PRIME

THE FORTRAN PROGRAMMER'S GUIDE

FDR3057

This is your PRIME **FORTRAN Programmer's Guide.** This one document contains everything you need to write, modify, compile, load, execute and debug most FORTRAN applications using the PRIMOS Operating System.

We have extracted from the PRIME Reference Documentation Family all the commands and functions you will need as a FORTRAN programmer.

Please read the next few pag[illegible] carefully. They tell you what this document is, how to use it, and where you can find what you need.

The **FORTRAN Programmer's Guide** contains the following parts:

- An Overview of PRIME FORTRAN
- How to Use FORTRAN under PRIM[illegible]
- Advanced FORTRAN Programming Techniques
- A Complete Reference to the FORTRAN Language.
- A Complete Reference to related Utilities

FDR Cover Design by William Agush, PRIME Staff

The FORTRAN Programmer's Guide

Published by Prime Computer, Incorporated
Technical Publications Department
145 Pennsylvania Avenue, Framingham, MA 01701

Copyright ©1979 by Prime Computer, Inc.
All rights reserved.

The information contained in this document is subject to change without notice and should not be construed as a commitment by Prime Computer, Incorporated. Prime Computer assumes no responsibility for any errors that may appear in this document.

This document reflects the software as of Master Disk Revision Level 16.

PRIMOS® is a trademark of Prime Computer, Inc.

Credits.

Concept and Production
William I. Agush

Typesetting.
Allied Systems

Covers.
Mark-Burton

Text.
Eusey Press

The FORTRAN Programmer's Guide

---

by Anthony Lewis

PRIME SOFTWARE DOCUMENTATION SUMMARY

Description	Software Rev. #	Document Number	Price
FORTRAN			
• The FORTRAN Programmer's Guide			
Bound edition	16	FDR3057-101A†	$15.00
Loose-leaf edition	16	FDR3057-101B†	$15.00
• The FORTRAN Programmer's Companion	16	FDR3338†	$ 2.00
COBOL			
• The COBOL Programmer's Guide	16	PDR3056†	$15.00
RPGII			
• The RPGII Programmer's Guide	16	PDR3031†	$15.00
• The RPGII Debugging Template	14-16	FDR3275	$ 2.00
BASIC/VM (COMPILED)			
• The BASIC/VM Programmer's Guide	16	PDR3058†	$15.00
• The BASIC/VM Programmer's Companion	16	FDR3341†	$ 2.00
BASIC (INTERPRETIVE)			
• The Interpretive BASIC Programmer's Guide	14,15	IDR1813	$15.00
Technical update	16	PTU59†	$ 2.00
ASSEMBLY LANGUAGE			
• The Assembly Language Programmer's Guide			
Bound edition	16	FDR3059-101A†	$15.00
Loose-leaf edition	16	FDR3059-101B†	$15.00
• The Assembly Language Programmer's Companion	15,16	FDR3340	$ 2.00
• The System Architecture Reference Guide	16	PDR3060†	$15.00
PRIMOS OPERATING SYSTEM/UTILITIES			
• The PRIMOS Commands Reference Guide			
Bound edition	16	FDR3108-101A†	$15.00
Loose-leaf edition	16	FDR3108-101B†	$15.00
• The PRIMOS Commands Programmer's Companion	16	FDR3250†	$ 2.00
• The System Administrator's Guide	16	PDR3109†	$15.00
• The System Administrator's Programmer's Companion	16	FDR3622†	$ 2.00
• The New User's Guide to EDITOR and RUNOFF			
Bound edition	15	FDR3104-101A	$15.00
Loose-leaf edition	15	FDR3104-101B	$15.00
Change sheet update	16	COR3104-001†	$ 3.00
• PRIMOS Subroutines Reference Guide	16	PDR3621†	$15.00
• LOAD and SEG Reference Guide	16	IDR3524†	$15.00
DATA MANAGEMENT			
• DBMS Administrator's Guide	16	PDR3276†	$15.00
• DBMS Schema Reference Guide	16	PDR3044†	$15.00
• DBMS FORTRAN Reference Guide	16	PDR3045†	$15.00
• DBMS COBOL Reference Guide	16	PDR3046†	$15.00
• The PRIME/POWER Guide	16	IDR3709	$15.00
• The MIDAS Reference Guide	14	IDR3061	$15.00
Technical update	16	PTU60†	$ 2.00
• The FORMS Programmer's Guide	16	PDR3040†	$15.00
STATISTICS			
• The SPSS Programmer's Guide	16	PDR3173†	$15.00
COMMUNICATIONS			
• The PRIMENET Guide	16	IDR3710†	$15.00
• The RJE/2780 Guide	16	PDR3067†	$15.00
• The HASP Guide	16	PDR3107†	$15.00
• The UT200 Guide	16	IDR3431†	$15.00
SYSTEM INSTALLATION			
• The System Installer's Guide	15	PDR3105†	$15.00

†-Denotes new or revised title

Part I

1 OVERVIEW OF PRIME'S FORTRAN

Introduction 1-1
    Figure 1-1. Sequence of FORTRAN program development 1-3
FORTRAN under PRIMOS 1-4
System resources supporting FORTRAN 1-5
    Table 1-1. FORTRAN mathematical functions 1-6
    Table 1-2. Matrix operations subroutines 1-6

2 OVERVIEW OF PRIMOS
Introduction 2-1
Glossary of Prime concepts and conventions 2-1
Command format conventions 2-4
Special terminal keys 2-5
System prompts 2-6
Using the file system 2-6
    Table 2-1. Types of files in PRIMOS 2-8
    Figure 2-1. Examples of files and directories in PRIMOS tree-structured file system 2-9

Part II

3 ACCESSING PRIMOS
Introduction 3-1
Accessing the system 3-2
Directory operations 3-2
System information 3-4
File operations 3-4
    Table 3-1. Useful system information 3-4
Completing a work session 3-8

4 ENTERING AND MANIPULATING SOURCE PROGRAMS
Entry from other media 4-1
Entering and modifying programs—the Editor 4-4
Listing programs 4-10
Renaming and deleting programs 4-11

5 COMPILING
Introduction 5-1
Using the compiler 5-1
End of compilation message 5-2
Compile error messages 5-2
Compiler parameters 5-3
    Table 5-1. Compiler parameter mnemonics 5-3
    Table 5-2. Concordance codes 5-9
Optimization 5-12

6 LOADING R-MODE PROGRAMS
Introduction 6-1
Using the loader under PRIMOS 6-1
Normal loading 6-2

Load maps 6-3
    Figure 6-1. Examples of load maps 6-5
Loading details 6-6
Command summary 6-9

7 LOADING SEGMENTED PROGRAMS

Introduction 7-1
Using SEG under PRIMOS 7-1
Normal loading 7-2
Load maps 7-3
    Figure 7-1. Example of load map 7-5
Advanced SEG features 7-7
Command summary 7-8
SEG-level commands 7-9
LOAD subprocessor commands 7-10
MODIFY subprocessor commands 7-13

8 EXECUTING PROGRAMS

Introduction 8-1
Execution of R-mode memory images 8-1
Executing segmented runfiles 8-2
Run-time error messages 8-2
Installation in the command UFD (CMDNC0) 8-4

9 DEBUGGING

Introduction 9-1
Coding strategy 9-1
Compiler usage 9-2

Part III

10 OPERATING SYSTEM FEATURES

Command file operations 10-1
Phantom users 10-8
Sequential job processor (CX) 10-11
Magnetic tape utilities 10-15
Using PRIMOS with networks 10-17
File copying, deleting, and listing (FUTIL) 10-18
    Figure 10-1. Overview of FUTIL commands 10-20
    Figure 10-2. FUTIL: COPYing, DELETing, and PROTECtion commands 10-21
    Figure 10-3. Typical tree structure 10-22
File manipulation 10-27
Setting terminal characteristics 10-30

11 EXTENDED SEGMENTED PROGRAM TECHNIQUES

Advanced features of LOAD subprocessor 11-1
The modification subprocessor 11-6
Shared code 11-7
COMMON blocks over 64K words long 11-12

12 INTERFACE TO OTHER SYSTEMS AND LANGUAGES

Introduction 12-1
Multiple Index Data Access System (MIDAS) 12-1
    Figure 12-1. User's functional overview of the MIDAS file system 12-3
    Figure 12-2. Sample of CREATK dialogue 12-4
Database Management System (DBMS) 12-6
Forms management system (FORMS) 12-6
    Figure 12-3. Example of data maintenance program 12-7
Other languages 12-8

13 OPTIMIZATION AND OTHER HELPFUL HINTS

Introduction 13-1
DO loops 13-1
Statement numbers 13-3
Multi-dimensioned arrays 13-3
Load sequence memory allocation 13-3
Function calls 13-4
V-mode vs. R-mode compilation 13-4
64V-mode COMMON 13-4
IF statements 13-5
Input/Output 13-5
Statement sequence 13-5
Parameter statements 13-6
Inefficient library calls 13-6
Statement functions and subroutines 13-6
Integer divides 13-6
Logical vs. arithmetic IF 13-6
Use of the compiler's -DYNM option 13-7
Conclusion 13-7
Request for contributions to this section 13-7

Part IV

14 FORTRAN LANGUAGE ELEMENTS

Legal character set 14-1
Line format 14-1
    Figure 14-1. Program line format 14-2
Operands 14-2
Generalized subscripts 14-5
Operators 14-6
Program composition 14-8
    Figure 14-2. Source program composition 14-8

15 FORTRAN STATEMENTS

Implemented statements 15-1
Header statements for subprograms 15-3
Specification statements 15-4
Storage statements 15-6
External procedure statements 15-7
Data definition statement 15-8
Compilation and run-time control statements 15-8
Assignment statements 15-9
Control statements 15-10
    Table 15-1. Data mode rules for assignment statements 15-11
Input/out statements 15-12
    Table 15-2. Devices and their default FORTRAN unit numbers 15-15

Coding statements 15–18
Format statements 15–19
    Table 15-3. Results of formats in output statements 15–20
    Table 15-4. Results of formats in input statements 15–22
    Table 15-5. Examples of B-format usage 15–24
Device control statements 15–25
Function calls 15–25
Subroutine calls 15–25

16 FORTRAN FUNCTION AND SUBROUTINE STRUCTURE

Functions 16–1
Subroutines 16–3

Part V

17 COMPILER REFERENCE

Prime FORTRAN compiler parameters 17–1
    Table 17-1. Compiler file specifications 17–2
Explicit setting of the A and B registers 17–6
    Figure 17-1. Bit-mnemonic correspondence (A and B registers) 17–7
    Table 17-2. A- and B-register bit correspondences of Parameter mnemonics 17–8
    Table 17-3. Bit/device correspondence 17–9

18 FORTRAN FUNCTION REFERENCE

FORTRAN function library 18–1

19 LIBRARIES REFERENCE

FORTRAN matrix (math) library 19–1
Sort and search library 19–7
Applications library 19–9
Operating system library 19–14

Appendices

A ERROR MESSAGES

Introduction A–1
Compiler error messages A–1
Loader error messages A–5
SEG loader error messages A–6
Run-time error messages A–8

B SYSTEM DEFAULTS AND CONSTANTS

C ASCII CHARACTER SET

Prime usage C–1
Keyboard input C–1
    Table C-1. ASCII character set (non-printing) C–2
    Table C-2. ASCII character set (printing) C–3

D PRIME MEMORY FORMATS OF FORTRAN DATA TYPES

Introduction D–1
Data types D–2

1 OVERVIEW

1

Overview of Prime's FORTRAN

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 Final Documentation Release, documenting Prime FORTRAN IV and supporting utilities at software revision level 16 (Rev. 16). It replaces the following documents:

> The FORTRAN programmer's Guide, PDR3057.
> Rev. 15 FORTRAN, PTU47.

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). An introductory section explains the basic concepts and features of the PRIMOS operating system (Section 2).

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 3)
- Creating a program (Section 4)
- Compiling (Section 5)
- Loading for relative address code (Section 6) or segmented-address code (Section 7)

- Executing (Section 8)
- Debugging concepts and the use of debugging tools (Section 9)

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 concepts for special programming needs, 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 10–13 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 14–16 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 other useful subroutines and where to find complete information about them.

Appendices A complete list of compiler, loader, and run-time error messages and their meanings (Appendix A); system defaults and constants (Appendix B); ASCII character set (Appendix C); and FORTRAN data type storage (Appendix D).

Related documents

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

Operating System Reference

Reference Guide, PRIMOS Commands

Reference Guide, PRIMOS Subroutines

Software Subsystem Reference

The FORTRAN Programmer's Companion

The New User's Guide to EDITOR and RUNOFF

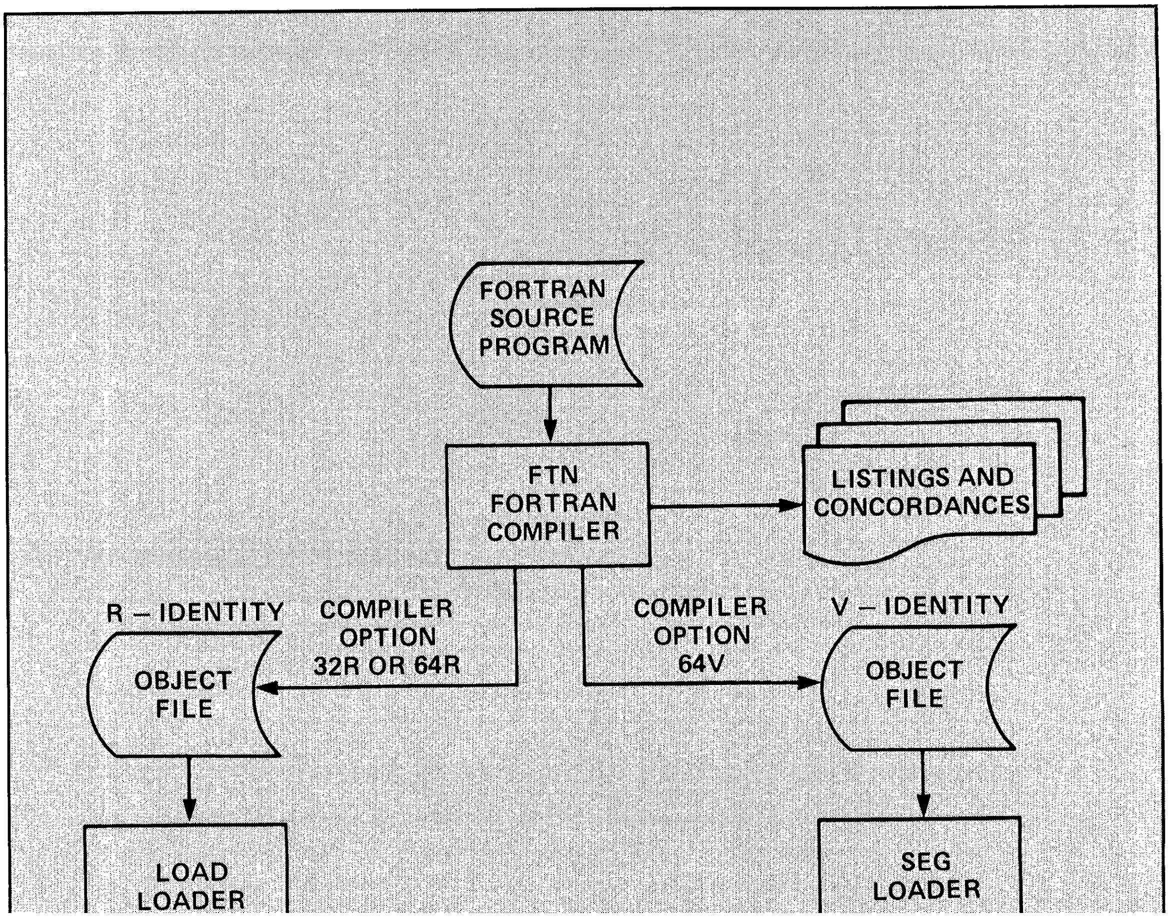
LOAD and SEG Reference Guide

Reference Guide, Multiple Index Data Access System (MIDAS)

Reference Guide for DBMS Schema DDL

FORTRAN Reference Guide for DBMS

FORMS Programmer's Guide


FORTRAN SOURCE PROGRAM
FTN FORTRAN COMPILER
LISTINGS AND CONCORDANCES
R – IDENTITY
COMPILER OPTION 32R OR 64R
COMPILER OPTION 64V
V – IDENTITY
OBJECT FILE
OBJECT FILE
LOAD LOADER
SEG LOADER

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, extensions, 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. Prime's FORTRAN supports both the ANSI and IBM formats for direct access READ and WRITE statements. 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.

