the subroutine used.

MADD, CMADD, DMADD, IMADD

CALL MADD (mats, mat1, mat2, n, m)

(This is the form for single precision number, for integer, complex, and double-precision numbers use IMADD, CMADD, and DMADD respectively). MADD adds the nxm matrix mat2 to the nxm matrix mat1 and returns the sum in a nxm matrix mats. In component form:

mats (i,j) = mat1 (i,j) + mat2 (i,j)

as i goes from 1 to n and j goes from 1 to m.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
mats	*	2	n,m	returned
mat1	*	2	n,m	passed
mat2	*	2	n,m	passed
n	Integer			passed
m	Integer			passed

* all of the same mode which determines the subroutine used.

MADJ, CMADJ, DMADJ, IMADJ

CALL MADJ (mato, mati, n, iwl, iw2, iw3, iw4, ierr)

(This is the form for single precision numbers. For integer, complex, or double-precision numbers use IMADJ, CMADJ or DMADJ respectively).

The subroutine calculates the adjoint of the nxn matrix mati and stores it in the nxn matrix mato. Each element of the output matrix is the signed cofactor of the corresponding element of the input matrix. The error flag, ierr, may have one of two values.

ierr

0	adjoint successfully constructed
1	n<2 - no adjoint may be constructed

Note

mato and mati must be distinct.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,n	returned
mati	*	2	n,n	passed
iwl	*	1	n	work
iw2	*	1	n	work
iw3	*	1	n	work
iw4	*	1	n	work
ierr	Integer			returned

* all of the same mode which determines the subroutine used.

MCOF, CMCOF, DMCOF, IMCOF

CALL MCOF (cof, mat, n, iwl, iw2, iw3, iw4, i, j, ierr)

(This is the form for single precision numbers. For integers, complex, or double-precision numbers use IMCOF, CMCOF, or DMCOF respectively). Calculates the signed cofactor of the element mat (i,j) of the nxn matrix mat and stores this value in cof. If i = 0 and j = 0, the determinant of mat is calculated. The integer error flag ierr has two possible values.

ierr

0	cofactor calculated successfully
1	no cofactor calculated for any of the following reasons:

1. n<2 - no cofactor possible
2. i = j = n = 0 - no determinant
3. i = 0 and j ≠ 0 or i ≠ 0 and j = 0 - subscript error
4. i>n and/or j>n - subscript error

Argument	Mode	Subscript(s)	Dimension(s)	Comments
cof	*			returned
mat	*	2	n,n	passed
n	Integer			passed
iwl	*	1	n	work
iw2	*	1	n	work
iw3	*	1	n	work
iw4	*	1	n	work
i	Integer			passed
j	Integer			passed
ierr	Integer			returned

* all of the same mode which determines the subroutine used.

MCON, CMCON, DMCON, IMCON

CALL MCON (mat, n, m, con)

(This is the form for single-precision numbers. For integer, complex, or double-precision numbers use IMCON, CMCON, or DMCON respectively). This subroutine sets every element of the nxm matrix mat equal to a constant con.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
mat	*	2	n,m	returned
n	Integer			passed
m	Integer			passed
con	*			passed

* all of the same mode which determine which subroutine is used.

MDET, CMDET, DMDET, IMDET

CALL MDET (det, mat, n, iwl, iw2, iw3, iw4, ierr)

(This is the form for single-precision numbers. For integer, complex, or double-precision numbers use IMDET, CMDET, or DMDET respectively). This subroutine calculates the determinant of the nxn matrix mat and stores it in det. The integer error flag ierr may have one or two values.

ierr

0	determinant formed successfully
1	n = 0 - no determinant possible

Argument	Mode	Subscript(s)	Dimension(s)	Comments
det	*			returned
mat	*	2	n,n	passed
n	Integer			passed
iwl	*	1	n	work
iw2	*	1	n	work
iw3	*	1	n	work
iw4	*	1	n	work
ierr	Integer			returned

* all of the same mode which determine the subroutine used.

MIDN, CMIDN, DMIDN, IMIDN

CALL MIDN (mat, n)

(This is the form for single-precision numbers. For integer, complex, or double-precision numbers use IMIDN, CMIDN, or DMIDN respectively).

This subroutine sets the nxn matrix mat equal to the nxn identity matrix. That is,

mat (i,j) = δij

where $\delta ij = \begin{matrix} 0 & i \neq j \\ 1 & i = j \end{matrix}$

Argument	Mode	Subscript(s)	Dimension(s)	Comments
mat	*	2	n,n	returned
n	Integer			passed

* the mode of this argument determines which subroutine is used and the representation of 1 in the matrix.

MINV, CMINV, DMINV

CALL MINV (mato, mati, n, work, npl, npn, ierr)

(This is the form for single-precision numbers. For complex or double-precision numbers use the subroutines CMINV or DMINV respectively. There is no integer form of this subroutine as there is no guarantee that the inverse of an integer matrix will be an integer matrix). Calculates the inverse of the nxn matrix mati and stores it in mato if successful. (The inverse of mati is mato if and only if

mati*mato = mato*mati = I

where * denotes matrix multiplication and I is the nxn identity matrix). The user must supply a npl x npn scratch matrix work, where npl = n+1 and npn = n+n. The integer error flag ierr will return one of the following values.

ierr

0	matrix inverted - inverted matrix stored in mato.
1	matrix is singular - no inversion possible. mato is filled with zeroes.
2	npl ≠ n+1 and/or npn ≠ n+n - return from subroutines with no calculations performed.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,n	returned
mati	*	2	n,n	passed
n	Integer			passed
work	*	2	npl,npn	work
npl	Integer			passed
npn	Integer			passed
ierr	Integer			returned

MMLT, CMMLT, DMMLT, IMMLT

```
CALL MMLT (matp, matl, matr, nl, n2, n3)
```

(This is the form for single-precision numbers. For integers, complex, or double-precision numbers use IMMLT, CMMLT, or DMMLT respectively. This subroutine multiplies the nlxn2 matrix matl (on the left) by the n2xn3 matrix matr (on the right) and stores the resulting nlxn3 product matrix in matp.

Note

matp must be distinct from matl and matr, although matl and matr may be the same. For example:

```
CALL MMLT (A, B, C, Nl, N2, N3)
CALL MMLT (A, B, B, N, N, N)        Legal

CALL MMLT (A, A, A, N, N, N)
CALL MMLT (A, A, B, N, N, N)        Illegal
CALL MMLT (A, B, A, N, N, N)
```

Argument	Mode	Subscript(s)	Dimension(s)	Comments
matp	*	2	nl,n3	returned
matl	*	2	nl,n2	passed
matr	*	2	n2,n3	passed
nl	Integer			passed
n2	Integer			passed
n3	Integer			passed

* are of the same mode which determines which subroutine is used.

MSCL, CMSCL, DMSCL, IMSCL

CALL MSCL (mato, mati, n, m, scon)

(This is the form for single-precision numbers. For integers, complex, or double-precision numbers use IMSCL, CMSCL, or DMSCL.)

This subroutine multiplies the nxm matrix mati by scalar constant scon and stores the resulting nxm matrix in mato. By components scalar multiplication is understood to be:

mato (i,j) = scon*mati (i,j)

for i from 1 to n, j from 1 to m.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,m	returned
mati	*	2	n,m	passed
n	Integer			passed
m	Integer			passed
scon	*			passed

* all of the same mode which determines which subroutine is used.

MSUB, CMSUB, DMSUB, IMSUB

CALL MSUB (matd, matl, mat2, n, m)

(This is the form for single-precision numbers. For integers, complex or double-precision numbers use IMSUB, CMSUB, or DMSUB respectively). Subtracts the nxm matrix mat2 from the nxm matrix matl and stores the difference in the nxm matrix matd.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
matd	*	2	n,m	returned
matl	*	2	n,m	passed
mat2	*	2	n,m	passed
n	Integer			passed
m	Integer			passed

* all of the same mode which determine the subroutine to be used.

MTRN, CMTRN, DMTRN, IMTRN

CALL MTRN (mato, mati, n)

(This is the form for single-precision numbers. For integers, complex, or double-precision numbers use IMTRN, CMTRN, or DMTRN respectively). Calculates the transpose of the nxn matrix mati and stores it in the nxn matrix

mato. The relationship between mati and mato is:

mato (i,j) = mati (j, i)

for i, j = 1 to n. mato and mati must be distinct.

Argument	Mode	Subscript(s)	Dimension(s)	Comment
mato	*	2	n,n	returned
mati	*	2	n,n	passed
n	Integer			passed

* all of the same mode which determines the subroutine used.

PERM

CALL PERM (iperm, n, iw1, iw2, iw3, last [, restrt])

PERM computes the next permutation of n elements with a single interchange of adjacent elements each time it is called. The first call to PERM returns the permutation 1, 2, 3, ..., n. This subroutine is self-initializing and proceeds through all n! permutations. At the last permutation it returns a value of last = 1 and resets itself. The PERM subroutine may be re-initialized by the user by passing a new value of n or by passing the restrt parameter with a value of 1. (The restrt parameter is optional, if re-initialization is not desired either omit this parameter from the calling sequence or set it to a value of 0. The calling program should not attempt to modify iperm, iw1, iw2, or iw3.

Argument	Mode	Subscript(s)	Dimension(s)	Comments
iperm	Integer	1	n	returned
n	Integer			pass
iw1	Integer	1	n	work
iw2	Integer	1	n	work
iw3	Integer	1	n	work
last	Integer			return
restrt	Integer			passed (optional)

For further details see:

"Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," Gideon Ehrlich, Journal of the ACM, 20, No3 (July 1973) pp 5000-5113.

SORT AND SEARCH LIBRARY

The subroutines listed here are contained in the library MSORTS in UFD=LIB. This is an R-mode library. There is, at present, no V-mode version. A complete discussion of these subroutines will be found in REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106.

See Knuth, Donald The Art of Computer Programming vol. 3 for complete discussion of these types of sorts.

Characteristics of the Sorts

Sort	Approximate relative running time		Comments
	Average	Maximum	
BUBBLE	N^2		only good for very small N
HEAP	(23N) ln N	(26N) ln N	inefficient for N<2000
INSERT	N^2	-	small N; very good on nearly ordered tables
QUICK	(12N) ln N	N^2	fastest but very slow on nearly ordered tables
SHELL	$N^{1.25}$	$N^{1.5}$	good for N<2000

Where:

N is the number of entries in the table (nentry).

These routines all sort the table in increasing order with the key treated as a single, signed multi-word integer.

RADXEX, however, treats the key as a single, unsigned multi-word (or partial word) integer. For example:

If the keys were 5, -1, 10, -3,
RADXEX would sort them to: 5, 10, -3, -1
The other routines would sort them to: -3, -1, 5, 10

Parameters Common to More Than One Subroutine

table — Pointer to first word of table of entries. Example: the table is an array TABLE (I,J), then ptable=LOC (table). (type: INTEGER)

nentry — Number of entries in the table (e.g., items to be sorted or searched). (type: INTEGER)

nwords — Number of words/entry. (type: INTEGER)

fword — Starting word of the key field in the entry. 0<fword<nwords (type: INTEGER)

nkwrds — Number of words in the key filed. 0<nkwrds<nwords. fword+nkwrds-1<nwords (the key field must be contained within the entry). (type: INTEGER)

tarray — A temporary one-dimensional array used as a work area; size varies with sort used.

npass — Returned pass counter (type: INTEGER)

altbp — An optional alternate return if an error is caused by a bad parameter (type: address constant). If altbp is not specified, then an error causes a normal return with npass=0.

General Requirements for Using In-memory Sorts

1. all entires must be of equal length
2. key words must be contiguous (no secondary keys)

Sorts

BUBBLE - interchange sort

CALL BUBBLE (ptable,nentry,nwords,fword,nkwrds,tarray,npass,altbp,incr)

Where tarray has dimension nkwrds

Where incr is used to sort non-adjacent entries in the tables. Default is INCR=1 (adjacent) (type: INTEGER)

HEAP - heapsort

CALL HEAP (ptable,nentry,nwords,fword,nkwrds,tarray,npass,altbp)

Where tarray has dimension nwords

INSERT - straight insertion sort

CALL INSERT (ptable,nentry,nwords,fword,nkwrds,npass,altbp,incr)

Where incr is used to sort non-adjacent entries in the table. Default is incr=1 (adjacent). (type: INTEGER)

QUICK - partition exchange sort

CALL QUICK (ptable,nentry,nwords,fword,nkwrds,tarray,npass,altbp)

Where tarray has dimension nwords

RADXEX - radix exchange sort

CALL RADXEX (ptable,nentry,nwords,fword,fbit,nbit,tarray,npass,altbp)

Where fbit is the first bit within FWORD of the key

nbit is the number of bits in the key

Note

fword+(nbit+fbit-2)/16<nwords

Where tarray has dimension 2*nbit

SHELL - diminishing increment sort

CALL SHELL (ptable,nentry,nwords,fword,nkwrds,npass,altbp)

Search

BNSRCH - search/maintain ordered table

CALL BNSRCH (ptable,nentry,nwords,fword,nkwrds,skey,fentry,index,
opflag,altnf,altbp)

skey - a search key array of dimension nkwrds

fentry - array of dimension nwords into which the found entry is read (see below under opflag=3 for special use)

index - entry number of the found entry

opflag - operation flag

0 locate

1 locate and delete

2 locate or insert

3 locate and update

altnf - alternate return if entry is not found

Simple binary searching (opflag=0) tests each entry's key field for a match with skey. If the entry is found, it is returned in fentry and the entry number is put into index. If the entry is not found, the not found alternate return (altnf) is taken. If altnf is not specified, the normal return is taken with index=0.

The operation for opflag=1 is the same as opflag=0 except that if the entry is found, it is deleted from the table as well as returned in fentry. In this case, index specifies where the entry was.

The operation for opflag=2 is the same as opflag=0 if the entry is found. If, however, the entry is not found, the contents of fentry will be inserted into the table and index will indicate the position of the new element. Also altnf will be taken.

The operation for opflag=3 is the same as opflag=0 if the entry is not found. If the entry is found, the contents of fentry and the found entry are interchanged, thus updating the table and returning the old entry.

APPLICATIONS LIBRARY

The applications library provides programmers with easy-to-use functions and service routines falling between very high-level constructs and very low-level systems routines. The applications library is located in UFD=LIB in the files APPLIB (R-mode programs) and VAPPLB (V-mode programs). All routines in VAPPLB are pure procedure and may be loaded into the shared portion of a shared procedure. The applications library should be loaded before loading the FORTRAN library.

Programs using applications library routines should use the $INSERT file SYSCOM>A$KEYS which defines the keys used in these routines.

The applications routines may be used as functions or as subroutine calls as desired. The function usage gives additional information. The type of value of the function (LOGICAL, INTEGER, etc.) is specified for each function.

A detailed description of this library will be found in REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106.

CLOS$A Closes a file open on funit

<logical> = CLOSE$A (funit)

or

CALL CLOSE$A (funit)

funit - PRIMOS file unit number (type: INTEGER*2)

If function form is used:

.TRUE. operation is successful

.FALSE. operation not successful

CNVA$A Converts ASCII digit string to numerical value for octal, decimal, hexadecimal numbers.

<logical> = CNVA$A(numkey,name,namlen,value)

or

CALL CNVA$A(numkey,name,namlen,value)

numkey - number base (type: INTEGER*2)

A$DEC decimal

A$OCT octal

A$HEX hexadecimal

name - ASCII number string variable

namlen - length of NAME in characters (type: INTEGER*2)

value - numerical value returned (type: INTEGER*4)

If function form is used:

.TRUE. successful conversion

.FALSE. unsuccessful conversion; VALUE=0

Note

Overflow for octal or hexadecimal is ignored (.TRUE.) but considered a failure (.FALSE.) for decimal

---

CTIM$A Returns CPU time since login in centiseconds

<real*8> = CTIM$A(cputim)

or

CALL CTIM$A(cputim)

cputim - returned value of CPU time (type: INTEGER*4)

If function form is used:

value is CPU time since login seconds (type: REAL*4 or REAL*8)

DATE$A Returns the current date

<real*8> = DATE$A*date)

or

CALL DATE$A(date)

date - returned value of the date as DAY, MON DD 19YR (date must be at least 16 characters long)

If function form is used:

value is date as MM/DD/YY (type: REAL*8)

Note

Valid from January 1, 1977 to December 31, 1986.

DELE$A Deletes a file with specified name

<logical> = DELE$A(name, namlen)

or

CALL DELE$A(name,namlen)

name - filename (or treename) of file to be deleted

namlen - length of name in characters (type: INTEGER*2)

If function form is used:

.TRUE. deletion successful

.FALSE. deletion unsuccessful (file not found, etc.)

DOFY$A Returns current day number in year

<real*8> = DOFY$A(dofy)

or

CALL DOFY$A(dofy)

dofy - value returned as DDD . (dofy must be at least 4 characters)

If function form is used:

value date as YR.DDD, suitable for FORTRAN format F6.3 (type: REAL*4 or REAL*8)

Note

Valid from January 1, 1977 to December 31, 1986.

DTIM$A Returns disk time since login in centiseconds

<real*8> = DTIM$A(dsktim)

or

CALL DTIM$A(dsktim)

dsktim - disk time in centiseconds (type: INTEGER*4)

If function form is used:

value is disk time since login in seconds (type: REAL*4 or REAL*8)

EDAT$A Returns date in European/military format

<real*8> = EDAT$A(edate)

or

CALL EDAT$A(edate)

edate - returned date in form DAY, DD MON 19YR (EDATE must be at least 16 characters long)

If function form is used:

value is DD/MM/YR (type: REAL*8)

Note

Valid from 1 January 1977 to 31 December 1986.

ENCD$A Encodes value in FORTRAN floating print format

<logical> = ENCD$A (array,width,dec,value)

or

CALL ENCD$A (array,width,dec,value)

array - array which will receive the encoded value (must be long enough to receive this)

width - field width as in FORMAT Fw.d (must be even; maximum = 16) (type: INTEGER*2)

dec - number of places to right of decimal point; d in FORMAT Fw.d (type: INTEGER*2)

value - value to be encoded (type: REAL*8)

If function form is used:

.TRUE. successful encoding

.FALSE. encoding failed (ARRAY filled with asterisks)

Note

This routine will attempt to encode value in the supplied Fw.d format if it will fit. If not, the dec argument is decremented (moving the decimal point to the right) until it will fit. If dec reaches 0, or is originally supplied as 0, value will be encoded in Iw format if the number

will fit into a 32-bit integer. If not, and if the field is wide enough (width > 7), the VALUE will be encoded in E format. If the field is not wide enough, it will be filled with asterisks.

Note that the largest value of width will be 16. If it is larger than 16, only the first 16 characters of array will be used.

EXST$A Checks for existence of specified file

<logical> = EXST$A (name,namlen)

name - filename (or treename) of file to be searched for

namlen - length of NAME in characters (type: INTEGER*2)

.TRUE. file exists

.FALSE. file does not exist or error encountered

FILL$A Fills buffer with specified character

<integer> = FILL$A(name,namlen,char)

or

CALL FILL$A(name,namlen,char)

name - buffer to be filled

namlen - length of name in characters (type: INTEGER*2)

char - fill character in FORTRAN A1 FORMAT (type: INTEGER(2)

If function form is used:

value is INTEGER (*2 or *4 according to compilation option)

GCHR$A Accesses character in specified array position

<integer> = GCHR$A (farray,fchar)

or

CALL GCHR$A (farray,fchar)

farray - a packed array from which character is to be obtained

fchar - character position in farray (type: INTEGER*2)

If function form is used:

value is the accessed character in FORTRAN A1 FORMAT; character is in leftmost byte, right padded with blanks.

GEND$A Positions file pointer on funit to End-of-File position

<logical> = GEND$A (funit)

or

CALL GEND$A (funit)

funit - PRIMOS file unit number (type: INTEGER2)

If function form is used:

.TRUE. pointer positioned

.FALSE. pointer not positioned

MCHR$A Replaces a character in one array with a character from another

<integer> = MCHR$A(tarray,tchar,farray,fchar)

or

CALL MCHR$A(tarray,tchar,farray,fchar)

tarray - receiving packed array

tchar - character position in tarray (type: INTEGER*2)

farray - source packed array

fchar - character position in farray (type: INTEGER*2)

If function form is used:

value is accessed character in FORTRAN A1 FORMAT; character is leftmost byte, right padded with blanks.

NLEN$A Returns operational length of buffer

<integer*2> = NLEN$A(name,namlen)

or

CALL NLEN$A(name,namlen)

name - buffer whose length is to be tested

namlen - length of name in characters (type: INTEGER*2)

If function form is used:

value is operation length (not including trailing blanks) of buffer.

OPEN$A Opens file on specified funit

<logical> = OPEN$A(opnkey+typkey,name,namlen,funit)

or

CALL OPEN$A(opnkey+typkey,name,namlen,funit)

opnkey - action to be taken (type: INTEGER*2)

A$READ open for reading

A$WRIT open for writing

A$RDWR open for reading and writing

typkey - type of file (type: INTEGER*2)

A$SAMF SAM file

A$DAMF DAM file

name - filename (or treename) of file to be opened

namlen - length of name in characters (type: INTEGER*2)

funit - PRIMOS file unit number on which to open file

In function form:

.TRUE. opened successfully

.FALSE. not opened

OPNP$A Gets a filename from user terminal and opens it on FUNIT

<logical> = OPNP$A(msg,msglen,opnkey+typkey,name,namlen,funit)

or

CALL OPNP$A(msg,msglen,opnkey+typkey,name,namlen,funit)

msg - user-supplied prompt message

msglen - length of MSG in characters (type: INTEGER*2)

opnkey - action to be taken (type: INTEGER*2)

A$READ open for reading

A$WRIT open for writing

A$RDWR open for reading and writing

typkey - type of file (type: INTEGER*2)

A$SAMF SAM file

A$DAMF DAM file

name - filename (or treename)

namlen - length of name in characters (type: INTEGER*2)

funit - PRIMOS file unit number (type: INTEGER*2)

In function form:

.TRUE. operation successful

.FALSE. operation unsuccessful or no name supplied

OPNV$A Opens a file on funit, verifies operations, retries if FILE IN USE

<logical> = OPNV$A(opnkey+typkey,name,namlen,funit,verkey,wtime,retrys)

or

CALL OPNV$A(opnkey+typkey,name,namlen,funit,verkey,wtime,retrys)

opnkey - action to be taken (type: INTEGER*2)

A$READ open for reading

A$WRIT open for writing

A$RDWR open for reading and writing

typkey - type of file (type: INTEGER*2)

A$SAMF SAM file

A$DAMF DAM file

name - filename (or treename) of file to be opened

namlen - length of name in characters (type: INTEGER*2)

funit - PRIMOS file unit number (type: INTEGER*2)

verkey - verification key (type: INTEGER*2)

A$NVER no verification

A$VNEW verify new (OK TO MODIFY OLD)

A$OVAP verify new plus overwrite or append if writing

A$VOLD verify old (already exists)

wtime - number of seconds to write if FILE IN USE (type: INTEGER*2)

Notes

Verification and delay are independent functions.