Extensions: Extensions to standard FORTRAN which the user should inspect are:

- Use of the $INSERT command for file insertion at compilation
- B Format
- TRACE instruction for debugging
- List-directed input/output
- Direct file access READ/WRITE statements
- Long integers
- Parameters
- IMPLICIT specification
- Subprogram structure
- Generalized subscripting

Input/Output: FORTRAN logical unit numbers must agree with those given in Section 15 of this document (or such others as are established by the system administrator). 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 4) after which file operations may be performed.

Functions: Prime supplies a large number of the normal mathematical functions plus a set of Boolean (logical) functions. These are listed in Section 18. 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 16.

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 normal usage will be found in Section 19, especially in the Applications Library, which is given here in summary. Subroutines for extended usage or special cases will be found in Reference Guide, PRIMOS Subroutines. User-defined subroutines should be written to the specifications in Section 16.

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 19, 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 8). The terminal is dedicated to the program during execution. 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 and debugging.
- 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 10) 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. The sequential job processor queues requests for phantom users and then executes these jobs one at a time (Section 10).

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

File system summary

PRIMOS allows the user to access up to 16 files at one time. These disk files may be created, modified and deleted through the use of the Applications Library subroutines and the file management subroutines of the Operating System (Section 19). The file system is discussed in Section 2. Files, opened by these subroutines, may be accessed by FORTRAN I/O statements such as READ, WRITE, ENCODE, DECODE. See Section 15 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.

Table 1-1. FORTRAN Mathematical Functions

	Data Mode of Argument and Value			
Operation	**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 no. 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*) **MAX0**	**AMAX1** **MAX1**(*3*)	**DMAX1**	*n/a* *n/a*
Minimum of List	**AMIN0**(*2*) **MIN0**	**AMIN1** **MIN1**(*3*)	**DMIN1**	*n/a* *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.

Libraries

Library functions and subroutines of use to the FORTRAN applications programmer are in Section 19 of this document. A complete treatment of all library and system subroutines is in Reference Guide, PRIMOS Subroutines.

A summary of the FORTRAN mathematical functions is given in Table 1-1. There are also FORTRAN functions for the Boolean (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 mathematical usage, a matrix library is provided (See Table 1-2 for a summary). A complete description of the In-memory

Table 1-2. Matrix Operations Subroutines

	Data Mode of Matrix Elements			
Operation	**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

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 4; a complete description of the Editor is in The New User's Guide to EDITOR and RUNOFF.

Multiple index direct 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 12, the complete documentation is Reference Guide, Multiple Index Data Access Systems (MIDAS).

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 and FORTRAN Reference Guide for DBMS.

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 applications programmer writing data entry programs. Details are in FORMS Programmer's Guide.

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 and The COBOL Programmer's Guide.

2

Overview of PRIMOS

2

Overview of PRIMOS

INTRODUCTION

This section is an introduction to the basic concepts of Prime's Operating System (PRIMOS) and its embedded file management system (FMS). This information is basic to the efficient usage of PRIMOS. Contents include:

- A glossary of Prime concepts and terms.
- Command format conventions.
- Special terminal keys.
- System prompts.
- Using the file system.

GLOSSARY OF PRIME CONCEPTS AND CONVENTIONS

The following is a glossary of concepts and conventions basic to Prime computers, the PRIMOS operating system, and the file system.

abbreviation of PRIMOS commands: Only internal PRIMOS commands may be abbreviated.

binary file: A translation of source file generated by a language translator (PMA, COBOL, FTN, RPG). Such files are in the format required as input to the loaders. Also called **object file**.

byte: 8 bits; 1 ASCII character.

CPU: Central Processor Unit (the Prime computer proper as distinct from peripheral devices or main memory).

current directory: A temporary working directory explained in the discussion on **Home vs current directories** later in this section.

directory: A file directory; a special kind of file containing a list of files and/or other directories, along with information on their characteristics and location. MFDs, UFDs, and subdirectories (sub-UFDs) are all directories. (Also see **segment directory**.)

directory name: The file name of a directory.

external command: A PRIMOS command existing as a runfile in the command directory (CMDNC0). It is invoked by name, and executes in user address space. External commands print GO when starting, and *cannot* be abbreviated.

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**.

filename: A sequence of 32 or fewer characters which names a file or a directory. Within any directory, each filename is unique. **Directory names** and a filename may be combined into a **pathname**. Most commands accept a pathname wherever a filename is required.

Filenames may contain only the following characters:

A–Z, 0–9, _ # $ - . * &

The first character of a filename must not be numeric. On some devices underscore (_) prints as backarrow (←).

filename conventions: Prefixes indicate various types of files. These conventions are established by the compilers and loaders, or by common use, and not by PRIMOS itself.

B_ filename	Binary (Object) file
C_ filename	Command input file
L_ filename	Listing file
M_ filename	Load map file
O_ filename	Command output file
filename	Source file or text file
***filename**	SAVED (Executable) R-mode runfile
#filename	SAVED (Executable) V-mode runfile

file-unit: A number between 1 and 63 ('77) assigned as a pseudonym to each open file by PRIMOS. This number may be given in place of a filename in certain commands, such as CLOSE. PRIMOS-level internal commands require octal values. Each user may have up to 16 file units open at the same time. Certain commands or activities use particular unit numbers by default:

PRIMOS assigned units	Octal	Decimal
INPUT, SLIST	1	1
LISTING	2	2
BINARY	3	3
AVAIL	5	5
COMINPUT	6	6
SEG's loadmap	13	11
COMOUTPUT	77	63
EDITOR	1,2	1,2
SORT	1–4	1–4
RUNOFF	1–3	1–3

file protection keys: See **keys, file protection.**

home directory: The user's main working directory, initially the login directory. A different directory may be selected with the ATTACH command. See the discussion on **Home vs current directory** later in this section.

identity: The addressing mode plus its associated repertoire 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. R-identity and V-identity are also called R-mode and V-mode.

internal command: A command that executes in PRIMOS address space. Does not overwrite the user memory image. Internal commands *can* be abbreviated. See **abbreviation of PRIMOS commands.**

keys, file protection: Specify file protection, as in the PROTEC command.

0	No access
1	Read
2	Write
3	Read/Write
4	Delete and truncate
5	Delete, truncate and read
6	Delete, truncate and write
7	All rights

LDEV: Logical disk device number as printed by the command STATUS DISKS. (See ldisk.)

ldisk: A parameter to be replaced by the logical unit number (octal) of a disk volume. It is determined when the disk is brought up by a STARTUP or ADDISK command. Printed as LDEV by STATUS DISKS.

logical disk: A disk **volume** that has been assigned a logical disk number by the operator or during system startup.

MFD: The Master File Directory. A special directory that contains the names of the UFDs on a particular disk or partition. There is one MFD for each logical disk.

mode: An addressing scheme. The mode used determines the construction of the computer instructions by a compiler or assembler. (See **identity**.)

nodename: Name of system on a network; assigned when local PRIMOS system is built or

number representations:

xxxxx	Decimal
'xxxxx	Octal
$xxxxx	Hexadecimal

object file: See **binary file**

open: Active state of a file-unit. A command or program *opens* a file-unit in order to read or write it.

output stream: Output from the computer that would usually be printed at a terminal during command execution, but which is written to a file if COMOUTPUT command was given.

packname: See **volume-name.**

page: A block of 1024 16-bit words within a segment (512 words on Prime 300).

partition: A portion [or all] of a multihead disk pack. Each partition is treated by PRIMOS as a separate physical device. Partitions are an integral number of heads in size, offset an even number of heads from the first head. A **volume** occupies a partition, and a "partition of a disk" and a "volume of files" are actually the same thing.

pathname: A multi-part name which uniquely specifies a particular file (or directory) within a file system tree. A pathname (also called treename) gives a path from the disk volume, through directory and subdirectories, to a particular file or directory. See the discussion on **Pathnames** in this section.

PDEV: Physical disk unit number as printed by STATUS DISKS. (See **pdisk.**)

pdisk: A parameter to be replaced by a physical disk unit number. Needed only for operator commands.

phantom user: A process running independently of a terminal, under the control of a command file.

runfile: Executable version of a program, consisting of the loaded binary file, subroutines and library entries used by the program, COMMON areas, initial settings, etc. (Created using LOAD or SEG.)

SEG: Prime's segmentation utility.

segment: A 65,536-word block of address space.

segment directory: A special form of directory used in direct-access file operations. Not to be confused with **directory**, which means "file directory".

segno: Segment number.

source file: A file containing programming language statements in the format required by the appropriate compiler or assembler.

subdirectory: A directory that is in a UFD or another subdirectory.

sub-UFD: Same as **subdirectory.**

treename: A synonym for **pathname.**

UFD: A User File Directory, one of the Directories listed in the MFD of a **volume**. It may be used as a LOGIN name.

unit: See **file-unit.**

volume: A self-sufficient unit of disk storage, including an MFD, a disk record availability table, and associated files and directories. A volume may occupy a complete disk pack or be a **partition** within a multi-head disk pack.

volume-name: A sequence of 6 or fewer characters labeling a volume. The name is assigned during formatting (by MAKE). The STATUS DISKS command uses this name in its DISK column to identify the disk.

word: As a unit of address space, two bytes or 16 bits.

COMMAND FORMAT CONVENTIONS

The conventions for PRIMOS command documentation are:

WORDS-IN-UPPER-CASE: Capital letters identify command words or keywords. They are to be entered literally. If a portion of an upper-case word appears in rust, the rust colored letters indicate the minimum legal abbreviation.

Words-in-lower-case: Lower case letters identify parameters. The user substitutes an appropriate numerical or text value.

Braces { }: Braces indicate a choice of parameters and/or keywords. Unless the braces are enclosed by brackets, at least one choice must be selected.

Brackets [ ]: Brackets indicate that the word or parameter enclosed is optional.

Hyphen - : A hyphen identifies a command line option, as in: SPOOL -LIST

Parentheses ( ): When parentheses appear in a command format, they must be included literally.

Ellipsis . . . : The preceding parameter may be repeated.

Angle brackets < >: Used literally to separate the elements of a pathname. For example:

<FOREST>BEECH>BRANCH37>TWIG43>LEAF4.

option: The word **option** indicates one or more keywords or parameters can be given, and that a list of options for the particular command follows.

Spaces: Command words, arguments and parameters are separated in command lines by one or more **spaces**. In order to contain a literal space, a parameter must be enclosed in single quotes. For example, a pathname may contain a directory having a password:

‘<FOREST>BEECH SECRET>BRANCH6’.

The quotes ensure that the pathname is not interpreted as two items separated by a space.

Conventions in examples

In all examples, the user's input is **rust-colored**, and the system's output is not. For example:

```
OK, ATTACH GOUDY
OK, ED SEGINFO
GO
EDIT
```

User input usually may be either in lower case or in UPPER CASE. The rare exceptions will be specified in the commands where they occur.

SPECIAL TERMINAL KEYS

CONTROL: The key labeled CONTROL (or CTRL) changes the meaning of alphabetic keys. Holding down CONTROL while pressing an alphabetic key generates a control character. Control characters do not print. Some of them have special meanings to the computer. (See **CONTROL-P, CONTROL-Q** and **CONTROL-S**, below.) Others are ignored.

RUBOUT: The key labeled RUBOUT has a special use in RUNOFF. It is not generally meaningful to other standard Prime software. On some terminals it is labeled DELETE or DEL.

RETURN: The RETURN key ends a line. PRIMOS edits the line according to any erase (“) or kill (?) characters, and either processes the line as a PRIMOS command, or passes it to a utility such as the editor. RETURN is also called CR or CARRIAGE-RETURN.

BREAK, ATTN, INTRPT: See **CONTROL-P.**

Special terminal characters

Caret(ˆ): Used in EDITOR to enter octal numbers and for literal insertion of Erase and Kill characters. On some terminals and printers, prints as up-arrow (↑).

Backslash (\): Default EDITOR tab character.

Double-quote (“): Default erase character for PRIMOS, EDITOR, and RUNOFF Command Mode. Each double-quote erases a character from the current line. Erasure is from right (the most recent character) to left. Two double-quotes erase two characters, three erase three, and so forth. You cannot erase beyond the beginning of a line. The PRIMOS command TERM (Section 10 of this guide) allows the user to choose a different erase character.

Question mark (?): Default kill character for PRIMOS, EDITOR, and RUNOFF Command Mode. Each question mark deletes all previous characters on the line. The PRIMOS command TERM (Section 10 of this guide) allows the user to choose a different kill character.

CONTROL-P: QUIT *immediately* (interrupt/terminate) from execution of current command and return to PRIMOS level. Echoes as QUIT. Used to escape from undesired processes. Will leave used files open in certain circumstances. Equivalent to hitting BREAK key.

CONTROL-S: Halt output to terminal, for inspection. No commands other than CONTROL-P (QUIT) or CONTROL-Q (Continue) may be given. This special function is activated by the command TERM -XOFF.

CONTROL-Q: Continue output to terminal following a CONTROL-S (if TERM -XOFF is in effect).

UNDERSCORE (_): On some devices, prints as a backarrow (←).