If wtime and retrys are specified non-zero and the file to be opened is IN USE, the open will be retried the specified number of times, with wtime seconds (elapsed time) between each attempt. If the number of retries expires, or if either wtime or retrys is initially 0 and the file is IN USE, the function returns .FALSE.. If verification is requested (verkey.NE.A$NVER), the following actions will be taken:

A$VNEW - if the file already exists and opnkey is either A$WRIT or A$RDWR, the user will be asked if it is OK to modify the old file. If the answer is "NO", the function returns .FALSE.. If the answer is "YES", the file is opened.

A$OVAP - this is the same as A$VNEW except that if an old file is to be modified, the user is also asked if the file should be overwritten or appended to. If the answer is "APPEND", the file will be positioned to End-of-File.

A$VOLD - this is the default case if opnkey=A$READ. If not, and if the named files does not already exist, a new file will not be created and the function returns .FALSE..

If any errors not covered above occur while opening the file or positioning it (A$OVAP), the function returns .FALSE.. If the open is ultimately successful, the function returns .TRUE..

retrys - number of times to retry if FILE IN USE (type: INTEGER*2)

OPVP$A Gets filename from user, opens file, verifies operations, retries if FILE IN USE

<logical> = OPVP$A(msg,msglen,opnkey+typkey,name,namlen,funit, verkey,wtime,retrys)

CALL OPVP$A(msg,msglen,opnkey+typkey,name,namlen,funit, verkey,wtime,retrys)

msg - user-supplied prompt message for name

msglen - length of msg in characters (type: INTEGER*2)

opnkey - action to be taken (type: INTEGER*2)

A$READ open for reading

A$WRIT open for writing

A$RDWR open for reading and writing

typkey - type of file (type: INTEGER*2)

A$SAMF SAM file

A$DAMF DAM file

name - filename (may be a treename)

namlen - length of name in characters (type: INTEGER*2)

funit - PRIMOS file unit number

verkey - verification key (type: INTEGER*2)

A$NVER no verification

A$VNEW verify new (OK TO MODIFY OLD)

A$OVAP verify, new plus overwrite or append if writing

A$VOLD verify old (already exists)

wtime - number of seconds to wait if FILE IN USE (type: INTEGER*2)

retrys - number of times to retry if FILE IN USE (type: INTEGER*2)

Notes

Verification and delay as described below are independent of each other.

If wtime and retrys are specified to be non-zero and the file to be opened is IN USE, the open will be retried the specified number of times, with wtime seconds (elapsed time) between each attempt. If the number of retries expires, or if either wtime or retrys is initially 0 and the file is IN USE, the function returns .FALSE..

If verification is requested (verkey .NE. A$NVER), the following actions will be taken:

A$VNEW - If the file already exists and opnkey is either A$WRIT or A$RDWR, the user will be asked if it is OK to modify the old file. If the answer is "NO", a new file name will be requested. If the answer is "YES", the file is opened.

A$OVAP - This is the same as A$VNEW except that if an old file is to be modified, the user is also asked if the file should be overwritten or appended to. If the answer is "APPEND", the file will be positioned to End-of-File.

A$VOLD - This is the default case if opnkey=A$READ. If not, and if the named file does not already exist, a new file will not be created and a new name will be requested.

If any errors not covered above occur while opening the file or positioning it (A$OVAP), or a name is not supplied when requested, the function returns .FALSE.. If the open is ultimately successful, the function returns .TRUE..

POSN$A Positions pointer in file open on funit

<logical> = POSN$A(poskey,funit,pos)

or

CALL POSN$A(poskey,funit,pos)

poskey - type of positioning (type: INTEGER*2)

A$ABS absolute position

A$REL relative position

funit - PRIMOS file unit number (type: INTEGER*2)

pos - position; relative or absolute (type: INTEGER*4)

If function form is used:

.TRUE. operation successful

.FALSE. operation not successful

RAND$A Updates seed for random number

<real*8> = RAND$A(seed)

or

CALL RAND$A(seed)

seed - input previous seed (type: INTEGER*4)

output new seed

If function form is used:

value is random number between 0.0 and 1.0 (type: REAL*4 or REAL*8)

RNAM$A Reads text input from terminal into buffer

<logical> = RNAM$A(msg,msglen,namkey,name,namlen)

msg - message text

msglen - length of msg in characters (type: INTEGER*2)

namkey - action (type: INTEGER*2)

A$FUPP force input to upper case

A$UPLW do not force upper case

A$RAWI read rest of line

name - name returned

namlen - length of name buffer in characters (maximum=80) (type: INTEGER*2)

This routine fills name with blanks, prints the supplied message and appends ":" to it. It then reads a user response. If the response is not a legal name or if the name provided is too long for the supplied buffer, the error will be reported and msg will be repeated. If no name is provided, the value of the function will be .FALSE.. If a legal name is provided, the function value will be .TRUE..

RNDI$A Generates initializing seed for random number generator

<real*8> = RNDI$A(seed)

or

CALL RNDI$A(seed)

seed - returned time of day in centiseconds (type: INTEGER*4)

In function form:

value is time of day in seconds (type: REAL*4 or REAL*8)

Note

This function is used to initialize the random number generator (see RAND$A). Hence, if the value is exactly zero, 1234567 or 12345.67 will be returned instead.

RNUM$A reads number input into variable with specified number base

<logical> = RNUM$A(msg,msglen,numkey,value)

msg - message text

msglen - length of msg in characters (type: INTEGER*2)

numkey - number base (type: INTEGER*2)

A$DEC decimal

A$OCT octal

A$HEX hexidecimal

value - returned value (type: INTEGER*4)

Notes

This routine will print the supplied message and append ":" to it. It then reads a user response. If the response is not a legal number or if the number provided has too many digits for an INTEGER*4 value, the error will be reported and msg will be repeated. If no number is provided, the value of the function will be .FALSE. and VALUE=0. If a legal number is provided, the function value will be .TRUE. and the value will be returned in value.

Numbers may be preceded by a "+" or "-".

RPOS$A Returns current absolute position of pointer on FUNIT

<logical> = RPOS$A (funit,pos)

or

CALL RPOS$A (funit,pos)

funit - PRIMOS file unit on which file is opened (type: INTEGER*2)

pos - returned absolute position (type: INTEGER*4)

In function form:

.TRUE. successful operation

.FALSE. unsuccessful operation

RWND$A Rewinds file open on funit

<logical> = RWND$A (funit)

or

CALL RWND$A (funit)

funit - PRIMOS file unit number (type: INTEGER*2)

In function form:

.TRUE. successful operation

.FALSE. unsuccessful operation

TEMP$A Opens temporary file in current UFD for reading and writing

<logical> = TEMP$A (typkey,name,namlen,funit)

or

CALL TEMP$A (typkey,name,namlen,funit)

typkey - type of file (type: INTEGER*2)

A$SAMF SAM file

A$DAMF DAM file

name - returned name of file (6-characters)

namlen - length of NAME buffer in characters (minimum 6 characters) (type: INTEGER*2)

funit - PRIMOS file unit number (type: INTEGER*2)

Note

The file name will be of the form T$xxxx where xxxx will be a 4-digit decimal number between 0000 and 9999 inclusive; the actual name is returned in name. If the file is opened successfully, the function value will be .TRUE., otherwise the value will be .FALSE..

TIME$A Returns time of day

<real*8> = TIME$A (time)

or

CALL TIME$A (time)

time - returned time of day in form HR:MN:SC (minimum of 8 characters).

In function form:

value is the time of day in decimal hours. (type: REAL*4 or REAL*8)

TREE$A Tests name to see if it is a treename

<logical> = TREE$A (name,namlen,fstart,flen)

name - file name to be tested

namlen - length of name in characters (type: INTEGER*2)

fstart - returned position of first character of final name (type: INTEGER*2)

flen - length of final filename in characters (type: INTEGER*2)

This routine will scan a file name and determine if it is a treename. If it is a treename, the function is .TRUE. and if not, it is .FALSE.. In addition, the final name (or entire name if not in a tree) is located in the string. Note that if the name is empty, fstart=flen=0.

TRNC$A Truncates file open in funit

<logical> = TRNC$A (funit)

or

CALL TRNC$A (funit)

funit - PRIMOS file unit (type: INTEGER*2)

In function form:

value is .TRUE. if the operation is successful and .FALSE. otherwise.

UNIT$A Tests if file is open on funit

<logical> = UNIT$A (funit)

funit - PRIMOS file unit (type: INTEGER*2)

In function form:

value is .TRUE. if the unit is open and .FALSE. if the unit is not open.

YSNO$A Asks question to be answered YES or NO

<logical> = YSNO$A (msg,msglen,defkey)

msg - user message

msglen - message length in characters (type: INTEGER*2)

defkey - default key (type: INTEGER*2)

A$NDEF	no default accepted
A$DNO	default="NO" (.FALSE.)
A$DYES	default="YES" (.TRUE.)

Notes

This routine will print the supplied message and append "?" to it. It then reads a user response. If the answer is "YES" or "OK", the function value is .TRUE.. If the answer is "NO", the function value is .FALSE.. If an illegal answer is provided or if no default is accepted, msg will be repeated.

User responses may be abbreviated to first 1 or 2 characters.

OPERATING SYSTEM LIBRARY

These subroutines are used mainly by PRIMOS. However, a number of them useful at the applications level are described in detail here.

Complete details will be found in REFERENCE GUIDE, FILE MANAGEMENT SYSTEM (FMS), PDR3110.

Filenames

Filenames can be up to 32 characters long, the first character of which must be alphabetic. Filenames can be composed only of the following characters:

A-Z, 0-9, or the special characters _ # $ & * - . and /

Lower case characters, if specified, are forced to upper case. Control characters (ASCII codes '0 - '237) are not allowed in file names. In the file system calls, file names are either ASCII, packed two characters per word, or character strings (the actual name preceded and followed by a single quote). If the name length specified in a call is longer than the actual length of the name, the name must be followed by a number of trailing blanks sufficient to match the given length.

Passwords

Passwords can be at most 6 characters long. Passwords less than 6 characters must be padded with blanks for the remaining characters. Passwords are not restricted by filename conventions and may contain any characters or bit patterns. It is strongly recommended that passwords not contain blanks, commas, the characters =,!,',@,{,},[,],(,) or lowercase characters. Passwords should not start with a digit. If passwords contain any of the above characters or begin with a digit, the passwords may not be given on a PRIMOS command line to the ATTACH command.

Keys and Error Codes

All keys and error codes are specified in symbolic, rather than numeric form. These symbolic names are defined as PARAMETERS for FORTRAN programs in $INSERT files in a UFD on the master disk called SYSCOM. The key definition file is named KEYS.F for FORTRAN. The error definition file is ERRD.F.

Error Handling

Errors occurring from a subroutine call cause a non-zero value of the argument CODE to be turned. Users should always test CODE after a call for non-zero values to be certain no errors are missed. Error printing and control are performed by the ERRPR$ subroutine:

CALL ERRPR$ (key,code,text,text-length,name,name-length)

key - action to be taken after printing message

K$NRTN exit to PRIMOS, do not allow return to calling program

K$SRTN exit to PRIMOS, return to calling program following a START command

K$IRTN return immediately to calling program

code - an integer variable containing the error code returned by the subroutine generating the error

text - user's message to be printed following standard error message (up to 64 characters)

text-length - length of text in characters

To omit text, specify both text and text-length as 0.

name - user-specified name of program or sub-system, detecting or reporting the error (up to 64 characters)

name-length - length of name in characters

To omit name, specify both name and name-length as 0.

The message format for non-zero values of CODE is:

<standard text>. <user's text, if any> (<name, if any>) e.g.,

ILLEGAL NAME. OPENING NEWFILE (NEWWRT)

These errors are included in the list of run-time errors in Appendix A. They are labelled as New file call errors.

Operating System Subroutines

ATCH$$ Attaches to a UFD and optionally makes it the home UFD.

CALL ATCH$$ (ufd-name,name-length,logical-disk,password,key,code)

ufd-name name of UFD to be attached to (if ufd-name= K$HOME and key=0, attachment is to home UFD)

name-length length in characters of ufd-name (if ufd-name=K$HOME, name-length is ignored.

logical-disk — logical disk to searched for ufd-name when key=K$IMFD

logical-disk	action
K$ALLD	search all started-up logical devices
K$CURR	search MFD of current disk

password — 3-word array containing the owner or non-owner password of ufd-name (if attaching to home UFD, password may be 0)

key — reference-key + set-key

reference-key

K$IMFD attach to ufd-name in MFD on logical-disk
K$ICUR attach to ufd-name in current UFD

set-key

K$SETH set current UFD to home after attaching

code — returns integer-valued error code

COMI$$ Switches command input stream from terminal to command file and vice-versa.

COMO$$ Switches ouptut stream from terminal to file and vice-versa.

CREA$$ Creates a sub-UFD in the current UFD.

CNAM$$ Changes a filename

CALL CNAM$$ (old-name,old-name-length,new-name,new-name-length,code)

old-name — name of file to be changed

old-name-length — number of characters in old-name

new-name — name to be changed to

new-name-length — number of characters in new-name

code — returns integer-valued error code.

Note

CNAM$$ requires owner-rights in the current UFD.

The names of the MFD,BOOT,BADSPT, or the packname may not be changed.

ERKL$$ Reads or sets the Erase and Kill characters.

GPAS$$ Returns passwords of sub-UFD in the current UFD.

NAMEQ$ Compares filename for equivalence.

PRWF$$ Reads, writes, and positions pointer in a SAM or DAM file.

CALL PRWF$$ (read-write-key+position-key+mode,file-unit,LOC(buffer) number-of-words,position-value,words-transferred,code)

read-write-key	action to be taken (mandatory)
K$READ	read number-of-words from file-unit into buffer
K$WRIT	write number-of-words from buffer to file-unit
K$POSN	set current position to value at 32-bit integer in position-value
K$TRNC	truncate files open on file-unit at current position
K$RPOS	return current positions as a 32-bit integer in position-value
position-key	indicates positioning (optional)
K$PRER	move file pointer of file-unit position-value words relative to current position; then perform read-write-key operation
KPOSR	performs read-write-key operation then move file pointer of file-unit position-value words relative to current position
K$PREA	move file pointer of file-unit to absolute position-value then perform read-write-key operation

K$POSA	perform read-write-key operation, then move pointer of file-unit to absolute position-value.

If position-key is omitted, K$PRER is used.

mode	transfer all or convenient number of words (optional)
omitted	read/write number-of-words
K$CONV	read/write convenient number of words up to number-of-words

See REFERENCE GUIDE, FILE MANAGEMENT SYSTEM (FMS), PDR3110 for a discussion of "convenient".

file-unit	file unit on which the file has been opened (by SRCH$$, PRIMOS command, etc.)
buffer	data buffer for read/write. If not needed, specify as LOC(0).
number-of-words	number of words to be transferred (mode=0) or maximum number of words to be transferred (mode=K$CONV). number-of-words may range from 0 to 65535.
position-value	relative or absolute position value (32-bit integer, INTEGER*4). If not needed, specify long-integer zero as 000000 or INTL(0)
words-transferred	the number of words actually transferred when read-write-key=K$READ or K$WRIT; other keys leave ~~RNW~~ this parameter unmodified. (INTEGER*2)
code	returns integer-valued error code

RDEN$$ Reads entry in a UFD.

RDLIN$ Reads line of characters from compressed or uncompressed ASCII disk file.

RDTK$$ Parses the command line, token by token.

REST$$ Restores an R-mode memory image to user memory from a disk file.

RESU$$ Restores an R-mode memory image from a file, sets initial values, and begins execution.

CALL RESU$$ (filename,name-length)

filename — name of the file containing the memory image

name-length — number of characters in filename

Note

An error in the call to RESU$$ causes an error message to be printed automatically and then returns command to PRIMOS.

SATR$$ Sets attributes (protection,data,time,etc.) in a UFD entry.

SAVE$$ Saves an R-mode memory image in user memory by writing it into a disk file.

SPAS$$ Sets the passwords in the current UFD.

SRCH$$ Opens or closes a file.

CALL SRCH$$ (action+reference+newfile,filename,name-length, file-unit,file-type,code)

action — action to be taken (mandatory)

- K$READ — open filename for reading on file-unit
- K$WRIT — open filename for writing on file-unit
- K$RDWR — open filename for reading and writing on file-unit
- K$CLOS — close file by filename or by file-unit
- K$DELE — delete filename
- K$EXST — check existence of filename

reference — modifies action (optional)

- K$IUFD — search for filename in current UFD (this is the default)
- K$ISEG — perform action option on the file that is a segment directory entry in the directory open on file unit filename.
- K$CACC — change access rights of file open on file-unit to action

new-file — specifies type of file to create if filename does not already exist

- K$NSAM — SAM file (this is the default)
- K$NDAM — DAM file
- K$NSGS — SAM segment directory
- K$NSGD — DAM segment directory

filename — name of the file to be opened. If reference=K$ISEG, filename is a file unit on which a segment directory is already open

name-length — number of characters of filename

file-unit — file unit number on which file is to be opened or closed

file-type — returns type of file opened. If call does not open file, its value is unchanged. The values are integers

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	UFD

code — returns an integer-valued error code.

Notes

A UFD may be opened only for reading.

A UFD cannot be deleted unless it is empty.

A segment directory cannot be deleted unless it is of length 0.

SGDR$$ Positions and reads segment directory entries.

SLEEP$ Suspends execution of user process

```
CALL SLEEP$ (interval)
```

interval — the suspension time in milliseconds (type: INTEGER*4)

Notes

1. If interval ≤ 0, no suspension occurs.
2. A QUIT, followed by a START at the user terminal causes immediate return from the SLEEP$ call

---

TEXTO$ Checks the validity of a filename.

TSRC$$ Opens or closes a file anywhere in the PRIMOS file structure.

CALL TSRC$$ (action+new-file,treename,file-unit,character-position,code)

action	action to be taken (mandatory)
K$READ	open treename for reading on file-unit
K$WRIT	open treename for writing on file-unit
K$RDWR	open treename for reading and writing on file-unit
K$DELE	delete file treename
K$EXST	check on existence of treename
new-file	specifies type of file to create if treename does not already exist
K$NSAM	SAM file (this is the default)
K$NDAM	DAM file
K$NSGS	SAM segment directory
K$NSGD	DAM segment directory
treename	a specification of any file in any directory or subdirectory stored in array treename packed two characters per word
file-unit	file unit number on which the file is to be opened or deleted. The file-unit is closed before any action is taken

character-position	a two-element integer array word 1 of entry: the first character in the array that is part of the treename (count starts at 0) returns: one past the last character that was part of the treename. word 2 - the number of characters in the treename.
file-type	returns type of file opened. If call does not open file, its value is unchanged. The values are integers. 0 SAM file 1 DAM file 2 SAM segment directory 3 DAM segment directory 4 UFD
code	returns an integer valued error code

Notes

TSRC$$ always closes the file unit, then attaches to the user's home UFD before attempting any action.

WTLIN$ Writes a line of characters in ASCII format to a disk file in compressed format.

PART 1

FORTRAN LANGUAGE AND COMPILER

GENERALIZED SUBSCRIPTS

There is now no syntactical limitation on subscript expressions. The FORTRAN compiler allows any integer-valued expression as an array subscript.

Use of Generalized Subscipts

Array references have the form

A(S1,S2,...,Sn)

A is the array name

Si is a subscript expression (1<=i<=7)

A subscript expression is any legal FORTRAN long- or short-integer-valued expression. It may contain constants, variables, function references, intrinsic references, and other array references. The nesting limit on any expression is 32 levels of parentheses, whether syntactical, array, or function reference parentheses. Non-integer constants and variables are not allowed within subscript expressions.

Note

Conversion functions (such as IDINT, IFIX, INT) may be used to convert non-integer expressions to integer within a subscript expression.

Example

The following FORTRAN program illustrates the use of generalized subscripts. It deliberately contains some rather bizarre expressions which show the flexibility of subscripting, but is not intended as a model of good coding practice. (POOP is a REAL-valued function.)

```
C
C     GENERALIZED SUBSCRIPTS
C

      REAL A(100,100),B(10),Z
      INTEGER G(3,4,5),H(3000),I,J,K
```

```
C
C     ASSIGNMENT
C
      Z=A(G(H(25*K**2),2,RS(I,H(2))),INTS(Z-A(1,10*H(J))))
     * +B(INTS(POOP(2)))
C
C     IF
C
      IF(Z.NE.B(RS(K,H(K*5)))) GOTO 1000
C
C     CALL
C
1000  CALL POOP1(A(H(INTS(POOP(1))),G(1,J*2,1)),Z)
C
C     ETC.
C
      END
```

FORTRAN DIRECT ACCESS CAPABILITY

Introduction

The FORTRAN compiler and run-time library now support direct access READ and WRITE statements. READ and WRITE statements may contain a record number to randomly access file records. With sequential access, record n-1 must be read or written before record n. The syntax implemented is compatible with both IBM FORTRAN and new ANSI standard FORTRAN.

Direct Access READ and WRITE Statements

The syntax of the direct access READ and WRITE statements is:

```
READ(u'r,f,ERR=s) list                (IBM format)

READ(u,f,REC=r,ERR=s) list            (ANSI format)

WRITE(u'r,f,ERR=s) list               (IBM format)

WRITE(u,f,REC=r,ERR=s) list           (ANSI format)
```

u is a long or short integer constant or variable whose value is the FORTRAN unit number.

Note

The apostrophe (') is required in the IBM form of the direct access READ and WRITE statements.

r is the long or short integer expression whose value is the record number to be accessed.

f is the statement number of the format specifier (optional).

s is the statement number to which control is transferred if a device or format error is encountered during transfer (optional).

The END= specifier is not allowed in the direct access read statement. This restriction is consistent with both IBM FORTRAN and the new ANSI standard FORTRAN.

Usage

Special action is required by the user when creating and opening files to be used for direct access I/O. Files used for direct access I/O should be DAM files. (Direct access I/O statements may be used with SAM files but execution time will be longer). If the file is formatted, the ATTDEV subroutine must be called so that fixed length records are written. (The ATTDEV subroutine is also used to set the record length.) DAM files are created by opening a new file using the K$NDAM subkey in either a SRCH$$ or TSRC$$ call. (See REFERENCE GUIDE, SOFTWARE LIBRARY, PDR3106 for details).

The ATTDEV subroutine may be used to alter the mapping of FORTRAN units to file system units or to change the record size from the default of 60 words (120 characters). The records of a direct access formatted file must be fixed length. This is done by setting the second argument of ATTDEV to 8. The records of an unformatted file are fixed length by default. If the record length of any file exceeds 66 words (132 characters), a COMMON declaration for F$IOBF must be included. The size of F$IOBF must be as large as the largest record size. (See CHANGING RECORD SIZE for details).

A program that creates a direct access file cannot write record n before record n-1 has been written. A separate program should be used. Once the file has been created it can be read or written in random order.

After a direct access I/O statement, the file is positioned at the record following the one just transferred. If the direct access file is then accessed sequentially, using other forms of the READ or WRITE statement, it is not necessary to include the record number. This enhances performance by eliminating the positioning call.

Formatted files used for direct access I/O may be examined by the Editor. They must not be modified using the Editor. The Editor compresses records, giving them variable lengths; files used for direct access I/O must have fixed length records.