SYSTEM PROMPTS

The OK prompt: The OK prompt indicates that the most recent command to PRIMOS has been successfully executed, and that PRIMOS is ready to accept another command from the user. The punctuation mark following the "OK" indicates to the user whether he is interfacing with a single-user level of PRIMOS. The prompt "OK:" indicates single-user PRIMOS (a version of PRIMOS II); the prompt "OK," indicates multi-user PRIMOS.

PRIMOS III and PRIMOS support **type-ahead.** The user need not wait for the "OK," after one command before beginning to type the next command. However, since each character echoes as the user types it, output from the previous command may appear on the terminal to be jumbled with the command being typed ahead. Type ahead is limited to 192 characters.

PRIMOS II *does not* support **type-ahead.** The user must wait for "OK:" before beginning to enter the next command.

The ER! prompt: The ER! prompt indicates that PRIMOS was unable to execute the most recent command, for one reason or another, and that PRIMOS is ready to accept another command from the user. The ER! prompt usually is preceded by one or more error messages indicating what PRIMOS thought the trouble was.

Common errors include:

- Typographical errors
- Omitting a password
- Being in the wrong directory
- Forgetting a parameter or argument

USING THE FILE SYSTEM

File and directory structures: A PRIMOS file is an organized collection of information identified by a filename. The file contents may represent a source program, an object program, a run-time memory image, a set of data, a program listing, text of an on-line document, or anything the user can define and express in the available symbols.

Files are normally stored on the disks attached to the computer system. No detailed knowledge of the physical location of a file is required because the user, through PRIMOS commands, refers to files by name. On some systems, files may also be stored on magnetic tape for backup or for archiving.

PRIMOS maintains a separate user file directory (UFD) for each user to avoid conflicts that might arise in assignment of filenames. A master file directory (MFD) is maintained by PRIMOS for each logical disk connected to the system. The MFD contains information about the location of each User File Directory (UFD) on the disk. In turn, each UFD contains information about the location and content of each file or sub-UFD in that directory.

The types of files most often encountered are shown in Table 2-1. For a description of the PRIMOS file system and a description of the ordering of information within files, refer to the Reference Guide, PRIMOS Subroutines.

Pathnames: The PRIMOS file directory system is arranged as a tree. At the root are the disk volumes (also called partitions, or logical disks). Each disk volume has a Master File Directory (MFD) containing the names of several User File Directories (UFDs). Each UFD may contain not only files, but subdirectories (sub-UFDs), and *they* may contain subdirectories as well. Directories may have subdirectories to any reasonable level.

A **pathname** (also called a **treename**) is a name used to specify uniquely any particular file or directory within PRIMOS. It consists of the names of the disk volume, the UFD, a chain of subdirectories, and the target file or directory. For example,

```
<FOREST>BEECH>BRANCH5>SQUIRREL
```

specifies a file on the disk volume FOREST, under the UFD BEECH and the sub-UFD BRANCH5. The file's name is SQUIRREL. Figure 2-1 illustrates how pathnames show paths through a tree of directories and files.

Disk volume names, and the associated logical disk numbers, may be found with the STATUS DISKS command, described later. A pathname can be made with the logical disk number, instead of the disk volume name. For example, if FOREST is mounted as logical disk 3,

```
<3>BEECH>BRANCH5>SQUIRREL
```

specifies the same file as the previous example.

Usually each UFD name is unique throughout all the logical disks. In our example that would mean that there would be only one UFD named BEECH in all the logical disks, 0 through 17. When that is the case, the volume or logical disk name may be omitted, and PRIMOS will search all the logical disks, starting from 0, until the UFD is found. For example, if there is no UFD named BEECH on disks 0, 1, or 2, then

```
BEECH>BRANCH5>SQUIRREL
```

will specify the same file as the previous two examples. This last form of pathname, in which the disk specifier is omitted, is called an **ordinary pathname** because it is very frequently used.

Pathnames vs filenames: Most commands accept a pathname to specify a file or a directory. So the terms "filename" and "pathname" may be used almost interchangeably. A few commands, however, require a filename, not a pathname. It is easy to tell a filename from a pathname. A pathname always contains a " > ", while a filename or directory name never does.

Home vs current directories: PRIMOS has the ability to remember two working directories for each user: the "home" directory, and the "current" directory. With few exceptions, the home and current directories are the same. All work can be accomplished while treating them both under the single concept of "working directory".

When the user logs in to a UFD, that UFD becomes the working directory. The ATTACH command changes the working directory to any other directory to which the user has access rights. A working directory may be an MFD, UFD, or sub-UFD.

The ATTACH command has a home-key option which allows the current directory to change while the home directory remains the same. See Reference Guide, PRIMOS Commands for details of this operation.

Relative pathnames: It is often more convenient to specify a file or directory pathname relative to the home directory, rather than via a UFD. For example, when the home directory is

```
BEECH>BRANCH5
```

the commands

```
OK, SLIST BEECH>BRANCH5>TWIG9>LEAF3
```

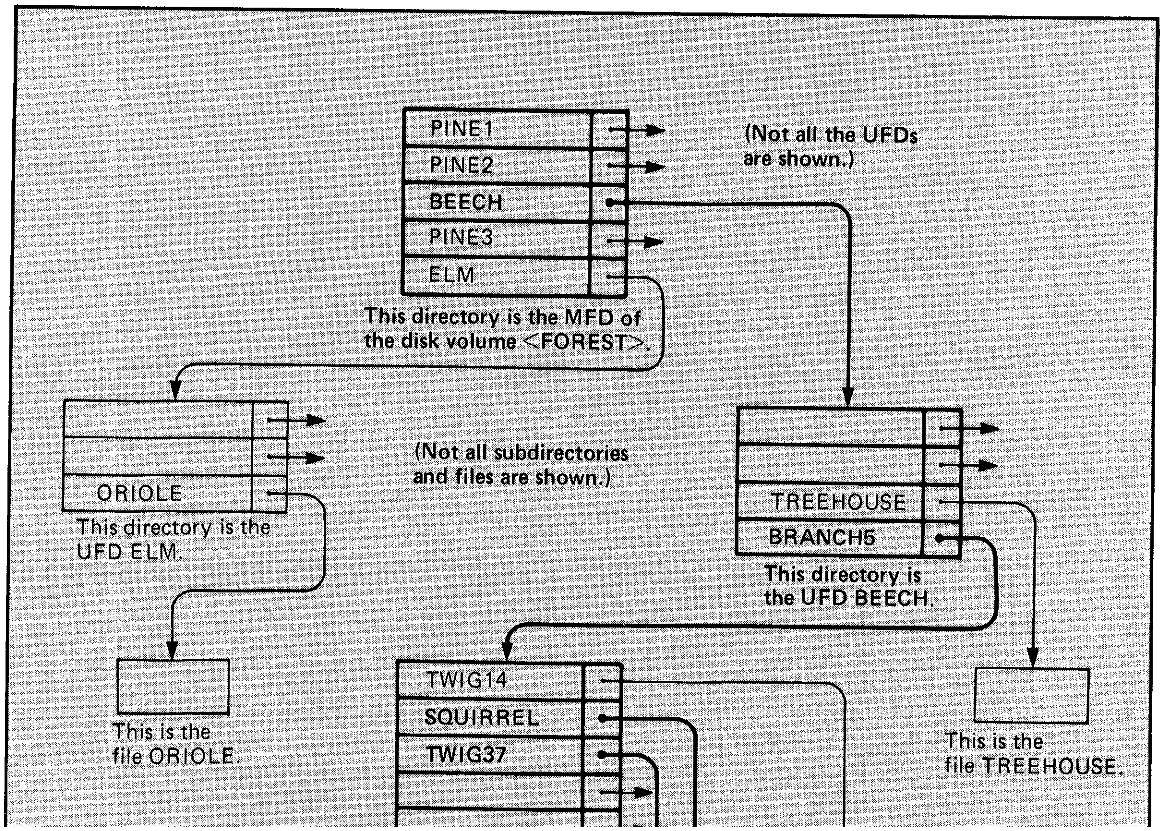
and

```
OK, SLIST *>TWIG9>LEAF3
```

have the same meaning. The symbol "*" as the first directory in a pathname means "home directory".

Table 2-1. Types of Files in PRIMOS

File Type	How Created	How Accessed	How Deleted	Use
ASCII, uncompressed	Programs **SORT** **COMOUTPUT**	Programs **ED** (examine only) **SLIST** **SPOOL** **FTN READ/WRITE**	**DELETE** **FUTIL DELETE**	Source files, text, data records for sequential access
ASCII, Compressed	**CMPRES** Some **COBOL** programs, **ED**	**EXPAND** to ASCII **SPOOLer** with **EXPAND** option **ED**	**DELETE** **FUTIL DELETE**	Same as compressed ASCII
Object (Binary)	Translators **FTN, PMA, COBOL,** Binary Editor (**EDB**)	**LOAD** or **SEG** Binary Editor (**EDB**)	**DELETE** **FUTIL DELETE** Binary Editor (**EDB**)	Input to **SEG** or **LOAD,** Libraries
Saved Memory Image	**LOAD** Applications programs	**TAP, PSD** Control panel	**DELETE** **FUTIL DELETE**	Runfiles
Segmented runfile	**SEG**	**SEG, VPSD** Control panel	**SEG DELETE** **FUTIL TREDEL**	Runfiles
Segmented data file	**SGDR$$** subroutine **MIDAS, DBMS**	**SGDR$$** subroutine **MIDAS** **DBMS**	**FUTIL TREDEL** **MIDAS KIDDEL**	Data records for direct access
UFD Sub-UFD	**CREATE**	Contents: **LISTF**	**DELETE** **FUTIL DELETE** **FUTIL TREDEL** **FUTIL UFDDEL**	Used by **PRIMOS**
MFD	**MAKE**	Contents: **LISTF**	*NO*	Used by **PRIMOS**
Disk record availability table **DSKRAT** file	**MAKE**	*NO*	*NO*	Used by **PRIMOS**
BOOT	**MAKE**	*NO*	*NO*	Used by **PRIMOS**
CMDNC0	**MAKE**	Contents: **LISTF**	*NO*	Used by **PRIMOS**


PINE1
PINE2
BEECH
PINE3
ELM
ORIOLE
TREEHOUSE
BRANCH5
TWIG14
SQUIRREL
TWIG37

(Not all the UFDs are shown.)

This directory is the MFD of the disk volume <FOREST>.

(Not all subdirectories and files are shown.)

This directory is the UFD ELM.

This directory is the UFD BEECH.

This is the file ORIOLE.

This is the file TREEHOUSE.

Current disk: Occasionally it will be necessary to specify a UFD on the disk volume you are currently using; this is, where your home directory is. For example, when developing a new disk volume with UFD names identical to those on another disk, it is necessary to carefully *specify* which disk is to be used, each time a pathname is given. The current disk is specified by

```
<*>BEECH>BRANCH5
```

for example. Do not confuse "<*>", meaning current disk, with the asterisk alone, which means home directory.

2

USING FORTRAN UNDER PRIMOS

3

Accessing PRIMOS

INTRODUCTION

Purpose of this section

This section is a brief overview of some of the fundamental features of the PRIMOS operating system for the FORTRAN programmer. It assumes that you are a FORTRAN programmer who has previous experience with an interactive computer system, although possibly not on a Prime computer. If you are not familiar with interactive computers, you may prefer to start with the New User's Guide to EDITOR and RUNOFF. This section also assumes you have read the concepts and definitions in Section 1.

Using the PRIMOS Programmer's Companion

In this section we introduce the essential PRIMOS commands so that you can begin working on the system. We recommend that you keep a Programmer's Companion handy as a summary of the commands explained in this section plus other PRIMOS commands. In this user's guide we have selected only those PRIMOS commands we know will be of use to the FORTRAN programmer. Depending upon your application, there are many other PRIMOS commands that may simplify your task or increase efficiency.

Using PRIMOS

PRIMOS recognizes more than one hundred commands, some of which invoke subsystems which themselves respond to subcommands or extensive dialogs. However, most FORTRAN users can do 99% of their program development using about a dozen commands. This section introduces the essential commands needed by all users. These commands allow you to:

- Gain admittance to the computer system (LOGIN).
- Change the working directory (ATTACH).
- Create new directories for work organization (CREATE).
- Secure directories against intrusion (PASSWD).
- Remove directories which are no longer needed (DELETE).
- Examine the location of the working directory and its contents (LISTF).
- Look at the availability and current usage of system resources— space, users, etc. (AVAIL, STATUS, USERS).
- Create files at the terminal or enter them from tape, etc. (MAGNET, CRSER, ED. See Section 4).
- Rename files (CNAME).
- Determine file size (SIZE).
- Examine files (SLIST).
- Print files (SPOOL).
- Remove unneeded files (DELETE).
- Allow controlled access to files (PROTEC).
- Complete a work session (LOGOUT).

ACCESSING THE SYSTEM

In order to access or work in the system, the user must first follow a procedure known as 'login'. 'Logging in' identifies the user to the system and establishes the initial contact between system and user (via a terminal). Once logged in, the user has access to a working directory (work area), to files and to other system resources. The format of the LOGIN command is:

LOGIN ufd-name [password] [-ON nodename]

ufd-name	The name of your login directory.
password	*Must* be included if the directory has a password.
-ON nodename	Used for remote login across PRIMENET network.

Example:

```
LOGIN DOUROS NIX
DOUROS (21) LOGGED IN AT 10'33 112878
```

The number in parentheses is the PRIMOS-assigned user number (also called 'job' number). The time is expressed in 24-hour format. The date is expressed as mmddyy (Month Day Year). The word NIX, in this example, is the password on the login directory.

When logging into the system, typing errors, incorrect passwords, or similar errors may cause error messages to be displayed. Most are self-explanatory. For a detailed discussion, see the New User's Guide to EDITOR and RUNOFF.

DIRECTORY OPERATIONS

Changing the working directory

After logging in, the user's working directory is set to the login UFD by PRIMOS. The user can move to another directory in the PRIMOS tree structure (i.e., attach) with the ATTACH command. The format is:

ATTACH new-directory

new-directory is the pathname of the new working directory.

Note

If any of the directories in the pathname have passwords, the entire pathname must be enclosed in single quotes, as in:

```
A 'BEECH SECRET>BRANCH5'
```

To set the MFD of a disk as the working directory, the format is slightly different:

ATTACH '<volume>MFD mfd-password'

volume is either the literal volume name or the logical disk number, and **mfd-password** is the password of the MFD. A password is always required for a MFD.

Recovering from errors while attaching: If an error message is returned following an ATTACH command (for example, if a UFD is not found), the user remains attached to the previous working directory.

Creating new directories

To organize tasks and work efficiently, it is often advantageous to create new sub-UFDs. These sub-UFDs can be created within UFDs or other sub-UFDs with the CREATE command. They can contain files and/or other sub-directories. The format is:

CREATE pathname

The **pathname** specifies the directory in which the sub-UFD is being created, as well as the name of the new directory.

Example:

```
CREATE <1>TOPS>MIDDLE>BOTTOM
```

The sub-UFD BOTTOM is created in the sub-UFD MIDDLE, which in turn is found in the UFD TOPS, which is in the MFD of disk volume 1.