IBM Compatibility

The READ and WRITE statements are identical to IBM FORTRAN. The DEFINE FILE and FIND statements of IBM FORTRAN are not supported. The record size in the DEFINE FILE statement must appear in the ATTDEV call. The record size in the DEFINE FILE statement is measured in bytes or 32-bit words rather than 16-bit words required by ATTDEV. If the U specifier is used in the DEFINE FILE statement, the record size of the DEFINE FILE statement should be doubled for the ATTDEV call; otherwise, the record size should be halved.

The ATTDEV call requires INTEGER*2 arguments. If the INTL option is used during compilation, constants used as arguments in the ATTDEV calls must be converted to INTEGER*2 by the INTS function (e.g., INTS(8)).

There is no equivalent of the DEFINE FILE associated variable in Prime's implementation of direct access files. In IBM FORTRAN, the value of the associated variable is the number of the record that follows the record just transferred.

Example 1

```
C     THIS PROGRAM CREATES A DIRECT ACCESS FILE
C     NOTICE CALLS TO ATTDEV AND SRCH$$
C
C
      IMPLICIT INTEGER*2 (A-Z)
C
      PARAMETER NUMREC=100  /* NUMBER OF RECORDS IN FILE
      PARAMETER RECSIZ=40   /* SIZE OF RECORDS IN INTEGER*2 WORDS
      PARAMETER UNIT=5      /* UNIT # USED IN FORTRAN READS/WRITES
      PARAMETER FUNIT=1     /* FILE UNIT # USED IN SRCH$$
$INSERT SYSCOM>KEYS.F
C
C     ATTDEV CALL - FORCES FIXED LENGTH RECORDS
C                   ESTABLISHES MAPPING OF UNIT TO FUNIT
C                   SET RECORD SIZE
      CALL ATTDEV(UNIT,8,FUNIT,RECSIZ)
C
C     OPEN FILE, USE K$NDAM SUBKEY TO FORCE DAM FILE
      CALL SRCH$$(K$WRIT+K$NDAM,'T$SCRATCH',9,FUNIT,TYPE,CODE)
      IF(CODE.NE.0)CALL ERRPR$(K$NRTN,CODE,0,0,0,0)
C
C     IF FILE ALREADY EXISTS, IT MIGHT NO BE DAM FILE
      IF(TYPE.NE.1) WRITE(1,1)
1     FORMAT('NOT A DAM FILE')
C
      DO 10 I=1,NUMREC
C     THE RECORD NUMBER IN THE FOLLOWING STATEMENT IS UNNECESSARY
C     BECAUSE THE RECORDS ARE BEING WRITTEN SEQUENTIALLY
10    WRITE(UNIT'I,2)I
2     FORMAT('THIS IS THE ',I3,' RECORD')
```

```
      CALL EXIT
      END
```

Example 2

```
C     THIS PROGRAM RANDOMLY ACCESSES PREVIOUSLY CREATED
C     DIRECT ACCESS FILE
C
C
      IMPLICIT INTEGER*2 (A-Z)
C
      INTEGER*2 IBUF(40)
      PARAMETER NUMREC=100  /* NUMBER OF RECORDS IN FILE
      PARAMETER RECSIZ=40   /* SIZE OF RECORDS IN INTEGER*2 WORDS
      PARAMETER UNIT=5      /* UNIT # USED IN FORTRAN READS/WRITES
      PARAMETER FUNIT=1     /* FILE UNIT # USED IN SRCH$$
$INSERT SYSCOM>KEYS.F
C
C     ATTDEV CALL - FORCES FIXED LENGTH RECORDS
C                   ESTABLISHES MAPPING OF UNIT TO FUNIT
C                   SET RECORD SIZE
      CALL ATTDEV(UNIT,8,FUNIT,RECSIZ)
C
C     OPEN THE FILE
      CALL SRCH$$(K$RDWR,'T$SCRATCH',9,FUNIT,TYPE,CODE)
      IF(CODE.NE.0)CALL ERRPR$(K$NRIN,CODE,0,0,0,0)
C
C     CHECK IF DAM FILE
      IF(TYPE.NE.1) WRITE(1,1)
1     FORMAT('NOT A DAM FILE')
C
30    WRITE(1,6)
6     FORMAT('RECORD #?')
      READ(1,7)REC
7     FORMAT(I6)
      WRITE(1,2)
2     FORMAT('READ OR WRITE?')
      READ(1,3)I
3     FORMAT(A1)
      IF(I.EQ.'R')GO TO 10
      IF(I.EQ.'W')GO TO 20
      CALL EXIT
C
10    READ(UNIT,4,REC=REC)IBUF
4     FORMAT(40A2)
      WRITE(1,4)IBUF
      GO TO 30
C
20    WRITE(1,8)
8     FORMAT('RECORD INFO?')
      READ(1,4)IBUF
      WRITE(UNIT'REC,4)IBUF
```

```
GO TO 30
END
```

CHANGING RECORD SIZE

The default formatted record length is 60 words (120 characters). A larger record size can be set with the ATTDEV subroutine. This subroutine has two functions:

- Change record size associated with a FORTRAN logical I/O unit number.
- Change the correspondence between the I/O unit number and the physical device.

The syntax is:

```
CALL ATTDEV(logical-unit,device,unit,record-size)
```

logical-unit is the FORTRAN I/O unit number. This is the number used in READ and WRITE statements (1=terminal, 2=paper tape punch/reader, etc.).

device is the position of the physical device in the device-type tables (CONIOC). The acceptable values are:

1	User terminal
2	Paper tape punch/reader
7	Disk file system (Compressed ASCII)
8	Disk file system (Uncompressed ASCII)

unit is the unit number for multi-unit devices (e.g., magnetic tape drive 0-3). If device is the disk file system (7 or 8) then unit is the file unit number (1-16).

record-size is the maximum record size in INTEGER*2 words for the logical-record. Each word will store 2 characters.

If the record size is to exceed 128 words (256 characters), the buffer used by internal FORTRAN subroutines must be increased. This is done by loading a user-created F$IOBF COMMON before loading the FORTRAN library. Insert this statement in the user program:

```
COMMON/F$IOBF/array-name(size)
```

array-name is an arbitrary name

size is the desired buffer size in INTEGER*2 words. Each word stores 2 characters.

CAUTION

It is not possible to increase the buffer size by loading a user-created F$IOBF if the shared libraries are used.

COMPILER ERROR MESSAGES ADDED AT REV. 15

ARRAY NESTING OVFLO

Use of arrays as subscripts in other arrays exceeds allowable nesting limit (32).

END/REC PROHIBITED

The END=statement-number expression cannot be used in a direct access READ or WRITE statement.

INTERNAL ERROR

Some combination of source code statements has generated an unresolvable error. The programmer should never see this error.

PAREN NESTING>31

Nesting of parentheses (syntactical, array, or function reference) in expressions may not exceed 31.

TOO FEW SUBSCRIPTS

Number of subscripts used in an array is fewer than the number originally declared in a DIMENSION or mode specification statement.

64V-MODE COMMON

The FORTRAN compiler and SEG allow some 64V mode FORTRAN programs faster access to variables in COMMON. If a COMMON block is loaded into the same segment as the procedure area or link area which accesses it, the compiled program will address the COMMON variables directly, rather than through a two-word indirect pointer. Thus, careful loading of

routines with frequently accessed COMMON areas into the same segment in 64V mode will cause an appreciable increase in execution speed.

As a consequence, FORTRAN 64V programs compiled with Rev.15 FTN must be loaded using Rev.15 SEG. Attempts to load with earlier versions of SEG will result in SEG errors.

PART 2

REV. 15 PRIMOS AND UTILITIES

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, IDR3109. 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 2000 (octal) and not equal to 6000 (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, 0 is the default. n-levels is the number of levels to go down doing the setting. The read-write lock is interpreted as follows: 0 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 0 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 0. 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.

LOAD - REV. 15 VIRTUAL LOADER

At REV. 15 the R-mode loader (LOAD) is a virtual loader; it has the capability of loading to a temporary file when the size of an object module exceeds the available space in memory. Following is a summary of the new features. For full details, see PTU5Ø.

LOAD first attempts to load into the actual memory locations specified by the object module. As delivered, the buffer space is all of memory below '122ØØØ. For programs compiled in 32R mode, the entire load may be completed in real memory; in that case the action is the same as previous loaders, and the program can be started directly by the EXecute command. Existing command files should execute exactly as before.

For larger 64R mode programs, the loader uses memory as temporary buffer space which can be paged into a temporary file opened in the home UFD. This permits loadng of programs of up to 64K words. This requires that the load module be SAVEd before an EXECUTE. The temporary file is opened when LOAD is first invoked and is not closed until a QUIT or EXECUTE command is given.

Because the loader must remain attached to the home UFD throughout loading, for access to the temporary file, it is no longer possible to do temporary ATTACHes to other directories. Instead, treenames should be used in LOAD commands when files in other directories are required. Existing command files should be modified accordingly.

New Features

- Improved error reporting - short text descriptions instead of 2-letter codes.
- Quit and Attach may be abbreviated "Q" and "A".
- LOAD family of subcommands (LO, FO, P/, F/) enhanced to accept base area definitions:

$$\left\{\begin{matrix}\text{LO}\\ \text{FO}\\ \text{P/LO}\\ \text{F/LO}\end{matrix}\right\} \text{filename} \left\{\begin{matrix}\text{Addrss}\\ *\\ \text{symbol}\end{matrix}\right\} \quad [\text{setbase-}][\text{setbase-2}]...[\text{setbase-9}]$$

- treenames are accepted wherever filenames are required.
- New commands

  PB Set PBREAK to new value

  SY Define or equate symbols

CH Check PBRK against symbol value

ER Control error handling

SZ Enable/disable use of Sector Ø for links

SS Save Symbols (prevent specified symbols from being deleted by EXpunge).

DC Defer common definition.

EN Entire Save. Save memory image of loader and temporary file for creation of overlays

PA Pause. Return to PRIMOS for internal command execution

- 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 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 loadng 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.

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 400 (or higher) for V-mode programs. For details, see PTU54.

SEG

At Rev. 15, there are the following enhancements to SEG as summarized below. For details, see PTU50.

1. Words Ø through '40 in all procedure segments (and segment '4000)

must be reserved for use by SEG and the operating system. These locations cannot be used as constants or temporaries.

2. SEG now supports up to 256 ('400) segments. Split segments no longer occupy two segment positions.

3. All commands which leave SEG's Loader (QUit, REturn and EXecute) now perform the SAve function. This insures that SEG's DELETE command will work on all SEG run files.

4. QUit and ATtach may now be abbreviated to Q and A respectively.

5. SEG's loader now keeps track of the length of COMMON blocks up to one segment in length. An attempt to redefine a COMMON block to a longer length will cause an error. Redefinition to a shorter length is not flagged.

6. Two new map commands have been added:

   MA ... 1Ø Causes all symbols to be printed out one per line in ascending order by address.

   MA ... 11 Causes all symbols to be printed out in alphabetical order, one per line.

   In both cases, the symbol is described by type (COMMON, DIRECT ENTRY CALL, etc).

7. Map format has changed slightly to allow eight-character file names and the COMMON block section has been reformatted to include the length of the COMMON block when this is known. The length of the COMMON block is printed in octal. Also, the map format for procedure symbols has been enhanced. SEG now prints the length of the link frame and the segment of the link frame wen these are known.

8. *SYM now reports the number of symbols in the symbol table (in octal).

9. It is no longer necessary to worry about mixing default loading (SEG assigns the segment numbers) and specific loading (the user assigns the segment numbers). The command formats have not changed, but SEG will now prevent segment assignment conflicts. In addition, SEG keeps track of the end of the procedure portion of a split segment and will not load procedure into the data portion. A segment may be split anywhere (not just on '4000 word boundaries. In addition, programs may now be loaded under the MI option (see below) which permits mixing of data and procedure in the same unsplit segment. This feature may make split segments obsolete.

1Ø. SEG now assigns a stack to a shared procedure in the user's segments (above '4000) rather than placing it in the first

available procedure segment which is usually below '4000.

11. There is a new command at SEG command level which allows the user to determine the Rev. of SEG. The format for this command is VE(rsion).

12. SEG's Loader has a few new commands. These are:

    SE (setbase)
    SS (save symbol)
    MI (mix procedure/data)
    MV (move)

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 IDR3109, the System Administrator's Guide.

PART 3

ERRATA TO PDR3Ø57, FORTRAN PROGRAMMER'S GUIDE

All references to PRIMOS INTERACTIVE USER GUIDE, MAN26Ø2 and its updates, PTU31 and PTU42 should be replaced by REFERENCE GUIDE, PRIMOS COMMANDS, PDR31Ø8 (pp. 1-11, 2-1, 1Ø-1, 11-1).

Page	Change
1-3	In the lower boxes of the figure, add "P35Ø".
3-13	In the -DEFER option add: The time should be enclosed in single quotes.
15-3	In the range of long integers: 2147463647 should be 2147483647.
15-1Ø	Change Internal Function to Statement Function. Change the CALL example to CALL subrname [(arg-1,...arg-n)]
16-17	In the title: insert "Default" before "FORTRAN".
16-22	The line drawn above "General" should be drawn below "General".
19-3	In the D/xx command, the last line should be: D/ may be combined with F/ as either D/F/xx or F/D/xx.
2Ø-5	In EXP add: SP=EXP(SP).
A-2	In the error CONSTANT REQUIRED: "three" should be "where"; "statment" should be "statement".
A-23	add the following error message: NULL READ UNIT PRIMOS Program has attempted to read with a bad unit number. This may be caused by the program overwriting itself (array out of bounds).
C-2	Right-hand column header. Change "Keyboard Input" to "Control Characters".

Note:

The following corrections were previously distributed in OOPS sheet PDR3057-001.

Page	Change
1-5	Replace description of ↑ character with the following: ↑ Escape symbol used in text editor to enter octal codes of non-printing characters, as in ↑207 for CTRL-G or Bell code.
1-6	Under Filename Conventions change the definition of C←filename to "Command Input File" and add the following entries: O←filename Command Output File Note On some output devices backarrow (←) may print as underscore (_).
1-9	in Functions change "logical" to "Boolean (logical)".
1-10	in Sequential Job Processing, the second sentence should read: "Sequential job processing queues requests for phantom users and then executes these jobs one at a time."
1-12	changes comment line from "*" to "/*. (The old comment still works)"
1-13	in paragraph 2 under Libraries change "logical" to "Boolean (logical)."

1-14 in FORMS, second sentence should read:

"These screen forms are an extremely useful tool for the applications programmer writing data entry programs."

3-2 at bottom of page: "C treename" should be "C filename"

3-4 data representation table should be:

ASCII	transfer
BCD	translation to ASCII from 7-track tape
BINARY	transfer verbatim
EBCDIC	transfer to ASCII

3-5 Reading Punched Paper Tape add sentence. First load tape into reader; then assign tape reader.

After "input from tape reader" add "; tape is being read."

3-6 Under Special Characters replace second paragraph with:

In input mode the semicolon (;) is equivalent to a CR (ends a line of input). In edit mode semicolons in a character string are treated as a printing character; otherwise, semicolons separate multiple commands entered on the same line.

3-7 Tab Settings in the last sentence change "commands" to "comments"

Modifying... change "relative" to "absolute"

3-9 For CHANGE, alter "[n] [G] " to "[G] [n]"

For ERASE, change definition to "Sets erase character to character"

3-10 For KILL, change definition to "Sets kill character to character"

LINESZ LINESZ [n]

3-11 For MODIFY, change "[n] [G]" to "[G] [n]"

Replace entire MOVE entry with:

MOVE buffer-1 { buffer-2 / /string/ } Move string or contents of buffer-2 into buffer-1

13 *[n] should be *[n]

F /;D;n; *10 should be F /;D; *10

5-2 bottom of page: under PRIMOS II for FILMEM ALL

clear all user space
except locations occupied by PRIMOS II

6-9 add at top of page

"The file filename must be a file containing object text compiled (or"

13-5 in answers to KEY SIZE delete "W" (3 places)

13-9 DBMS change "COBOL" to "FORTRAN" (2 places)

15-4 Replace the sentence beginning "Long integers range..."

with

"Long integers range from Ø(:ØØØØØØ) to 2147483647 (:17777777777) and from -2147483648 (:2ØØØØØØØØØØ) to -1(:37777777777). The range is from -(2**31) to +(2**31-1)."

15-5 Double Precision Numbers add at end "Only 2 digits can be printed for the exponents."

16-16 PRINT should read:

PRINT f [,list]

Prints the list of elements on the user's terminal according to the format specified in statement f. Equivalent to WRITE (1,f) [list].

20-3 AMAXØ, AMAX1, AMINØ, AMIN1. Remove V-identity restrictio

20-4 DATAN2 "CP2" should be "DP2"

20-5 DMAX1, DMIN1, Remove V-identity restriction

20-7 MAXØ, MAX1, MINØ, MIN1. Remove V-identity restriction

RND-in table heading "I1" should be "I";"12" should be "SP"

20-43 words-transmitted "RNW" should be "this parameter"

C-1 CTRL-P "'200" should be "'220".

APPENDICES

APPENDIX A

ERROR MESSAGES

INTRODUCTION

Error messages are given in the following order:

FORTRAN Compiler Error Messages
Linking Loader Error Messages
SEG Loader Error Messages
Run-Time Error Messages

In each group errors are listed alphabetically.

Run-time error messages beginning with a filename, device name, UFDname, etc., are alphabetized according to the first word which is constant. The user should have no trouble in determining this word (the second word in the message). Leading asterisks, etc., are ignored in alphabetizing. All run-time errors have been grouped together to facilitate lookup by the user.

Table A-1. Compiler Error Messages

Error Message	Meaning
ARG LIST REQUIRED	Argument list not specified in FUNCTION statement.
ARRAY NAME REQUIRED	Something other than an array name appeared in a position where only an array name is allowed. (example: ENCODE or DECODE statement)
ARRAY/BLOCK OVERFLOW	Array/block exceeds space allocated to user.
CHAR STRING SIZE	A character string was not terminated, or a string in a DATA statement was longer than the associated variable list.
COMMON NAME ILL.	Illegal use of a name already declared in COMMON.
COMPILER OVERFLOW	Insufficient memory to compile program.
CONFLICTING DECLARN	Name(s) declared as more than one data mode.
CONSTANT REQUIRED	A name appeared ~~three~~ where only a constant or parameter is allowed. (i.e., DIMENSION ~~statment~~ statement in a main program)
CONSTANT TOO LARGE	Constant exponent excessive for data type.
DATA MODE ERROR	Illegal mode mixing in expression, expression mode not of required type, or constant in DATA statement is of different mode than associated name in variable list.
DIVISION BY ZERO	Attempt has been made to divide by a zero constant.
EXCESS CONSTANTS	Number of constants in DATA statement exceed variables for storing them.
EXCESS SUBSCRIPTS	Too many subscripts in EQUIVALENCE or DATA list item.
FUNCT VAL UNDEFINED	The function name was not assigned a value in FUNCTION subprogram.

GBL MDE/IMPL CNFLCT	IMPLICIT statement and global mode specification may not be used in same program unit.
ILL. CONSTANT EXPR.	Variables found in a PARAMETER statement.
ILL. DO TERMINATION	Improper DO loop nesting, or an illegal statement terminating a DO loop.
ILL. EQUIVALENCE	EQUIVALENCE group violates EQUIVALENCE rules or specifies an impossible equivalencing.
ILL. LOGICAL IF	A logical IF contained in a logical IF, or a DO statement contained in a logical IF.
ILL. OVER 64K COMMON	A COMMON area exceeds 64K words of user memory.
ILL. STMT NO. REF	Reference to a specification statement number.
ILL. UNARY OP USAGE	Improper use of an operator in an expression.
ILL. USE OF ARG	SUBROUTINE or FUNCTION statement used in COMMON, EQUIVALENCE, or DATA statement.
ILL. USE OF CLMN. 6	Continuation line found without a continuation or statement line preceding it.
ILL. USE OF STMT	Statement illegal in context of the program. For example, RETURN in a main program, SUBROUTINE not the first statement, or specification statements out of order. If an undeclared array name is used on the left in an assignment statement, the compiler will assume it is a statement function definition and therefore generate this error.
INCONSISTENT USAGE	The use of the name listed in the error message conflicts with earlier usage. This message also will be generated at the END statement in a SUBROUTINE subprogram if the subroutine name is used within the subprogram.
INTEGER REQUIRED	A non-integer name or constant appeared where only an integer name or constant is allowed.
MULT DEF STMT NO.	The statement number of the current line has already been defined.
NAME REQUIRED	A constant appeared where only a name is allowed.

NO END STMT	The last statement in the source was not an END statement.
NO PATH TO STMT	The current statement does not have a statement number and the previous statement was an unconditional transfer of control.
NONCOMMON DATA	A BLOCK DATA subprogram initialized data not defined in COMMON or contained executable statements.
PARENTHESIS MISSING	Incorrect parenthesis used in an implied DO loop in an I/O statement.
PROG SIZE OVERFLOW	Program too large for allocated user space.
SAVE ITEM ILLEGAL	Improper item in SAVE statement (function name, array element, etc).
STMT NAME SPELLING	A statement name was recognized by its first four characters, but the remaining spelling was incorrect.
STMT NO. MISSING	A FORMAT statement appeared without a statement number.
SUBPGM/ARR NAME ILL	Illegal usage of subprogram or array name.
SUBPROGRAM NAME ILL	Illegal usage of subprogram name.
SYMBOLIC SUBSCR ILL	Illegal usage of symbolic subscript in a specification statement.
SYNTAX ERROR	General syntax error, context usually shows offending character(s).
UNDECLARED VARIABLE	The listed variable did not appear in a specification statement (generated when the undeclared variable check option is enabled).
UNDEFINED STMT NO.	The listed statement number was not defined in the subprogram. The listed line number is the line number of the last reference to the statement number.
UNRECOGNIZED STMT	The compiler could not identify the statement.

Table A-2. Linking Loader Error Messages

Error Messages	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 FORTRAN, 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 op code with an address of zero. Loading continues. Object code may be in 64V mode; recompile and then restart load.
MO	Memory Overflow Errors As users' programs become larger, MO (memory overflow) errors become more common. 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: 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. The sector zero base area is full - the next free location is '1000. The size of the sector zero base may be increased by a SETB '100 command at the beginning of the load

- if locations ´100 to ´200 are free - or an AU command may be used to insert base areas throughout the load. Alternatively, recompile using the Loader base area conservation option. (see AUTOMATIC, SETBASE)

c. *CMLOW is near *PBRK. COMMON should be moved to higher memory using the COMMON command. Re-initialize using the FILMEM command. If COMMON must be moved above ´100000, it may be necessary to recompile the program in 64R mode and the program load must begin with a MO D64R command. (see COMMON, MODE)

d. 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 6).

e. None of the above. The user´s program requires initialized COMMON. COMMON is usually defaulted to overwrite the space used by the Loader. Those locations between the bottom of the symbol table and the top of the Loader cannot be initialized as this would destroy the loader. The solution is to use a COMMON command to move COMMON out of the way of the loader. Possibly the user will want to use HILOAD to permit COMMON to use the locations normally used by the Loader. (see COMMON)

OR

Out of Reach. An attempt has been made to reference a COMMON area that is out of reach of the load mode.