Two files or sub-UFDs of the same name are not permitted in a directory. If this is inadvertently attempted, PRIMOS will return the message: ALREADY EXISTS.

Assigning directory passwords

Directories may be secured against unauthorized users by assigning passwords with the PASSWD command. There are two levels of passwords: owner and non-owner. If you give the owner password in an ATTACH command, you have owner status; if you give the non-owner password in an ATTACH command, you have non-owner status. Files can be given different access rights for owners and non-owners with the PROTEC command (see **Controlling file access**).

The PASSWD command replaces any existing password (s) on the working directory with one or two new passwords, or assigns passwords to this directory if there are none. The format is:

PASSWD owner-password [non-owner-password]

The **owner-password** is specified first; the **non-owner-password,** if given, follows. If a non-owner password is not specified, the default is null; then, any password (except the owner password) or none allows access to this directory as a non-owner.

Example:

```
OK, A DOUROS NIX
OK, PASSWD US THEM
```

The old password NIX is replaced by the owner password US, and the non-owner password THEM.

Deleting directories

When directories are no longer needed they may be removed from the system to provide more room for other uses. The DELETE command can also delete empty subdirectories from a given directory. The format is :

DELETE pathname

Sub-UFDs that are not empty, i.e., that still contain files or subdirectories, *cannot* be deleted with this command. All entries in the directory must be deleted first. If an attempt is made to delete directories containing files, PRIMOS prints the message:

DIRECTORY NOT EMPTY

Examining contents of a directory

After logging in or attaching to a directory, the user can examine the contents of this directory with the LISTF command which generates a list of the files and sub-directories in the current directory. The format is:

LISTF

For example, the working directory is called LAURA. The following list will be generated when LISTF is entered at the terminal:

```
OK, LISTF

UFD=LAURA  6 OWNER

$QUERY  BOILER  EX      LETTER  QUERY   OLISTF  BASICPROGS
OUTLINE $OUTLINE        MQL     $MQL    $LETTER MQL.LETTER
EXAMPLES        FUTIL.10        $FUTIL.10

OK,
```

The number following the UFD-name is the logical device number, in this case, 6. The words OWNER or NONOWN follow this number, indicating the user status in this directory. (See Securing Directories).

If no files are contained in a directory, .NULL. is printed instead of a list of files.

SYSTEM INFORMATION

Table 3-1 summarizes useful information you may need about the system and how to obtain it.

FILE OPERATIONS

Creating and modifying files

Text files may be created and modified using the text editor (ED). Files may be transferred from other systems using magnetic tape (MAGNET command), paper tape (ED command), or punched cards (CRSER command). These commands are described in Section 4.

Changing file names

It is often convenient or necessary to change the name of a file or a directory. This is done with the CNAME command. The format is:

CNAME old-name new-name

old-name is the pathname of the file to be renamed, and **new-name** is the new filename.

Example:

```
CN TOOLS>FORTRAN>TEST  OLDTEST
```

The file named TEST in the sub-UFD FORTRAN in the UFD TOOLS is changed to OLDTEST. Since no disk was specified all MFDs (starting with logical disk 0) are searched for the UFD TOOLS.

Table 3-1. Useful System Information

Item	Use	PRIMOS commands
Number of users	*Indicates system resource usage and expected performance.*	**STATUS USERS** (*user list*) **USERS** (*number of users*)
User login UFD	*Identifies user who spooled text file (printed on banner).*	**STATUS, STATUS UNITS**
User number		**STATUS**
User line number	*Changes terminal characteristics.*	**STATUS**
User physical device		**STATUS**
Open file units	*Avoids conflict when using files.*	**STATUS, STATUS UNITS**
Disks in operation		**STATUS, STATUS DISKS**
Assigned peripheral devices	*Tells if spool printer is working; if devices are available.*	**STATUS USERS**
User priorities		**STATUS USERS**
Other user numbers		**STATUS USERS**
Your phantom user number	*Logs out your phantoms.*	**STATUS USERS**
Network information	*Tells if network is*	**STATUS, STATUS NET**

Determining file size

The size (in decimal records) of a file is obtained with the SIZE command. This command returns the number of records in the file specified by the given pathname. The number of records in a file is defined as the total number of data words divided by 448. However, a zero-word length file always contains one record. The format is:

SIZE pathname

Example:

```
OK, SIZE GLOSSARY
GO
    14 RECORDS IN FILE

OK,
```

Examining file contents

Contents of a program or any text file can be examined at the terminal with the SLIST command. The format is:

SLIST pathname

The file specified by the given **pathname** is displayed at the terminal. It is possible to suspend the terminal display as it is printing. This procedure is explained in Section 10 (Terminal operations).

Obtaining copies of files

Printed copies of files from a line printer are obtained with the SPOOL command. It has several options, some of which will not apply to all systems, as systems may be configured differently. The format is:

SPOOL pathname

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

YOUR SPOOL FILE IS PRTxxx (length)

xxx is a 3-digit number which identifies the file in the Spool Queue List. The reason for a list, rather than just having each file spooled out as the request comes, is that some requests are very long—hundreds of pages. PRIMOS spools out the shorter files as soon as possible, rather than make the user wait while the long files are printed. The **length** (SHORT or LONG) which follows the SPOOL message is the category to which the file has been assigned. It is possible to check the status of a SPOOL request by giving the command:

SPOOL -LIST

Example:

```
OK, SPOOL $S2.3057
GO
YOUR SPOOL FILE IS PRT006  (LONG)  REV 15.2**

OK, SPOOL -LIST
GO

 USER    FILE    DATE/TIME    OPTS   SIZE  NAME          FORM   DEFER
------  ------  -----------  ------  ----  -----------  ------  -----
SOPHIE  PRT005  10/25 14:26  S          5  $UNFUNDED    W.WIBA
TEKMAN  PRT006  10/25 15:46  L         22  $S2.3057

OK,
```

To cancel a spool request, the command format is:

SPOOL -CANCEL PRTxxx

xxx is the number of your spool file.

For example:

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

OK,
```

Deferring printing: The -DEFER option tells the Spooler not to begin printing the indicated file until the system time matches the time specified with DEFER. This also permits you to enter SPOOL requests at your convenience, rather than waiting for the appropriate hour.

Specify the DEFER option by:

SPOOL filename -DEFER 'time'

The value for **'time'** can be expressed either in 24-hour format (00:00 = Midnight) or in 12-hour format followed by AM or PM (12:00 AM = Midnight). The format for 'time' is 'HH:MM', where HH is hours, ":" is any character, and MM is minutes. If you specify -DEFER but omit time you will get the prompt:

ENTER DEFERRED PRINT TIME:

If 'time' is not in the correct format, you will get the above prompt again, plus this informational message:

CORRECT FORMAT IS HH:MM AM/PM.

Printing on special forms: Line printers traditionally use one of two types of paper — "wide" listing paper, on which most program listings appear, and 8-½ × 11-inch white paper, which is standard for memos and documentation. Computer rooms often stock a variety of special paper forms for special purposes, such as 5-copy sets, pre-printed forms (checks, orders, invoices), or odd sizes or colors of paper.

Request a specific form by:

SPOOL filename -FORM form-name

form-name is any six-character (or less) combination of letters. A list of available form names should be obtained from the System Administrator.

Deleting files

When files or programs are no longer needed they may be removed from the system to provide more room for other uses. The DELETE command deletes files from the working directory. The format is:

DELETE pathname

Controlling file access

Assigning passwords to directories allows users working in a directory to be classified as owners or non-owners, depending upon which password they use with the ATTACH command. Controlled access can be established for any file using the PROTEC command. This command sets the protection keys for users with owner and non-owner status in the directory (see Assigning directory passwords above). The format is:

PROTEC pathname [owner-rights] [non-owner-rights]

pathname	The name of the file to be protected.
owner-rights	A key specifying owner's access rights to file *(original value = 7)*.
non-owner-rights	A key specifying the non-owner's access rights *(original value = 0)*.

The values and meanings of the access keys are:

key	**Rights**
0	No access of any kind allowed
1	Read only
2	Write only
3	Read and Write
4	Delete and truncate
5	Delete, truncate and read
6	Delete, truncate and write
7	All access

Note

The default protection keys associated with any newly created file or UFD are: 7 0. The owner is given ALL rights and the non-owner is given none.

Example:

```
PROTEC <OLD>MYUFD>SECRET 7 1
```

In this example, protection rights are set on the file SECRET in the UFD MYUFD so that all rights are given to the owner and only read rights are given to the non-owner.

COMPLETING A WORK SESSION

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

LOGOUT

PRIMOS acknowledges the command with the following message:

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

user-number	The number assigned at LOGIN.
terminal-time	The amount of elapsed clock time between LOGIN and LOGOUT in hours and minutes.
CPU-time	Central Processing Unit time consumed in minutes and seconds.
I/O-time	The amount of input/output time used in minutes and seconds.

It is a good practice to log out after every session. This closes all files and releases the PRIMOS process to another user. However, if you forget to log out, there is no serious harm done. The system will automatically log out an unused terminal after a time delay. This delay is set by the System Administrator (the default is 1000 minutes but most System Administrators will lower this value).

4

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 by using 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** (0-7)
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 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 deck-image

deck-image	The pathname of the file into which the card images are to be loaded.

Source deck header control cards are set up as follows:

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

Reading continues until a card with $E in columns 1 and 2 is encountered (end of deck); control returns to PRIMOS and the file is closed. If the cards are exhausted (or the reader is halted by the user), control returns to PRIMOS but the file is not closed. If more cards are to be read into the file, the reader should be reloaded; reading is resumed by the command START at the terminal.

The command:

CLOSE ALL

or

CLOSE deck-image

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

x is the tape drive unit number: 0-7.

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

```
OK, MAGNET
GO

MAGNET 15.2 15-JULY-78

OPTION: READ
```

MTU# = unit-number [/tracks]

unit-number	The number of the magnetic tape drive unit which was previously assigned.
tracks	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	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.
BCD	Translate to ASCII from 7-track tape.
BINARY	Transfer verbatim.
EBCDIC	Translate to ASCII.

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	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

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

Reading punched paper tape

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

```
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
EDIT                     Tape is being read
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 editor whose line pointer is always located at the last line processed (whether the processing action 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 character (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.

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.

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 edited 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 simple; the use of such 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 in conjunction with the TABSET command.

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 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 (FDR3104).

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

•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/[G] [n]

Replaces **string-1** with **string-2** for **n** lines. If **G** is omitted, only the first occurrence of string-1 on each line is changed; if G is present, all occurrences on n lines are changed.

•DELETE [n]

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

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 $\left\{\begin{array}{l}\text{(ASR)}\\ \text{(PTR)}\\ \text{(TTY)}\end{array}\right\}$

Reads text from the specified input device: **ASR** (Teletype paper-tape reader), **PTR** (high-speed paper tape reader) or **TTY** (terminal). Default is TTY.

•INSERT string

Inserts **string** after current line.

KILL character

Sets kill character to **character.**

LINESZ [n]

Changes maximum line length.

•LOAD filename

Loads **filename** into text following the current line.

•LOCATE string

Moves pointer forward to the 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** $\left\{\begin{array}{l}\textbf{PRINT}\\ \textbf{BLANK}\\ \textbf{SUPPRESS}\end{array}\right\}$

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 and EDIT modes.

MODE NPROMPT

Stops printing of INPUT and EDIT prompt characters (default).

MODIFY/string-2/string-1/[G] [n]

Superimposes **string-1** onto **string-2** 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 $\left\{\begin{matrix}\textbf{buffer-2}\\ \textbf{/string/}\end{matrix}\right\}$

Move **string** or contents of **buffer-2** into **buffer-1.**

•NEXT [n]

Moves the pointer **n** lines forward or backward (default n = 1).

NFIND string

Moves 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 $\left\{\begin{matrix}\textbf{(ASR)}\\ \textbf{(PTP)}\end{matrix}\right\}$ **[n]**

Punches **n** lines of 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 tab stops 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.

LISTING PROGRAMS

Terminal listing

Source programs may be listed at the terminal, by using the SLIST command described in Section 3.

Line printer listing

Use the SPOOL command (Section 3) to obtain a copy of a source file on the system line printer. Additional options of use to the FORTRAN programmer 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.

The -FTN and -LNUM options are incompatible.

RENAMING AND DELETING PROGRAMS

Renaming

Programs may be renamed with the PRIMOS command CNAME (Section 3). You must have owner status in the UFD in order to use this command.

Deleting

Programs may be deleted with the PRIMOS command DELETE (Section 3). You must have owner status in order to use this command.

5

Compiling

INTRODUCTION

Prime's FORTRAN 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 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 6; V-identity code is loaded with Prime's segmented-addressing utility (SEG), described in Section 7. Segmented-addressing code can be executed on Prime 350 (or higher) computers.

USING THE COMPILER

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

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

or

FTN [-parameter-1] -I pathname . . . [-parameter-n]

pathname	The pathname of the FORTRAN source program file.
parameter-1, etc.	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 pathname (alternatively, -S pathname) but not both.

Examples:

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

and

```
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	The number of compilation errors; 0000 indicates a successful compilation.
yyyyyy	Program module identification: .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	The PRIMOS revision number.

Example:

```
0000 ERRORS [<.MAIN.>FTN-REV16.0]
```

indicates the successful compilation of a main FORTRAN program by the REV.16 Compiler.

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

COMPILE ERROR MESSAGES

The general format of the error message is:

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

nnnn	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	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 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 files. 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 A-and B-register values explicitly) is still valid. This means existing command files which set the A- and B- registers need not be changed. (See Section 17).

Compiler functions

The compiler functions enabled by the mnemonic parameters may be considered to fall into four groups (Table 5-1).

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

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

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 pathname	Define input file/device. (alternatively -SOURCE) (example: -I TEST or -S TEST).
-I pathname	The source program filename is **pathname**.
-BINARY	To override default, define binary (object) file/device.
-B pathname	The binary file will be created with the **pathname** 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.

Table 5-1. 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
PBECB	Loads ECBs (Entry Control Blocks) into the procedure frame *(64V subroutines only)*
• **SAVE**	Static allocation of local storage
• **32R**	32K words of relative-addressed user space
64R	64K words of relative-addressed user space
64V	Up to 256 × 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

If the BINARY parameter is not included in the

-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 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 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-REV16.0]
0003 ERRORS [<.MAIN.>FTN-REV16.0]
```

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-REV16.0]
0003 ERRORS [<.MAIN.>FTN-REV16.0]
```

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-I
(0004)        I=3
(0005)  20    GO TO (100,310,320), I
        000001:   FLD   =24576
        000003:   FST   X
        000005:   LDA   I
        000006:   MPY   =5
        000007:   PIM
        000010:   JST   C$12
        000011:   FST   B
        000013:   LDA   =5
        000014:   SUB   I
        000015:   JST   C$12
        000016:   FST   C
        000020:   LDA   =3
        000021:   STA   I
(0006)  320   A = B+C
        000022:   LDA   I
        000023:   JST   F$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
        000045:   OCT   000000
        000046:   LINK  C
        000046:   OCT   000000
        000047:   OCT   000000
        000050:   LINK  I
        000050:   OCT   000000
        000051:   LINK  X
        000051:   OCT   000000
        000052:   OCT   000000
        000053:   LINK  =3
        000053:   OCT   000003
        000054:   LINK  =5
        000054:   OCT   000005
        000055:   LINK  =24576
        000055:   OCT   060000
        000056:   OCT   000206
        000041:   DAC    100
```

```
**** LINE 0011  [ END ]  _110 - UNDEFINED STMT NO.
       000022:   DAC    _20
       000001:   DAC    _310
       000030:   DAC    _320
0003 ERRORS [<.MAIN.>FTN-REV15.1]
0003 ERRORS [<.MAIN.>FTN-REV15.1]
```

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-REV15.1]
0003 ERRORS [<.MAIN.>FTN-REV15.1]
```

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 5-2.

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 8), 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-REV15.1]

OK, LOAD
GO
$ LO B_PRIME
$ LI
LOAD COMPLETE
$ SA *PRIME
$ EX
FOLLOWING IS A LIST OF PRIME NUMBERS FROM 2 TO 50
                   2
                   3
                   5
                   7
 K=      3
 (2)
                  11
 (4)
 K=      3
 (2)
                  13
 (4)
 K=      3
 (4)
 K=      4
 (2)
                  17
 (4)
 K=      4
 (2)
                  19
 (4)
 K=      4
 (4)
 K=      4
 (2)
                  23
 (4)
 K=      5
 (2)
 (4)
 K=      5
 (4)
 K=      5
 (2)
 (2)
                  29
 (4)
 K=      5
 (2)
 (2)
                  31
 (4)
 K=      5
 (4)
 K=      5
 (2)
 (4)
 K=      6
 (2)
 (2)
                  37
 (4)
 K=      6
 (4)
 K=      6
 (2)
 (2)
                  41
 (4)
 K=     6
 (2)
 (2)
                   43
 (4)
 K=     6
 (4)
 K=     6
 (2)
 (2)
                   47
 (4)
 K=     7
 (2)
 (2)
 (4)
THIS IS THE END OF THE LIST

****ST
```

Table 5-2. 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

Code	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.

Memory usage

32R/64R/64V: 32R mode is the default. The complier 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 256 × 64K words (segmented-address) respectively.

NOBIG/BIG: BIG treats all dummy arrays as arrays that span segment boundaries. BIG forces the 64V mode and thus *cannot* be used in the 32R or 64R modes. If a dummy argument array may become associated with an array spanning a segment boundary (through a subroutine call or function reference) the compiler must be made aware of this by including BIG in the parameter list. The code generated will work whether or not the array actually spans segment boundary.

NOBIG is the default parameter (see Section 11 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 6 for explanation of base areas.)

PBECB: Generates code to load ECBs (Entry Control Blocks) into procedure frame, allowing ECBs to be shared *(64V subroutines only)*.

Operations

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

FP/NOFP: Suppresses 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 the CPU does not have the floating-point hardware, suppressing these instructions will speed up execution.

INTS/INTL: The Prime FORTRAN system has both Long (INTEGER*4) and Short (INTEGER*4) 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 17).

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
NOXREF	XREFL or XREFS
PBECB	32R or 64R
SAVE	DYNM
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.

OPTIMIZATION

An extended version of the FORTRAN compiler optimizes DO loops. It is invoked by entering:

FTNOPT source-file [options]

source-file is the pathname of the source program to be compiled. **options** are identical to those of the FTN compiler with two additional options. These new options are:

-OPT	Optimizes all DO loops that *do not* contain GO TO statements.
-UNCOPT	Unconditionally optimizes *all* DO loops.

If any DO loop in the program has an extended range, the -UNCOPT option should not be used; use the -OPT option instead. The optimized object program will be longer than the non-optimized version but it will execute faster.

6

Loading R-mode programs

INTRODUCTION

The PRIMOS LOAD utility converts object modules (such as those generated by the FORTRAN compiler) into runfiles that execute in the 32R or 64R addressing modes. (Runfiles to execute in the 64V mode *must* be loaded using the segmentation utility, SEG.)

The following description emphasizes the loader commands and functions that are of most use to the FORTRAN programmer. For a complete description of all loader commands, including those for advanced system-level programming, refer to Reference Guide, LOAD and SEG.

USING THE LOADER UNDER PRIMOS

The PRIMOS command:

```
LOAD
```

transfers control to the R-mode loader, which prints a $ prompt character and awaits a loader subcommand. After executing a command successfully, the loader repeats the $ prompt character.

If an error occurs during an operation, the Loader prints an error message, then the $ prompt character. Loader error messages and suggested handling techniques are discussed elsewhere in this section 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 IN USE, ILLEGAL NAME, NO RIGHT, etc.) is encountered, the loader prints this system error and returns its prompt symbol, $.

The loader remains in control until a QUIT or PAUSE subcommand returns control to PRIMOS, or an EXECUTE subcommand starts execution of the loaded program.

Load subcommands can be used in command files, but comment lines result in a CM (command error) message.

NORMAL LOADING

Loading is normally a simple operation with only a few straightforward commands needed. The loader also has many additional features to optimize runfile size or speed, perform difficult loads, and deal with possible complications. The most frequently used load commands and operations are presented first; this enables immediate use of the loader. Advanced features are then described followed by a summary of all loader commands.

The following commands (shown in abbreviated form) accomplish most loading functions:

PRIMOS-Level Commands:

FILMEM	Initializes user space in preparation for load.
LOAD	Invokes loader for entry of subcommands.
RESUME	Starts execution of a loaded, SAVEd runfile.

LOAD Subcommands

MODE option	Sets runfile addressing mode as D32R (default) or D64R.
LOAD pathname	Loads specified object file.
LIBRARY [filename]	Loads library object files from UFD LIB. (Default is FTNLIB.)
MAP [option]	Prints loadmap. Option 3 shows unresolved references.
INITIALIZE	Returns loader to starting condition in case of command errors or faulty load.
SAVE pathname	Saves loaded memory image as runfile.
QUIT or **PAUSE**	Return to PRIMOS.

Most loads can be accomplished by the following basic procedure:

1. Use the PRIMOS FILMEM command to initialize memory.
2. Invoke LOAD.
3. Use the MODE command to set the addressing mode, if necessary. (The default is 32R mode.)
4. Use loader's LOAD subcommand to load the object file (B_filename) and any separately compiled subroutines.
5 Use loader's LIBRARY subcommand to load subroutines called from libraries (the default is FTNLIB in the UFD = LIB). Other libraries, such as SRTLIB or APPLIB, must be named explicitly.
6. If you do not have a LOAD COMPLETE, do a MAP 3 to identify the unsatisfied references, and load them.
7. SAVE the runfile under an appropriate name.

If these commands produce a LOAD COMPLETE message, then loading was accomplished. If there is a problem, it will become apparent by the absence of a LOAD COMPLETE message or some other loader error message. (See Appendix A for a complete list of all loader error messages and their probable cause and correction.)

After a successful load, you can either start runfile execution from LOAD command level, or quit from the loader and start execution through the PRIMOS RESUME command. An example of such a load is:

```
OK, LOAD
GO
$ MO D64R
$ DC
$ LO B_ARRAY
$ LI
$ SA *ARRAY
$ MA M_ARRAY
$ QU

OK,
```

Order of loading

The following loading order is recommended:

1. Main program
2. Separately compiled user-generated subroutines (preferably in order of frequency of use).
3. Other Prime libraries (LI filename)
4. Standard FORTRAN library (LI)

Loading library subroutines

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library file FTNLIB in the LIB UFD. The appropriate subroutines from this file are loaded by the LIBRARY command given without a filename argument. If subroutines from other libraries are used, such as MATHLB, SRTLIB, or APPLIB, additional LIBRARY commands are required which include the desired library as an argument.

LOAD MAPS

During loading, the loader collects information about the results of the load process, which can be printed at the terminal (or written to a file) by the MAP command:

MAP [pathname] [option]

The information in the map can be consulted to diagnose problems in loading, or to optimize placement of modules, linkage areas and COMMON in complex loads.

Load information is printed in four sections, as shown in Figure 6-1. The amount of information printed is controlled by MAP option codes, such as:

Option	Load Map Information
None, 0 or 4	Load state, base area, and symbol storage; symbols sorted by address
1	Load state only
2	Load state and base areas
3	Unsatisfied references only
6	Undefined symbols, sorted in alphabetical order
7	All symbols, sorted in alphabetic order

Load State

The load state area shows where the program has been loaded, the start-of-execution location, the area occupied by COMMON, the size of the symbol table, and the UII status. All locations are octal numbers.

***START:** The location at which execution of the loaded program will begin. The default for FORTRAN programs is '1000.

***LOW:** The lowest memory image location occupied by the program. Executable code normally starts at '1000, but sector 0 address links (if any) begin at '200.

***HIGH:** The highest memory image location occupied by the program (excluding any area reserved for COMMON).

***PBRK:** "Program Break": The next available location for loading. It normally is the location following the last loaded module, but can be moved by PBRK or the LOAD family of commands.

***CMLOW:** The low end of COMMON.

***CMHGH:** The top of COMMON.

***SYM:** The number of symbols in the loader's symbol table. This is usually of no concern unless the symbol space crowds out the last remaining runfile buffer area. (There is room for about 4000 symbols before this is a risk.)

***UII:** A code representing the hardware required to execute the instructions in loaded modules. Codes and other information are described later in ths section.

Base areas

The base area map includes the lowest, highest and next available locations for all defined base areas. 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

```
*START  001000    *LOW    000200    *HIGH   007775    *PBRK   106376
*CMLOW  077777    *CMHGH  077777    *SYM    000070    *UII    000001

*BASE   000200  000225  000777  000777
*BASE   001534  001600  001605  001605
*BASE   002576  002660  002661  002661
*BASE   003624  003663  003665  003665
*BASE   004664  004706  004707  004707

**GHOST   001025    F$WA     001031    F$WX     001037    F$IO     001113
  F$A1    001606    F$A3     001606    F$A5     001613    F$A2     001621
  F$A6    001627    F$A7     001645    F$CB     002326    F$IOBF   005405
  WRASC   005507    IOCS$    005514    IOCS$T   005613    WATBL    005625
  LUTBL   005644    PUTBL    005701    RSTBL    005736    O$AD07   005773
  O$AD08  006136    O$AA01   006200    PRSPE$   006235    OERRT$   006427
  ERRPR$  007433    PRWF$$   007436    WTLIN$   007441    ERRSET   007444
  F$ER    007447    F$HT     007454    EXIT     007534    AC1      007537
  AC2     007540    AC3      007541    AC4      007542    AC5      007543
  TNOU    007544    TONL     007634    T1OU     007641    T1IB     007661
  T1OB    007666    F$AT     007673    F$AT1    007675    GCHAR    007740
  SCHAR   007755

COMMON BLOCKS

  LIST    000001                      007776  076400
```

Symbol storage

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).
Symbols are listed by ascending address (default) or in alphabetical order (MA 6 or MA 7). The list may be restricted to unsatisfied references only (MA 3 or MA 6).

COMMON blocks

The low end and size of each COMMON area are listed, along with the name (if any). Every map includes a reference to the special COMMON block LIST, defined as starting at location 1.

LOADING DETAILS

When standard loading goes well, the user can ignore most of the loader's advanced features. However, situations can arise where some detailed knowledge of the loader's tasks can optimize size or performance of a runfile, or even make a critical load possible. From that viewpoint, the main tasks of the loader are:

- Convert block-format object code into a run-time version of the program (executable machine instructions, binary data and data blocks).
- Resolve address linkages (translate symbolic names of variables, subroutine entry points, data items, etc. into appropriate binary address values).
- Perform address resolution (discussed later).
- Detect and flag errors such as unresolved external references, memory overflow, etc.
- Build (and, on request, print) a load map. The map may also be written to a file.
- Reserve COMMON areas as specified by object modules.
- Keep track of runfile's hardware execution requirements and make user aware of need to load subroutines from UII library.

Virtual loading

The loader occupies the upper 32K words of the user's 64K-word virtual address space. Programs up to 32K words are loaded directly into the memory locations from which they execute. Programs loaded in this manner can be started by the loader's EXECUTE command without being saved. For larger 64R-mode programs, the loader uses the available memory as buffer space and transfers loaded pages of memory to a temporary file that accomodates a full 64K-word memory image. When loading is complete, the file must be assigned a name by the loader's SAVE command; it can then be executed either through the loader's EXECUTE command or the PRIMOS RESUME command.

The loader remains attached to the working directory throughout loading, for access to the temporary file. Files in other directories can be loaded by giving a pathname in a LOAD command.

Use of pathnames

Pathnames can be used to specify object files in all commands except LIBRARY, which accepts only a simple filename of a file within the LIB UFD.

Object code

Inputs to the loader are in the form of object code—a symbolic, block-format file generated by all of Prime's language translators. Prime's standard library files consist of subroutines in this format.

The loader combines the user's main program object file with the object files of all referenced subroutines (either those in the library, or those generated and separately compiled by the user) into a single runfile. The runfile is then ready for execution, either directly through the loader's EXECUTE command or through the PRIMOS RESUME command.

Runfiles

A runfile consists of a header block followed by the runfile text in memory image format. The header contains information that enables the runfile to be brought into memory by the PRIMOS RESTORE or RESUME command. Contents of the header can be examined after a RESTORE by the PM command. (See Reference Guide, PRIMOS Commands.)

Selecting the addressing mode

The 32R addressing mode is retained as the loader's default for compatibility with existing command files. The only significant difference between 32R and 64R for small programs is that 32R permits multiple indirect links, while 64R allows only one level of indirection. In certain situations such as processing of multi-dimensional arrays, 32R mode may enable the compiler to produce a runfile that is somewhat more compact or runs slightly faster. However, for programs that approach the 32K word boundary, 64R mode ensures successful loading with no significant penalties of size or speed. Thus MODE D64R is recommended for most applications.

Base areas

"Base Area" is an assembly language concept that can be disregarded by the FORTRAN programmer except when one of the following is printed:

BASE SECTOR 0 FULL

symbolname XXXXXX NEED SECTOR 0 LINK

This condition, usually encountered only when loading large programs, can be avoided in several ways:

- Give the AUTOMATIC command to enable the loader to assign local linkage areas before and after individual subroutines.
- Use setbase parameters with a LOAD or LIBRARY command to insert local linkage areas where they are needed.
- Use the SETBASE command to designate a base area where it is required.
- During compilation, use the -DEBASE option.

Locating COMMON

By default, the loader sets the high end of FORTRAN COMMON at '077777 (the 32K word boundary) and allocates it downward from there. If a PROGRAM-COMMON OVERLAP message occurs, COMMON can be moved higher by the COMMON or DC (Defer Common) subcommands. DC is recommended. (If DC is used, a LOAD COMPLETE message will not occur until a SAVE or EXECUTE command is given.)