Begin the load with an MO D64R command, or move COMMON to ´100000 or lower with the CO command. (see COMMON, MODE)

NS

Never Sectored. Code is being loaded in 16S or 32S mode, which will not properly execute in a sectored mode. Loading is discontinued.

Don´t include the D16S or D32S command in the load session, or check the PMA source module to see if it includes one of these commands. (see MODE). Unlikely error for FORTRAN programmer to encounter.

N6

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

Loading is discontinued.

Recompile 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)

Table A-3. SEG Loader Error Messages

Error Message	Meaning
BAD OBJECT FILE	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	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	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	An unrecognized command was entered or the filenames/parameters following the command are incorrect. Usually not fatal.

EXTERNAL MEMORY REFERENCE TO ILLEGAL SEGMENT

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.

ILLEGAL SPLIT ADDRESS

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

An attempt was made to load a 64R mode progran wherein COMMON would be allocated to an illegal segment number. Recompile in 64V mode. FATAL, abort load.

NO FREE SEGMENTS TO ASSIGN

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

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

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.

SECTOR ZERO BASE AREA FULL

Extremely unlikely to occur. Not correctable at applications level. Check with analyst. FATAL, abort load.

SEGMENT WRAP AROUND TO ZERO	An attempt has been made to load a 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.

Table A-4
Run-Time Error Messages

Message	Meaning	Origin
ACCESS VIOLATION	Attempt to perform operations in segments to which user has no right.	64V mode
****AD	Overflow or underflow in double-precision addition/subtraction (A$66,S$66).	R-mode function
ALL REMOTE UNITS IN USE	Attempt made to assign a remote unit when none are available. (Network error) [E$FUIU]	New file call
**** ALOG/ALOG 10 - ARGUMENT <=0	Argument not greater than zero used in logarithm (ALOG, ALOG 10) function.	V-mode function
<filename> ALREADY EXISTS	Attempt to create a file or UFD with the name of one already existing. [CZ]	Old file call
ALREADY EXISTS	Attempt made to create, in the UFD, a sub-UFD with the same name as one already existing. (CREA$$) [E$EXST]	New file call
****AT	Both arguments are zero in the ATAN2 function.	R-mode function

**** ATAN2 - BOTH ARGUMENTS = Ø V-mode function

Both arguments are zero in the ATAN2 function.

**** ATTDEV - BAD UNIT V-mode call

Incorrect logical device unit number in the ATTDEV subroutine call.

BAD CALL TO SEARCH Old file call

Error in calling the SEARCH subroutine, e.g., incorrect parameter. [SA]

BAD DAM FILE Old file call

The DAM file specified has been corrupted - either by the programmer or by a system problem. [SS]

BAD DAM FILE New file call

The DAM file specified has been corrupted - either by the programmer or by a system problem. (PRWF$$, SRCH$$). [E$BDAM]

BAD FAM SVC New file call

System problem; will not be seen by applications programmer [E$BFSV]

BAD KEY New file call

Incorrect key value specified in subroutine argument. (ATCH$$, RDEN$$, SATR$$, SRCH$$, SGDR$$) [E$BKEY]

BAD PARAMETER Old file call

Incorrect parameter value in subroutine call [SA]

BAD PASSWORD — Old file call

Incorrect password specified in ATTACH subroutine. Returns to PRIMOS level attached to no UFD. [AN]

BAD PASSWORD — New file call

Incorrect password specified in ATCH$$ subroutine. Returns to PRIMOS level attached to no UFD. [ATCH$$] [E$BPAS]

Note

To protect UFD privacy the system does not allow the user to trap BAD PASSWORD errors.

BAD RTNREC — PRIMOS

System error.

BAD SEGDIR UNIT — New file call

Error generated in accessing segment directory, i.e., PRIMOS file unit specified is not a segment directory. (SRCH$$) [E$BSUN]

BAD SEGMENT NUMBER — New file call

Attempt made to access segment number outside valid range. [E$BSGN]

BAD SVC — PRIMOS

Bad supervisor call. In FORTRAN usually caused by program writing over itself.

BAD TRUNCATE OF SEGDIR — New file call

Error encountered in truncating segment directory. (SGDR$$) [E$BTRAN]

BAD UFD — New file call

UFD has become corrupted. (ATCH$$, CREA$$, GPAS$$, RDEN$$, SATR$$, SRCH$$) [E$BUFD]. Calls to RDEN$$ return this as a trappable error; other commands return to the PRIMOS command level.

BAD UNIT NUMBER — New file call

PRIMOS file unit number specified is invalid - outside legal range. (PRWF$$, RDEN$$, SRCH$$, SGDR$$). [E$BUNT]

BEGINNING OF FILE — New file call

Attempt was made to access locations before the beginning of the file. (PRWF$$, RDEN$$, SGDR$$) [E$BOF]

****BN<n> — R-mode function

Device error in REWIND command on FORTRAN logical unit n.

BUFFER TOO SMALL — New file call

Buffer as defined is not large enough to accomodate entry to be read into it. (RDEN$$) [E$BFTS]

**** DATAN - BAD ARGUMENT — V-mode function

The second argument in the DATAN2 function is zero.

****DE — R-mode function

The exponent of a double-precision number has overflowed.

DEVICE IN USE — New file call

Attempt was made to ASSIGN a device currently assigned to another user. [E$DVIU]

DEVICE NOT ASSIGNED — New file call

Attempt was made to perform I/O operations on a device before assigning that device. [E$NASS]

DEVICE NOT STARTED — New file call

Attempt was made to access a disk not physically or logically connected to the system; if disk must be accessed, systems manager must start it up. [E$DNS]

**** DEXP - ARGUMENT TOO LARGE — V-mode function

The argument of the DEXP function is too large; i.e., it will give a result outside the legal range.

**** DEXP - OVERFLOW/UNDERFLOW — V-mode function

An overflow or underflow condition occurred in calculating the DEXP function.

DIRECTORY NOT EMPTY — New file call

Attempt was made to delete a non-empty directory. (SRCH$$) [E$DNTE]

DISK FULL — Old file call

No more room for creating/extending any type of file on disk. [DJ]

DISK FULL — New file call

No more room for creating/extending any type of file on disk. (CREA$$, PRWF$$, SRCH$$, SGDR$$). [E$DKFL]

Note

Space may be made available. Use the internal PRIMOS commands ATTACH, LISTF, and DELETE to remove files which are no longer needed.

DISK I/O ERROR — New file call

A read/write error was encountered in accessing disk. Returns immediately to PRIMOS level. Not correctable by applications programmer. (ATCH$$, CREA$$, GPAS$$, PRWF$$, RDEN$$, SATR$$, SRCH$$, SGDR$$). [E$DISK]

DK ERROR — Old file call

A read/write error was encountered in accessing disk. [WB]

****DL — R-mode function

Argument was not greater than zero in DLOG or DLOG2 function.

**** DLOG/DLOG2 - ARGUMENT <=Ø — V-mode function

Argument not greater than zero used in DLOG or DLOG2 function.

****DN <n> — R-mode function

Device error (end of file) on FORTRAN logical unit n.

**** DSIN/DCOS - ARGUMENT RANGE ERROR — V-mode function

Argument outside legal range for DSIN

or DCOS function.

**** DSQRT - ARGUMENT <Ø V-mode function

Negative argument in DSQRT function.

****DT R-mode function

Second argument is zero in DATAN2 function. (D$22)

DUPLICATE NAME Old file call

Attempt to create/rename a file with the name of an existing file. [CZ]

****DZ R-mode function

Attempt to divide by zero (double-precision)

END OF FILE New file call

Attempt to access location after the end of the file. (PRWF$$, RDEN$$, SGDR$$) [E$EOF]

****EQ R-mode function

Exponent overflow. (A$81)

****EX R-mode function

Exponent function value too large in EXP or DEXP function.

**** EXP - ARGUMENT TOO LARGE V-mode function

The argument of the EXP function is too large, i.e., it will give a result outside the legal range.

**** EXP - OVERFLOW V-mode function

Overflow occurred in calculating the EXP function.

FAM ABORT — New file call

System error. [E$FABT]

FAM - BAD STARTUP — New file call

System error. [E$FBST]

FAM OP NOT COMPLETE — New file call

Network error. [E$FONC]

****FE — R-mode funciton

Error in FORMAT statement. FORMAT statements are not completely checked at compile time. (F$IO)

FILE IN USE — New file call

Attempt made to open a file already opened or close/delete a file opened by another user, etc. (SRCH$$) [E$FDEL]

FILE OPEN ON DELETE — New file call

Attempt made to delete a file which is open. (SRCH$$) [E$FDEL]

FILE TOO BIG — New file call

Attempt made to increase size of segment directory beyond size limit. (SGDR$$) [E$FLITB]

****FN <n> — R-mode function

Device error in BACKSPACE command on FORTRAN logical unit n.

**** F$BN - BAD LOGICAL UNIT — V-mode function

FORTRAN logical unit number out of range.

**** F$FLEX - DOUBLE-PRECISION DIVIDE BY ZERO 64V mode

Attempt has been made to divide by zero

**** F$FLEX - DOUBLE-PRECISION EXPONENT OVERFLOW 64V mode

Exponent of a double-precision number had exceeded maximum.

**** F$FLEX - REAL => INTEGER CONVERSION ERROR 64V mode

Magnitude of real number too great for integer conversion.

**** F$FLEX - SINGLE-PRECISION DIVIDE BY ZERO 64V mode

Attempt has been made to divide by zero.

**** F$FLEX - SINGLE-PRECISION EXPONENT OVERFLOW 64V mode

Exponent of a single-precision number has exceeded maximum.

**** F$IO - FORMAT ERROR V-mode function

Incorrect FORMAT statement; FORMAT statements are not completely checked at compile time.

**** F$IO - FORMAT/DATA MISMATCH V-mode function

Input data does not correspond to FORMAT statement.

**** F$IO - NULL READ UNIT V-mode function

FORTRAN logical unit for READ statement not configured properly.

****II R-mode function

Exponentiation exceeds integer size.
(E$11)

ILLEGAL INSTRUCTION AT <octal-location> R mode and 64V mode

An instruction at <octal-location>
cannot be identified by the computer.

ILLEGAL NAME New file call

Illegal name specified for a file or
UFD (CREA$$, SRCH$$) [E$BNAM]

ILL REMOTE REF New file call

Attempt to perform network operations
by user not on network. [E$IREM]

ILLEGAL SEGNO 64V mode

Program referenced a non-existent
segment or a segment number greater
than those available to the user.

****IM R-mode function

Overflow or underflow occurred during a
multiply. (M$11, E$11)

<filename> IN USE Old file call

Attempt made to open a file already
opened, or close/delete a file opened
by another user, etc. [SI]

INVALID FAM FUNCTION CODE New file call

System error. [E$FIFC]

**** I**I - ARGUMENT ERROR V-mode function

Exponentiation exceeds integer size.

****LG R-mode function

Argument not greater than zero in ALOG or ALOG10 function.

MAX REMOTE USERS EXCEEDED New file call

No more users may access the network. [E$TMRU]

NAME TOO LONG New file call

Length of name in argument list exceeds 32 characters. [E$NMLG]

NO AVAILABLE SEGMENTS 64V mode

Additional segment(s) required - none available. User should log out to release assigned segments and try again later.

NO RIGHT New file call

User does not have access right to file, or write access in a UFD to create a sub-UFD. (CREA$$, GPAS$$, SATR$$, SRCH$$, SGDR$$) [E$NRIT]

NO TIME New file call

Clock not started; system error. [E$NTIM]

NO UFD ATTACHED Old file call

User not attached to a UFD [AL, SL]. Usually after attempt to attach with a bad password.

NO UFD ATTACHED New file call

User not attached to a UFD. (ATCH$$, CREA$$, GPAS$$, SATR$$, SRCH$$). [E$NATT] Usually after attempt to attach with a bad password.

NO VECTOR R and 64V mode

User error in program has caused PRIMOS to attempt to access an unloaded element.

1. A UII, PSU, or FLEX to location Ø
2. Trap to location Ø
3. SVC switch on, SVC trap and location ´65 is Ø.

NOT A SEGDIR New file call

Attempt to perform segment director operations on a file which is not a segment directory. (SRCH$$) [E$NTSD]

NOT A UFD Old file call

Attempt to perform UFD operations on a file which is not a UFD. [AR]

NOT A UFD New file call

Attempt to perform UFD operations on a file which is not a UFD. (ATCH$$, GPAS$$, SRCH$$). [E$NTUD]

<device-name> NOT ASSIGNED PRIMOS

User program has attempted to access an I/O device which has not been assigned to the user by a PRIMOS command.

<filename> NOT FOUND Old file call

File specified in subroutine call not found. [AH, SH]

<filename> NOT FOUND New file call

File specified in subroutine call not found. (ATCH$$, GPAS$$, SATR$$, SRCH$$) [E$FNTF]

<filename> NOT FOUND IN SEGDIR — New file call

Filename specified in subroutine call not found in specified segment directory. (SRCH$$, SGDR$$) [E$FNTS]

NULL READ UNIT — PRIMOS *

OLD PARTITION — New file call

Attempt to perform, in an old file partition, an operation possible only in a new file partition; e.g., date/time information access. (SATR$$) [E$OLDP]

****PA<n> — R-mode function

PAUSE statement n (octal) encountered during program execution.

**** PAUSE<n> — V-mode function

PAUSE statement n (octal) encountered during program execution.

POINTER FAULT — 64V mode

Reference has been made to an argument or instruction not in memory. The two usual causes of this are an incomplete load (unsatisfied references), or incomplete argument list in a subroutine or function call.

POINTER MISMATCH — PRIMOS

internal file pointers have become corrupted; no user remedial action possible. System manager must correct [PC, DC, AC]

PROGRAM HALT AT <octal-location> — R mode and 64V mode

Program control has been lost. The program has probably written over itself or the load was incomplete (R-mode).

* Program has attempted to read with a bad unit number. This may be caused by the program overwriting itself (array out of bounds).

PRWFIL BOF — Old file call

Attempt by PRWFIL subroutine to access location before beginning of file. [PG]

PRWFIL EOF — Old file call

Attempt by PRWFIL subroutine to access location after end of file. [PE]

PRWFIL POINTER MISMATCH — Old file call

The internal file pointers in the PRWFIL subroutine have become corrupted.

PRWFIL UNIT NOT OPEN — Old file call

The PRWFIL subroutine is attempting to perform operations using a PRIMOS file unit number on which no file is open.

PTR MISMATCH — New file call

Internal file pointers have become corrupted. No user remedial action possible. (ATCH$$, CREA$$, GPAS$$, PRWF$$, RDEN$$, SATR$$, SRCH$$, SGDR$$). [E$PTRM]. Consult system manager.

REMOTE LINE DOWN — New file call

Remote call-in access to computer not enabled. [E$RLDN]

****RI — R-mode function

Argument is too large for real-to-integer conversion. (C$12)

****RN<n> — R-mode functon

Device error or end-of-file in READ statement on FORTRAN logical unit n.

****SE R-mode funciton

Single precision exponent overflow.

SEG-DIR ER Old file call

Error encountered in segment directory operation. [SQ]

SEGDIR UNIT NOT OPEN New file call

Attempt has been made to reference a segment directory which is not open. (SRCH$$) [E$SUNO]

SEM OVERFLOW New file call

System error. [E$SEMO]

**** SIN/COS - ARGUMENT TOO LARGE V-mode function

Argument too large for SIN or COS function.

****SQ R-mode function

Negative argument in SQRT or DSQRT functon.

**** SQRT - ARGUMENT<Ø V-mode functon

Negative argument in SQRT function.

****ST<n> R-mode function

STOP statement n (octal) encountered during program execution.

**** STOP<n> V-mode function

STOP statement n (octal) encountered during program execution.

****SZ R-mode function

Attempt to divide by zero (single-precision).

TOO MANY UFD LEVELS New file call

Attempt to create more than 72 levels of sub-UFDs. This error occurs only on old file partitions; new file partitions have no limit on UFD levels. [E$TMUL]

UFD FULL Old file call

No more room in UFD. [SK]

UFD FULL New file call

UFD has no room for more files and/or sub-UFD's. Occurs only in old file partitions. (CREA$$, SRCH$$)L [E$FDFL]

UFD OVERFLOW Old file call

No more room in UFD.

UNIT IN USE Old file call

Attempt to open file on PRIMOS file unit already in use [SI].

UNIT IN USE New file call

Attempt to open file on PRIMOS file unit already in use. (SRCH$$). [E$UIUS]

UNIT NOT OPEN Old file call

Attempt to perform operations with a file unit number on which no file has been opened. [PD, SD]

UNIT NOT OPEN New file call

Attempt to perform operations with a file unit number on which no file has been opened. (PRWF$$, RDEN$$, SRCH$$, SGDR$$). [E$UNOP]

UNIT OPEN ON DELETE — Old file call

Attempt to delete file without having first closed it. [SD]

****WN<n> — R-mode function

Device error or end-of-file in WRITE statement on FORTRAN logical unit n.

****XX — R-mode function

Integer argument >32767.

APPENDIX B

SYSTEM DEFAULTS AND CONSTANTS

```
EDITOR (ED)
   INPUT (TTY)
   LINESZ 144
   MODE NCOLUMN
   MODE NCOUNT
   MODE NNUMBER
   MODE NPROMPT
   MODE PRALL
   VERIFY
Symbols
   BLANK    #
   CPROMPT  $
   DPROMPT  &
   ERASE    "
   ESCAPE   ↑
   KILL     ?
   SEMICO   ;   end of line or command
   TAB      \
   WILD     !

LINKING-LOADER (LOAD or HILOAD)
   Memory Location:
     LOAD   '60000-'63777
     HILOAD '174000-'177777
   Loading address:  current *PBRK value
   Library:  FTNLIB  Fortran library
   MODE D32R
   Sector Zero Base Area:
     Base start at location '200
     Base range '600 words

SEGMENTED-LOADER (SEG)
   Loading address:  current TOP+1 in
            current procedure segment
   Stack size:  '6000 words
   Library:  PFTNLB and IFTNLB libraries

EXECUTION
   A-register value       0
   B-register value       0
   X-register value       0
   Program start address  '1000
   Bits 4-6 of Keys:
     000 16K, sector-address
     001 32K, sector-address
     010 64K, relative-address
     011 32K, relative-address
     110 64K, segmented-address
```

```
PRIMOS
   ERASE        "
   INTERRUPT    CTRL/P or BREAK
   KILL         ?
   Files:
     created with protection
        owner all access rights ('7)
        non-owner no access rights ('0)

FORTRAN COMPILER (FTN)
   BINARY       disk-file
   ERRTTY
   FP
   INPUT        disk-file
   INTS
   LISTING NO   no listing file
   NOBIG
   NODCLVAR
   NOTRACE
   NOXREF
   SAVE
   32R
```

APPENDIX C

ASCII CHARACTER SET

The standard character set used by Prime is the ANSI, ASCII 7-bit set.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage.

- Output Parity is normally transmitted as a zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (e.g., MLC) may have hardware to assist in parity generations.
- Input Parity is ignored by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (e.g., MLC) may assist in parity error detection.
- The Prime internal standard for the parity bit is one, i.e., '200 is added to the octal value.

KEYBOARD INPUT

Non-printing characters may be entered into text with the logical escape character ^ and the octal value. The character is interpreted by output devices according to their hardware.

Example: Typing ^207 will enter one character into the text.

CTRL-P	('~~200~~ 220)	is interpreted as a .BREAK.
.CR.	('215)	is interpreted as a newline (.NL.)
"	('242)	is interpreted as a character erase
?	('277)	is interpreted as a line kill
\	('334)	is interpreted as a logical tab (Editor)

Table C-1

ASCII Character Set (Non-Printing)

Octal Value	ASCII Character	Comments/Prime Usage	~~Keyboard Input~~ Control Characters
200	NULL	Null character - filler	^@
201	SOH	Start of header (communications)	^A
202	STX	Start of text (communications)	^B
203	ETX	End of text (communications)	^C
204	EOT	End of transmission (communications)	^D
205	ENQ	End of I.D. (communications)	^E
206	ACK	Acknowledge affirmative (communications)	^F
207	BEL	Audible alarm (bell)	^G
210	BS	Back space one position (carriage control)	^H
211	HT	Physical horizontal tab	^I
212	LF	Line feed; ignored as terminal input	^J
213	VT	Physical vertical tab (carriage control)	^K
214	FF	Form feed (carriage control)	^L
215	CR	Carriage return (carriage control) (1)	^M
216	SO	RRS - red ribbon shift	^N
217	SI	BRS - black ribbon shift	^O
220	DLE	RCP - relative copy (2)	^P
221	DC1	RHT - relative horizontal tab (3)	^Q
222	DC2	HLF - half line feed forward (carriage control)	^R
223	DC3	RVT - relative vertical tab (4)	^S
224	DC4	HLR - half line feed reverse (carriage control)	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronicity (communications)	^V
227	ETB	End of transmission block (communications)	^W
230	CAN	Cancel	^X
231	EM	End of Medium	^Y
232	SUB	Substitute	^Z
233	ESC	Escape	^[
234	FS	File separator	^\
235	GS	Group separator	^]
236	RS	Record separator	^^
237	US	Unit separator	^_

Notes

1. Interpreted as .NL. at the terminal.

2. .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceeding line.

3. Next byte specifies number of spaces to insert.

4. Next byte specifies number of line feeds to insert.

Conforms to ANSI X3.4-1968

The parity bit ('200) has been added for Prime-usage.

Non-printing characters (^c) can be entered at most terminals by typing the (control) key and the c character key simultaneously.

Table C-2

ASCII Character Set (Printing)

Octal Value	ASCII Character	Octal Value	ASCII Character	Octal Value	ASCII Character
240	.SP. (1)	300	@	340	` (9)
241	!	301	A	341	a
242	" (2)	302	B	342	b
243	# (3)	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' (4)	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	313	K	353	k
254	, (5)	314	L	354	l
255	-	315	M	355	m
256	.	316	N	356	n
257	/	317	O	357	o
260	0	320	P	360	p
261	1	321	Q	361	q
262	2	322	R	362	r
263	3	323	S	363	s
264	4	324	T	364	t
265	5	325	U	365	u
266	6	326	V	366	v
267	7	327	W	367	w
270	8	330	X	370	x
271	9	331	Y	371	y
272	:	332	Z	372	z
273	;	333	[	373	{
274	<	334	\	374	|
275	=	335	]	375	}
276	>	336	^(7)	376	~ (10)
277	? (6)	337	_ (8)	377	DEL (11)

Notes

1. Space forward one position

2. Terminal usage - erase previous character

3. £ in British use

4. Apostrophe/single quote

5. Comma

6. Terminal usage - kill line

7. 1963 standard ↑; terminal use - logical escape

8. 1963 standard ←

9. Grave

10. 1963 standard ESC

11. Rubout - ignored

Conforms to ANSI X3.4-1968
1963 variances are noted

The parity bit ('200) has been added for Prime usage.

INDEX