UII handling

The loader can keep track of the CPU hardware required to execute the instructions generated by the modules already loaded. This is shown in the UII entry in the load-state section of a load map. The codes are:

UII Value	CPU Required
100	Prime 450 and up
57	Prime 350 or 400
17	Prime 300 with FP Hardware
3	Prime 300
1	Prime 100 with HSA or 200 with HSA
0	Prime 100 or 200

If the UII code on the load map is *greater* than the value for the target CPU, then it will be necessary to load part of the UII library to make execution possible. When a CPU encounters an instruction not implemented by hardware, a UII (Unimplemented Instruction Interrupt) occurs and control is transferred to the appropriate UII routine. This routine simulates the missing hardware with software routines.

However, the UII routine must be loaded by the command LI UII, which should be the last LOAD command before the program is saved. The appropriate routines will be selected from this library to satisfy the additional hardware requirements of the program.

To make sure that only the required subroutines are loaded, the user can "subtract" hardware features that are present in the CPU by entering a HARDWARE command. For example, assume:

- A load session produces a load map UII value of 57.
- The target CPU is a Prime 300 with floating point (UII value 17).

The command:

```
HA 17
```

reduces the load state UII value to 40 (i.e., '57-'17) and ensures that the floating point subroutines do not occupy space in the runfile.

If, after a HARDWARE command, the load state UII value is 0, the UII library need not be loaded.

System programming features

The following commands are primarily of interest to assembly language and systems programmers. They are described in more detail in Reference Guide, LOAD and SEG:

F/	Prefix to LOAD and LIBRARY which forceloads unreferenced modules.
P/	Prefix to LOAD and LIBRARY which starts loading on next page boundary. (Can reduce paging time.)
PBRK	Program Break. Resume loading at a new location.
CH,SS,SY,XP	Symbol control commands.
EN	ENtire save; saves copy of load session for building of program overlays.
ER	Controls action taken by loader following errors.
SZ	Controls use of Sector 0.

COMMAND SUMMARY

Following is a summary of all LOAD commands, in alphabetical order. All file and directory names may be specified by pathnames, except in the LIBRARY command. All numerical values must be octal.

ATTACH [pathname]

Attaches to specified directory.

AUTOMATIC base-length

Inserts base area of specified **length** at end of routine if >'300 locations loaded since last base area.

CHECK [symbol-name] [offset-1]. . .[offset-9]

Checks value of current PBRK against symbol or number. **symbol-name** is a 6 character symbol defined in the symbol table. **offset-1** through **9** are summed to form an address or offset from symbol name. Numbers preceded by "–" are negative.

COMMON address

Moves top/starting COMMON location to **address.**

DC [END]

Defers definition of COMMON block until SAVE command is given. (Low end of COMMON follows top of load.) **[END]** turns off DC.

ENTIRE pathname

Saves entire state of loader as runfile, along with temporary file, for building overlays.

ERROR n

Determines action taken in case of load errors.

n	**Meaning**
0	SZ errors treated as multiple indirect, others act as n=1.
1	Display multiple indirects on TTY but continue LOAD; abort load of file for all other errors.
2	Abort to PRIMOS

EXECUTE [a] [b] [x]

Starts execution with specified register values.

F/ $\left\{\begin{matrix}\text{FORCELOAD}\\ \text{LIBRARY}\\ \text{LOAD}\end{matrix}\right\}$ [pathname] [parameters]

Forceloads all modules in specified object file. See LOAD for parameters.

HARDWARE definition

Specifies expected level of instruction execution.

CPU	Definition
P500	100
P350,P400	57
P300/FP	17 FP = Floating Point
P300	3
P200/HSA	1 HSA = High-speed arithmetic
P100/HSA	1
P200	0
P100	0

HARDWARE, if given, must precede loading of UII library.

INITIALIZE [pathname] [parameters]

Initializes LOADER and, optionally, does a LOAD. See LOAD for parameters.

LIBRARY [filename] [loadpoint]

Attaches to LIB=UFD, loads specified library file (FTNLIB is default), and re-attaches to home directory.

LOAD [pathname] [parameters]

Loads the specified object module. The parameters may be entered in three formats:

1. **loadpoint [setbase-1]. . .[setbase-8]**
2. *** [setbase-1]. . .[setbase-9]**
3. **symbol [setbase-1]. . .[setbase-9]**

In form 1, **loadpoint** is the starting location of the load. In form 2, the load starts at the current PBRK location (*). In form 3, the load address can be stated symbolically (**symbol**). The remaining numeric parameters (**setbase-1,** etc.) specify the size of linkage areas to be inserted before and after modules during loading. If the last parameter is '177777, the loader requests more setbase values.

MAP [pathname] [option]

Generates load-state map on terminal, or in a file, if **pathname** is specified.

Option	Meaning
0	Load state, base area, symbol storage map; symbols sorted by address (default)
1	Load state only
2	Load state and base area
3	Unsatisfied references only
4	Same as 0
5	System Programmer map
6	Undefined symbols sorted alphabetically
7	All symbols sorted alphabetically
10	Special symbol map for PSD (in a file)

$\textbf{MODE} \left\{\begin{array}{l}\textbf{D32R}\\ \textbf{D64R}\\ \textbf{D16S}\\ \textbf{D32S}\end{array}\right\}$

Specifies address resolution mode for next load module (32K Relative, D32R, is default). If used, MODE *must* precede other LOAD commands.

$\textbf{P/} \left\{\begin{array}{l}\textbf{FORCELOAD}\\ \textbf{LIBRARY}\\ \textbf{LOAD}\end{array}\right\} \textbf{[pathname] [parameters]}$

Begins loading at next page boundary. See LOAD for **parameters**.

PAUSE

Leaves loader to execute internal PRIMOS command. Return via START.

$\textbf{PBRK} \left\{\begin{array}{ll}\textbf{[symbol-name]} & \textbf{[offset-1]...[offset-9]}\\ \textbf{* offset-1} & \textbf{[offset-2]...[offset-9]}\end{array}\right\}$

Sets a program break to value of **symbol** plus offset or a number. * treats sum of numbers as offset from current PBRK. Offsets may be negative.

QUIT

Deletes temporary file, closes map file (if loader opened it), and returns to PRIMOS.

SAVE pathname

Writes a memory image of the loaded runfile to the disk.

$\textbf{SETBASE} \left\{\begin{array}{ll}\textbf{[base-start]} & \textbf{[base-range]}\\ \textbf{*} & \textbf{base-range}\end{array}\right\}$

Defines starting location and size of base area. * is current value of PBRK.

SS symbol-name

Save symbol. Exempts specified **symbol** from action of XPUNGE.

$\textbf{SYMBOL} \left\{\begin{array}{ll}\textbf{symbol-name} & \textbf{[offset-1]...[offset-6]}\\ \textbf{* offset-1} & \textbf{[offset-2]...[offset-6]}\end{array}\right\}$

Establishes locations in memory map for common blocks, relocation load points, or to satisfy references. * is current value of PBRK. **Offsets** are summed and may be negative.

$\textbf{SZ} \left\{\begin{array}{l}\textbf{YES}\\ \textbf{NO}\end{array}\right\}$

Permits/prohibits links in sector zero.

VIRTUALBASE base-start to-sector

Copies base sector to corresponding locations in **to-sector**. Used for building RTOS modules.

XPUNGE dsymbols dbase

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

7

Loading segmented programs

INTRODUCTION

The PRIMOS SEG utility converts object modules (such as those generated by the FORTRAN Compiler) into segmented runfiles that execute in the 64V addressing mode and take full advantage of the architecture and instruction set of the Prime 350 and up. Segmented runfiles offer the following advantages:

- Much larger programs: up to 256 segments per user program (32 Megabytes)
- Access to V-mode instructions and architecture (Prime 350 and up) for faster execution.
- Ability to install shared code: single copy of a procedure can service many users, significantly reducing paging time.
- Reentrant procedures permitted: procedure and data segments can be kept separate.

The following description emphasizes the commands and functions that are of most use to the FORTRAN programmer. Extended features are described in Section 11. For a complete description of all SEG commands, including those for advanced system-level programming, refer to Reference Guide, LOAD and SEG.

USING SEG UNDER PRIMOS

SEG is invoked by the PRIMOS command:

SEG [pathname]

A **pathname** is given only when an existing SEG runfile is to be executed. Otherwise, the command transfers control to SEG command level, which prints a "#" prompt character and awaits a subcommand. After executing a subcommand successfully, the loader repeats the prompt character. SEG employs two subprocessors, LOAD and MODIFY, which accept further subcommands. The subprocessors use the "$" prompt character.

If an error occurs during an operation, SEG prints an error message, then the prompt character. Error messages and suggested handling techniques are discussed elsewhere in this section and in Appendix A.

When a system error (FILE IN USE, ILLEGAL NAME, NO RIGHT, etc.) is encountered, SEG prints the system error and returns the prompt symbol. SEG remains in control until a QUIT subcommand returns control to PRIMOS, or an EXECUTE subcommand starts execution of the loaded program.

SEG subcommands can be used in command files, but comment lines are accepted only within the LOAD subprocessor.

NORMAL LOADING

Loading is normally a simple operation with only a few straightforward commands needed. SEG also has many additional features to optimize runfile size or speed, perform difficult loads, load for shared procedures, and deal with possible complications. The most frequently used commands and operations are described first; this enables the immediate use of SEG. Advanced features are then described, followed by a summary of all SEG commands.

The following commands (shown in abbreviated form) accomplish most loading functions:

SEG-Level Commands:

DELETE	Deletes segmented runfile.
HELP	Prints a list of SEG commands at terminal.
LOAD	Invokes loader subprocessor for entry of subcommands.

LOAD Subcommands:

LOAD pathname	Loads specified object file.
LIBRARY [filename]	Loads library object files from UFD LIB. (Default is PFTNLB and IFTNLB, in that order.)
MAP [option]	Prints loadmap. Option 3 shows unresolved references.
INITIALIZE	Returns loader to starting condition in case of command errors or faulty load.
SAVE	Saves loaded memory image as runfile.
RETURN	Returns to SEG command level.
QUIT	Returns to PRIMOS.

Most loads can be accomplished by the following basic procedure:

1. Invoke SEG from PRIMOS level.
2. Enter the LOAD command to start the LOAD subprocessor ($ prompt).
3. Use the load subprocessor's LOAD subcommand to load the object file (B_filename) and any separately compiled subroutines.
4. Use load subprocessor's LIBRARY subcommand to load subroutines called from libraries (the default is PFTNLB and IFTNLB in the UFD LIB). Other libraries, such as VSRTLB or VAPPLB, must be named explicitly.
5. If you do not have a LOAD COMPLETE, do a MAP 3 to identify the unsatisfied references, and load them.
6. SAVE the runfile.

If these commands produce a LOAD COMPLETE message, then loading was accomplished. If there is a problem, it will become apparent by the absence of a LOAD COMPLETE message or some other SEG error message. (See Appendix A for a complete list of all SEG error messages and their probable cause and correction.)

After a successful load, you can either start runfile execution from loader command level, or quit from the loader and start execution through the PRIMOS SEG command. An example of such a load is:

```
OK, SEG
GO
# LOAD
SAVE FILE TREE NAME: #ARRAY
$ LO B_ARRAY
$ LI
$ SA
$ MA M_ARRAY
$ QU

OK,
```

Order of loading

The following loading order is recommended:

1. Main program
2. Separately compiled user-generated subroutines (preferably in order of frequency of use).
3. Other Prime Libraries (LI filename)
4. Standard FORTRAN library (LI)

Loading library subroutines

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library files PFTNLB and IFTNLB in the LIB UFD. The appropriate subroutines from this file are loaded by the LIBRARY command given without a filename argument. If subroutines from other libraries are used, such as VSRTLB or VAPPLB, additional LIBRARY commands are required which include the desired library as an argument.

LOAD MAPS

During loading, SEG collects (and stores, as part of the segmented runfile) information about the results of the load process. This can be printed at the terminal (or written to a file) by the load subprocessor's MAP command:

MAP [pathname] [option]

The information in the map can be consulted to diagnose problems in loading, or to optimize placement of modules, linkage areas and COMMON in complex loads. If a file **pathname** is given, the map is written to a file instead of being printed at the terminal. 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).

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.

The full SEG load map consists of seven sections, not all of which may be present in any load. (See Figure 7-1) In particular, Section III may not be present in small SEG loads. The amount of information printed is controlled by MAP option codes:

Option	**Load Map Information**
None, 0 or 4	Extent, segment assignments, base areas, symbol storage (symbols sorted by address), direct entry links, common blocks, and other symbols.
1	Extent and segment assignments only
2	Extent, segment assignments and base areas
3	Undefined symbols, sorted by address
6	Undefined symbols, sorted alphabetically
7	Full map, symbols, sorted in alphabetic order
10	Symbols, sorted by ascending address
11	Symbols, sorted alphabetically

Section I - Extent

The extent area shows where the program has been loaded, the start-of-execution location, and the size of the symbol table. All locations are octal numbers.

***START:** The segment number and word location for the start-of-execution. At the beginning of a load, the start address is initialized to 000000 000000. SEG fills in *START for the first segmented procedure encountered (usually the main program).

***STACK:** Segment number and word location 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 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; the top and bottom of the symbol table will be identical and *SYM will also be the location of the top.

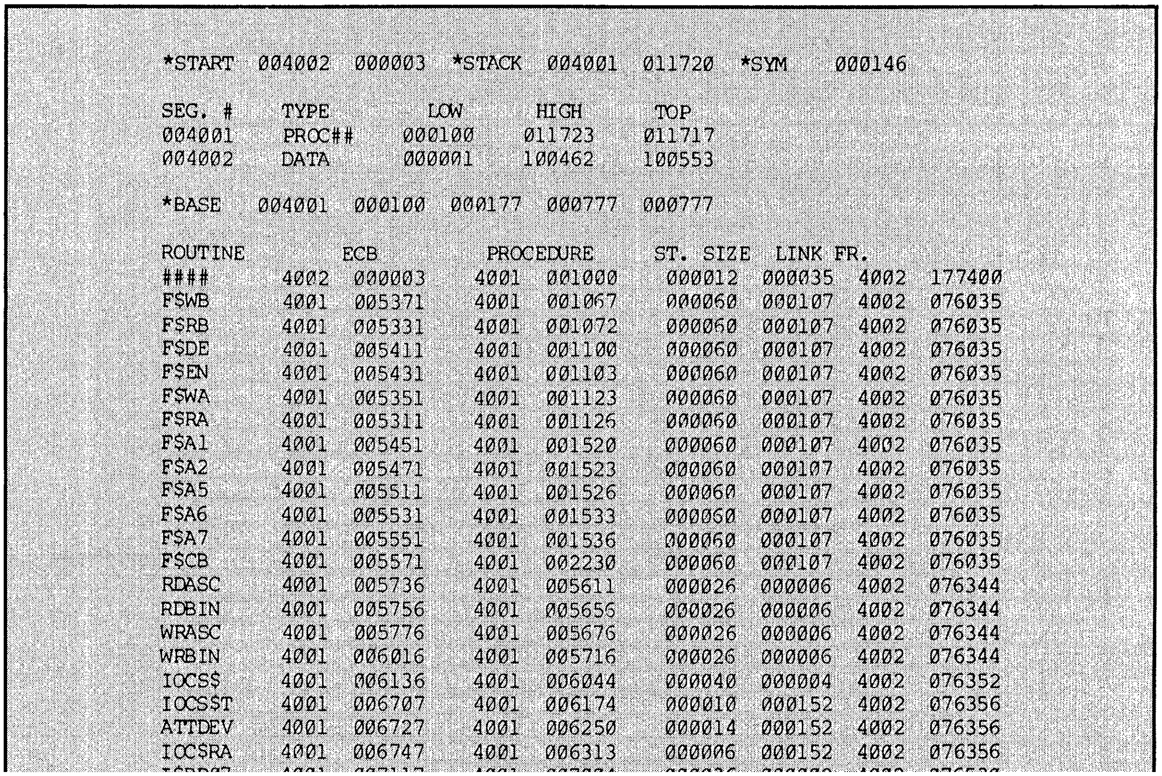
Section II - Segment assignments

Each segment is labeled as procedure (PROC) or data (DATA); the segment chosen for the stack is identified by ## following the segment type. The list is sorted in order of segment assignment.

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



```
*START  004002  000003  *STACK  004001  011720  *SYM    000146

SEG. #    TYPE        LOW      HIGH      TOP
004001    PROC##    000100    011723    011717
004002    DATA      000001    100462    100553

*BASE   004001  000100  000177  000777  000777

ROUTINE         ECB          PROCEDURE      ST. SIZE  LINK FR.
####      4002  000003    4001  001000    000012  000035  4002  177400
F$WB      4001  005371    4001  001067    000060  000107  4002  076035
F$RB      4001  005331    4001  001072    000060  000107  4002  076035
F$DE      4001  005411    4001  001100    000060  000107  4002  076035
F$EN      4001  005431    4001  001103    000060  000107  4002  076035
F$WA      4001  005351    4001  001123    000060  000107  4002  076035
F$RA      4001  005311    4001  001126    000060  000107  4002  076035
F$A1      4001  005451    4001  001520    000060  000107  4002  076035
F$A2      4001  005471    4001  001523    000060  000107  4002  076035
F$A5      4001  005511    4001  001526    000060  000107  4002  076035
F$A6      4001  005531    4001  001533    000060  000107  4002  076035
F$A7      4001  005551    4001  001536    000060  000107  4002  076035
F$CB      4001  005571    4001  002230    000060  000107  4002  076035
RDASC     4001  005736    4001  005611    000026  000006  4002  076344
RDBIN     4001  005756    4001  005656    000026  000006  4002  076344
WRASC     4001  005776    4001  005676    000026  000006  4002  076344
WRBIN     4001  006016    4001  005716    000026  000006  4002  076344
IOCS$     4001  006136    4001  006044    000040  000004  4002  076352
IOCS$T    4001  006707    4001  006174    000010  000152  4002  076356
ATTDEV    4001  006727    4001  006250    000014  000152  4002  076356
IOC$RA    4001  006747    4001  006313    000006  000152  4002  076356
```

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 - Base areas

***BASE VVVVVV WWWWWW XXXXXX YYYYYY ZZZZZZ**

VVVVVV	Segment number.
WWWWWW	Lowest location for base area.
XXXXXX	Next available location if starting down from highest 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 - Symbols

A main program or subroutine compiled in 64V mode is called a procedure. A procedure is composed of a procedure frame (the executable code), and ECB (the entry control block), a link frame (static storage, constants, transfer vectors) and a stack frame (dynamically allocated storage which is assigned 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.

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.

LINK FR.: is the size of the link frame.

The last two columns are the link frame segment and offset. Note that the offset 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 for the ECB.

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

Section V - Direct entry links

PRIMOS 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 is not concerned with this map section.

Section VI - COMMON blocks

Lists each COMMON block, its segment number, starting word address in the segment, and size.

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 **.

The numbers for unsatisfied references (segment and word address) locate the last request for the routine processed by the Loader. This allows the routines calling missing routines to be identified.

ADVANCED SEG FEATURES

When standard loading goes well, the user can ignore most of the SEG's advanced features. However, situations can arise where some detailed knowledge of SEG and segmented runfile organization can optimize size or performance of a runfile, or even make a critical load possible. The following topics are of particular use to the FORTRAN programmer.

Segment usage

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 runfile of up to 256 segments.

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.

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 also works but is slower than SEG's DELETE.)

Note

> It is good practice to use the PRIMOS DELSEG command to release segments assigned by SEG during a load session. Otherwise those segments remain assigned to the user until logout, precluding their use by anyone else.

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 directory. A runfile compiled and loaded in 32R or 64R mode 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.

Object files

Object files of the program modules must have been created using the FORTRAN compiler's -64V option. Modules written in other languages may also be loaded, if they have been compiled or assembled in 64V mode.

Code and data are loaded in separate segments to support re-entrant procedures. Data includes 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, using the MIXUP subcommand of the LOAD subprocessor.

The stack

The loader assigns a stack (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 is 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.

Use of pathnames

Pathnames can be used to specify object files in all commands except LIBRARY, which accepts *only* a simple filename of a file within the LIB UFD.

Base areas

"Base area" is an assembly language concept that can be disregarded by the FORTRAN programmer unless the following message is printed:

SECTOR 0 BASE AREA FULL

This condition, which is *extremely* unlikely to occur, can be avoided by using the SETBASE command to designate a base area where it is required.

Locating COMMON

SEG makes sure there is no overlap of program and COMMON. The user has the option of moving COMMON by a COMMON or SYM command, but he takes on the responsibility of making sure it doesn't run into the stack.

COMMAND SUMMARY

Following is a summary of all SEG commands, in alphabetical order within three groups:

1. SEG-level commands
2. LOAD-subprocessor
3. MODIFY subprocessor.

Files and directory names may be specified by pathnames, except in the LIBRARY command. All numerical values must be octal. The following conventions are followed for parameters.

addr	Word address within a segment.
segno	Segment number.
psegno	Procedure segment number.
lsegno	Linkage segment number.
[a] [b] [x]	Values for A, B, and X registers.

Note

Segment numbers may be absolute or relative. See Section 11 for further information.

SEG - LEVEL COMMANDS

Commands at SEG level are entered in response to the "#" prompt.

DELETE [pathname]

Deletes a saved SEG runfile.

HELP

Prints abbreviated list of SEG commands at terminal.

[V]LOAD [pathname]

Defines runfile name and invokes virtual loader for creation of new runfile (if name did not exist) or appending to existing runfile (if name exists). If **pathname** is omitted, SEG requests one.

MAP pathname-1 [pathname-2] [map-option]

Prints a loadmap of runfile (pathname-1) or current loadfile (*) at terminal or optional file (pathname-2).

	Option
0	Full map [*default*]
1	Extent map only
2	Extent map and base areas
3	Undefined symbols only
4	Full map [identical to 0]
5	System programmer's map
6	Undefined symbols, alphabetical order
7	Full map, sorted alphabetically
10	Symbols by ascending address
11	Symbols alphabetically

MODIFY [pathname]

Invokes MODIFY subprocessor to create a new runfile or modify an existing runfile.

PARAMS [pathname]

Displays the parameters of a SEG runfile.

PSD

Invokes VPSD debugging utility.

QUIT

Returns to PRIMOS command level and closes all open files.

RESTORE [pathname]

Restores a SEG runfile to memory for examination with VPSD.

RESUME [pathname]

Restores runfile and begins execution.

SAVE [pathname]

Synonym for MODIFY.

SHARE [pathname]

Converts portions of SEG runfile corresponding to segments below '4001 into R-mode-like runfiles. (See Section 11 for more information.)

SINGLE [pathname] segno

Creates an R-mode-like runfile for any segment.

TIME [pathname]

Prints time and date of last runfile modification.

VERSION

Displays SEG version number.

VLOAD

See **LOAD**.

LOAD SUBPROCESSOR COMMANDS

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

Attaches to directory.

A/SYMBOL symbolname [segtype] segno size

Defines a symbol in memory and reserves space for it using absolute segment numbers.

COMMON $\left\{ \begin{matrix} \textbf{[ABS]} \\ \textbf{REL} \end{matrix} \right\}$ **segno**

Relocates COMMON using absolute or relative segment numbers.

D/ $\left\{ \begin{matrix} \textbf{IL} \\ \textbf{LOAD} \\ \textbf{LIBRARY} \\ \textbf{FORCELOAD} \\ \textbf{PL or RL} \end{matrix} \right\}$

Continues a load using parameters of previous load command.

Note

D/ and F/ may be combined, as in D/F/LI.

EXECUTE [a] [b] [x]

Performs SAVE and executes program.

$$\textbf{F/}\left\{\begin{array}{l}\textbf{IL}\\ \textbf{LOAD}\\ \textbf{LIBRARY}\\ \textbf{FORCELOAD}\\ \textbf{PL}\\ \textbf{RL}\end{array}\right\}\textbf{[pathname] [addr psegno lsegno]}$$

Forceloads all routines in object file.

IL [addr psegno lsegno]

Loads impure FORTRAN library IFTNLB.

INITIALIZE [pathname]

Initializes and restarts load subprocessor.

LIBRARY [filename] [addr psegno lsegno]

Loads library file (PFTNLB and IFTNLB if no **filename** specified).

LOAD [pathname] [addr psegno lsegno]

Loads object file.

MAP [pathname] option

Generates load map (see SEG-level MAP command).

$$\textbf{MIXUP}\left\{\begin{array}{l}\textbf{[ON]}\\ \textbf{OFF}\end{array}\right\}$$

Mixes procedure and data in segments and permits loading of linkage and common areas in procedure segments. *Not* reset by INITIALIZE.

MV [start-symbol move-block desegno]

Moves portion of loaded file (for libraries). If options are omitted information is requested.

OPERATOR option

Enables or removes system privileges 0 = enable, 1 = remove. Caution: this command is intended *only* for knowledgeable creators of specialized software.

PL [addr psegno [segno]

Loads pure FORTRAN library, PFTNLB.

$$\textbf{P/}\left\{\begin{array}{l}\textbf{IL}\\ \textbf{LOAD}\\ \textbf{LIBRARY}\\ \textbf{FORCELOAD}\\ \textbf{PL}\\ \textbf{RL}\end{array}\right\}\textbf{[pathname] option [psegno] lsegno]}$$

Loads on a page boundary. The options are: PR = procedure only, DA = link frames only, none = both procedure and link frames.

QUIT

Performs SAVE and returns to PRIMOS command level.

RETURN

Performs SAVE and returns to SEG command level.

RL pathname [addr psegno lsegno]

Replaces a binary module in an established runfile.

R/SYMBOL symbol-name [segtype] segno size

Defines a symbol in memory and reserves space for it using relative segment assignment.(Default = data segment).

SAVE [a] [b] [x]

Saves the results of a load on disk.

SETBASE segno length

Creates base area for desectorization.

$$\textbf{SPLIT} \left\{ \begin{array}{l} \textbf{segno addr} \\ \textbf{addr} \\ \textbf{addr segno addr lsegno} \end{array} \right\}$$

Splits segment into data and prodecure protions. Formats 2 and 3 allow R mode execution if all loaded information is in segment 4000.

SS symbol-name

Saves symbol; prevents XPUNGE from deleting symbol-name.

STACK Size

Sets minimum stack size.

SYMBOL [symbol-name] segno addr

Defines a symbol at specific location in a segment.

$$\textbf{S/} \left\{ \begin{array}{l} \textbf{LIBRARY} \\ \textbf{FORCELOAD} \\ \textbf{PL or IL} \\ \textbf{RL or LOAD} \end{array} \right\} \textbf{[pathname] [addr psegno lsegno]}$$

Loads an object file in specified absolute segments.

XP dsymbol dbase

Expunges symbol from symbol table and deletes base information.

dsymbol	Action
0	Delete all defined symbols—including COMMON area.
1	Delete only entry points, leaving COMMON areas.

dbase	**Action**
0	Retain all base information.
1	Retain only sector zero information.
2	Delete all base information.

MODIFY SUBPROCESSOR COMMANDS

NEW pathname

Writes a new copy of SEG runfile to disk.

PATCH segno baddr taddr

Adds a patch (loaded between **baddr** and **taddr**) to an existing runfile and saves it on disk.

RETURN

Returns to SEG command level.

$$\textbf{SK} \left\{\begin{array}{l}\textbf{ssize} \\ \textbf{segno addr} \\ \textbf{ssize 0 esegno} \\ \textbf{ssegno addr esegno}\end{array}\right\}$$

Specifies stack size (ssize) and location. **esegno** specifies an extension stack segment.

START segno addr

Changes program execution starting address.

WRITE

Writes all segments above '4000 of current runfile to disk.

8

Executing programs

INTRODUCTION

This section treats the following topics:

- Execution of program memory images saved by the Loader
- Execution of segmented runfiles saved by SEG's Loader
- Run-time error messages
- Installation of programs in Command UFD (CMDNC0)

EXECUTION OF R-MODE MEMORY IMAGES

For programs loaded in 32R or 64R mode by the loader, execution is performed at the PRIMOS level using the RESUME command. Programs which are already resident in the user's memory may be executed by a START command.

RESUME pathname

RESUME brings the memory-image program **pathname** 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 (for example, 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 BREAK 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 Reference Guide, PRIMOS Commands.

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

EXECUTING SEGMENTED RUNFILES

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

SEG pathname

where **pathname** is the name 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 R-mode loader.

Example:

```
OK, SEG #TEST       User requests program
GO                  Execution begins
THIS IS A TEST      Output of program

OK,                 PRIMOS requests next command
```

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

A SEG runfile may be restarted by the command:

S 1000

if 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 messages 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 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.

File system calls

In the 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)

standard-text	The file system standard error message (listed in Appendix A).
user's-text-if-any	An optional message which the user may elect to have printed.
(name-if-any)	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$$)
```

Note

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

The file system is described in Reference Guide, PRIMOS Subroutines. In the list of standard error messages for 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.)

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 imply system problems beyond the scope of the applications programmer. If so, this is indicated in the explanation of a given error message.

INSTALLATION IN THE COMMAND UFD (CMDNC0)

Run-time programs in the command UFD (CMDNC0) 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=CMDNC0.

Even more space is saved during execution by multiple users if the program uses shared code (64V mode only). (See Section 11).

Program memory images saved by the loader

Installation in the command UFD is extremely simple. The runtime version of the program is copied into UFD=CMDNC0 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* 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-REV16.0]         Compiler message
OK, LOAD                                  Invoke the Loader
GO
$LO B_FARLEY                              Load the object file; the default
                                            name is used
$                                         Load other required modules
 .
 .
 .
$LI                                       Load the FORTRAN library
LOAD COMPLETE                             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:

1. Create the desired SEG runfile.
2. Attach to UFD=SEG, which contains the command file CMDSEG.
3. Run the command file CMDSEG using COMINPUT; it will ask for runfile pathname as the new SEG runfile name. This command file will create the interlude program under the name *TEST.
4. If you did not give a pathname for the runfile, 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.
5. A copy of *TEST should be placed in UFD=CMDNC0 using FUTIL's COPY command. The file name should be that by which the program will be invoked.

Example:

1. 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-REV16.0]

OK, SEG                              Invoke SEG utility
GO
# LOAD #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,
```

2. ATTACH to UFD=SEG

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

3. The command file CMDSEG creates the interlude program.

```
OK, CO CMDSEG

OK, * CMDSEG,SEG,CEH.04/05/78
OK, * COMMAND.FILE.TO.CREATE.'CMDNC0'.SEG.RUNFILES
OK, R *CMDMA
GO
RUN FILE NAME: FARLEY
OK, FTN S$$SEG 1/5707
GO
0000 ERRORS [<.MAIN.>FTN-REV16.0]
OK, FILMEM
OK, LOAD
$SZ
$ER 2
$MO D64R
$CO 173400
$LO B_S$$SEG 173400
$AU 2
$LO CMDLIB * 12 14 14 0 0 12 0 0 12
$AU 0
$LI
$MA 2
$SAVE *TEST
$AT
$QU
OK, DELETE S$$SEG
OK, DELETE B_S$$SEG
OK, CO TTY

OK,
```

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

```
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
```

5. 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 CMDNC0 ORDER              TO UFD=CMDNC0; 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 CMDNC0 is executed. This program attaches to the SEG UFD, restores the segmented runfile FARLEY, re-attaches to the user's home directory 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 '4000 and above) it is possible to include the interlude in the SEG runfile. This is discussed in Section 11.

9

Debugging

INTRODUCTION

This section discusses the various debugging tools and strategies available to the Prime FORTRAN programmer. For a good discussion of debugging techniques (as well as preventive programming methodology), the reader is referred to *The Elements of Programming Style*, Kernigan and Plauger, McGraw-Hill, 1978 (Second Edition).

Debugging is discussed in the following areas:

- Coding strategy
- Compiler usage
- Program execution
- The PM command
- Program validation

CODING STRATEGY

Coding strategy involves avoiding traditional errors so as to eliminate the need for debugging later. (Section 13 contains information on coding optimization.) The four major techniques for coding are:

1. Modular program structure.
2. Proper use of comments.
3. Effective use of indention and spacing.
4. Inserting TRACE statements to monitor program control flow.

Modular program structure

Modular program structure is the building up of a large program or system from a set of small, self-contained program modules. Each module performs a discrete, specific task, and contains all necessary comments, diagnostics and error messages. This permits the programmer to design, code, compile, load, execute, debug and maintain each portion of the master program individually (though certain programs may need to be run in "artificial" environments or with test routines that simulate other portions of the master program).

Once the master program nears completion, modular structure allows the programmer to isolate problems back to specific modules, permitting simpler and more reliable bug fixes.

Proper use of comments

As pointed out in *Elements of Programming Style*, the proper use of comments can vastly improve a program's usability by its own and other programmers, while bad comments can seriously interfere. Comments should, as a rule, offer succinct information as to the purpose and intent of upcoming code, and not simply restate the code.

Note

> One method of commenting worth consideration is that of placing the majority of comments on the right-hand side of the file (the actual code being on the left). This allows the programmer to cover over comments when re-inspecting code, leading to the possible discovery that it does *not* perform the claimed task as stated in the accompanying comment.

Effective use of indention and spacing

Indention and spacing, when properly used, help display the parallelism, symmetry and/or consistency (or lack thereof) in a given portion of code.

Inserting TRACE statements to monitor program control flow

The FORTRAN TRACE statement permits the monitoring of program control flow by displaying values of specified variables whenever they are changed during program execution. TRACE is explained in Section 15. By monitoring the values of given variables, you can often determine at what places your program is not working as desired, and from there investigate the cause.

COMPILER USAGE

Compile-time debugging consists of the following operations:

1. Syntax checking and compile-time errors.
2. DCLVAR and global TRACE compiler options.

Syntax checking

The FORTRAN compiler automatically performs syntax checking as part of the compiling process. Syntax errors are usually due to coding or typing errors. (Remember that what the compiler perceives as a syntax error may often be the result of some other error elsewhere in the program; e.g., the compiler will flag the statement GOTO 140 if there is no statement 140, or if there is an error in statement 140.)

If your program has syntax errors, do *not* attempt to load and execute it; make the necessary corrections first.

Other compile-time errors

The compiler also checks for non-syntactical errors, such as program length exceeding available user space. As with syntactical errors, do not attempt to load and execute a program which has non-syntactical errors.

The DCLVAR and global TRACE compiler options

The DCLVAR option to the FTN command causes the compiler to flag all variables which are not *explicitly* declared in specification statements. This procedure often uncovers minor spelling errors in the source file (e.g., you defined the variable TEMP.A1, but elsewhere typed it as TEMPA.1).

The TRACE option produces a trace for every variable in the program. This option takes precedence over any TRACE statement in your FORTRAN program, and is particularly helpful in conjunction with the PRIMOS COMOUTPUT command (given prior to the FTN command), which will thus send all TRACE output to a file. (See Section 10 for COMOUTPUT information).

See Section 5 for more information on these compiler options.

3

ADVANCED PROGRAMMING TECHNIQUES

10

Operating system features

This section discusses some PRIMOS utilities that are useful to most FORTRAN programmers. These are:

- Command file operations (COMINPUT, COMOUTPUT, PHANTOM and CX)
- Phantom users (PHANTOM)
- Sequential job processing (CX)
- Magnetic tape utilities (MAGNET, MAGSAV, MAGRST)
- PRIMENET
- File utility (FUTIL)
- SORT utility
- File compare and merge commands (CMPF, MRGF)
- Terminal control (TERM)

For more details on these and other topics, see Reference Guide, PRIMOS Commands.

COMMAND FILE OPERATIONS

PRIMOS offers three utilities that allow command sequences to run from files rather than from direct user interaction. They are:

COMINPUT	Reads commands from a specified file. Commands and responses appear on terminal. Terminal is dedicated to this operation during execution.
PHANTOM	Reads commands from a file but executes as another PRIMOS process, freeing terminal for other use. Limited number of phantom processes are available, so user may have to wait for a free process.
CX	Sequential job processor. Operates like PHANTOM but queues a large number of command files and can be interrogated about job status.

All of these utilities read commands from a command file, which is a file containing PRIMOS commands, utility subcommands, and dialog responses. The user creates the file with the editor, runs it under COMINPUT to verify operation, edits it to make changes, and thereafter runs it under COMINPUT, PHANTOM or CX. This is particularly useful for long program development operations that must be repeated whenever source code is changed, building libraries, production job runs, etc.

Supporting the three command processing utilities is the COMOUTPUT command which maintains an audit file of the dialog between PRIMOS and the command file. Other useful PRIMOS commands are TIME and DATE. These commands are described later in this section.

Command file requirements

Command input files may contain any legal PRIMOS commands, utility subcommands, or dialog responses, on a line-for-line basis i.e., each line in the file must correspond to a line as it would be typed at a terminal. Each utility imposes certain requirements:

- For COMINPUT, the last command should be COMINPUT -TTY or COMINPUT -END.
- For PHANTOM, the last command must be LOGOUT.
- For CX, the first command must be an ATTACH to the desired working directory and the last command must be CO -TTY, CX -E, or LOGOUT.

Comments: command input files can be made self-documenting by including comment lines at PRIMOS command level. A line beginning with a slash and asterisk, (/*), is interpreted as a comment and is ignored by PRIMOS. If a command output file is open, any comments entered at the terminal by the user or from a command file are written into the command output file. Any character may be used in a comment line. A comment may also be appended to a command at PRIMOS command level as in:

```
SLIST M_BENCH07                 /* PRINT MAP FILE
```

The COMINPUT Command

The COMINPUT command causes PRIMOS to read input from a specified command file rather than from the terminal. Commands are executed as if they were entered at the terminal. The format is:

COMINPUT [command-file] [-options] [file-unit]

command-file	The pathname of the file from which input is to be read.
options	Specify command control flow as detailed below.
file-unit	The PRIMOS file unit number on which the input file is to be opened. If omitted, file unit 6 is used. File units *must* be octal (i.e., decimal 8 is entered as 10).

Options:

-TTY **-END**	Either one switches the command input stream to the user terminal and closes the command input file.
-PAUSE	Switches command input stream to the user terminal but does not close the command input file.
-CONTINUE	Returns control to command input file following a CO -PAUSE or an error.
-START	Resumes command following a BREAK interruption of execution of a command file.

The -TTY, -END and -PAUSE options are used only within command files. The -CONTINUE and -START options are typed by the user.

The -TTY, -END option must be the final command in the command file (or in the last command file, if files are chained as described below.)

Chaining command files: The -CONTINUE option of COMINPUT allows command files to be chained. The following example illustrates the chaining of three command files, and shows how file unit conflicts can be avoided. The command file C_GO contains the following commands:

```
/* COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
/* LOAD THE PROGRAM
COMINPUT C_LOADTEST 7
CLOSE 7
/* RETURN COMMAND TO USER TERMINAL
COMINPUT -TTY
```

The command file C_LOADTEST contains the following commands:

```
/* LOADTEST COMMAND FILE
SEG
VLOAD #FTN.TEST
LO B_FTN.TEST
LI
SA
QU
COMINPUT C_MAPS 10
CLOSE 8
COMINPUT -CONTINUE
```

The command file C_MAPS contains the following commands:

```
/* GET FULL MAP AND UNSATISFIED REFERENCES
SEG
VLOAD * #FTN.TEST
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
QU
/* RETURN TO 'CALLING' COMMAND FILE
COMINPUT -CONTINUE
```

Typing COMINPUT C_GO causes the commands in C_GO to be executed; the COMINPUT C_LOADTEST 7 command causes input to be read from C_LOADTEST (opened on file unit 7). The COMINPUT -CONTINUE command in C_LOADTEST causes input to be read from the command file opened on unit 6 (C_GO). Since C_GO was not closed, its file pointer is at the command following the one invoking input from C_LOADTEST. In a similar manner the command file C_MAPS is invoked from C_LOADTEST on file unit 8 ('10). Execution of CO _GO results in the following terminal output:

```
OK, CO C_GO
OK, /* COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
GO
0000 ERRORS [<.MAIN.>FTN-REV16.1]

OK, /* LOAD THE PROGRAM
COMINPUT C_LOADTEST 7
OK, /* LOADTEST COMMAND FILE
SEG
GO
# VLOAD #FTN.TEST
$ LO B_FTN.TEST
$ LI
LOAD COMPLETE
$ SA
$ QU

OK, COMINPUT C_MAPS 10
OK, /* GET FULL MAP AND UNSATISFIED REFERENCES
SEG
GO
# VLOAD * #FTN.TEST
$ MAP M_LOADTEST 7
$ MAP M_UNSATISFIED 3
$ QU

OK, /* RETURN TO 'CALLING' COMMAND FILE
COMINPUT -CONTINUE
OK, CLOSE 7
OK, /* RETURN COMMAND TO USER TERMINAL
COMINPUT -TTY
OK,
```

Errors: Non-recoverable errors return input control to the terminal, leaving the command file open. The user may type a correct version of the offending line, and then resume input from the command file by the command CO -CONTINUE.

Closing command input files: In chaining command files, the 'called' files should be closed upon returning to the 'calling' files, either by file unit number (as in the example above) or by filename. The user should make certain that the file units to be used for the command input files are not already opened (or going to be opened) by user programs, utilities, or other command input files.

Note

The CLOSE ALL command should not be used in a command input file, as it closes all files, including the command input file from which this command is read. The message COMINP FILE EOF will be printed and input control will be switched to the terminal.

The COMOUTPUT command

The COMOUTPUT command writes, into a specified file, both the output stream directed to the terminal by PRIMOS and the input presented to PRIMOS. The input may originate as direct typing, or come from a command file running under COMINPUT, PHANTOM or CX. The resulting output file is a permanent record of the entire dialog.

Output to the terminal can be suppressed. Print suppression increases speed since it normally takes more time to write to a terminal than to a disk file.

The command format is:

COMOUTPUT [output-file] [-options]

output-file is the pathname of the file to which the output stream is sent. **options** specify terminal and file output and control flow as described below.

Terminal options: These can be used when the output file is first opened, or at any time before the command output file is closed. User input is always echoed at the terminal even if the -NTTY option is used.

-NTTY	Turn off terminal output.
-TTY	Turn on terminal output (default).

Error messages are printed in the output file and at the terminal, regardless of the terminal option selected. Any inter-user terminal output (e.g., messages from the supervisor terminal) is printed at the terminal but not in the output file.

File options: These stop or restart output to the command file. They may also be used to append output to an existing file.

-PAUSE	Stop output to command file; leave file open.
-CONTINUE	Resume output (halted by -PAUSE) to the command output file. Or, if at PRIMOS level, re-open an existing COMINPUT file and position the pointer so that new output will be appended.
-END	Stop output to command file; close file.

A BREAK turns terminal output on, but does not close the file. A LOGOUT turns terminal output on and also closes the command output file, as well as any other files the user has currently open.

Examples:

```
COMO O_FTNTEST
```

opens the file O_FTNTEST for output and positions the pointer to the start of the file. If O _FTNTEST already exists, its previous contents will be deleted. To open an existing file for appending, typing:

```
COMO O_FTNTEST -C
```

opens the file O_FTNTEST and positions the pointer at the end of the file.

Closing command output files: The command output file is normally closed by the COMO -END command. The user may desire to close the file at other times (say, after a BREAK). Since COMOUTPUT uses file unit 63 ('77), the CLOSE ALL command will not close this file. The file may be closed with:

CLOSE output-file

or

CLOSE 77 (*must* be octal value)

or

COMO -END

Using DATE and TIME in command files

The DATE command: The command DATE prints the system date and time at the user terminal.

```
OK, DATE
GO

Wednesday, June 7, 1978   10:11 AM

OK,
```

This feature allows command output files to be stamped with date/time information for identification, as an aid to program development and debugging. For example, the sequence of commands:

```
COMO O_TEST1
DATE
.
.
.
DATE
COMO -END
```

creates a file, O_TEST1. The first line of this file is the DATE command; the next line is the time and date of this interactive session.

DATE may also be included in command input files or in command files for the sequential job processor (CX).

The TIME command: The command TIME entered at the user terminal prints the current values in the time accounting registers. These are: connect time, compute time, and disk I/O time.

```
OK, TIME
  1'32   0'11   0'08
OK,
```

Connect time is the time since LOGIN (in hours and minutes). Compute time is the time accumulated executing commands or using programs (in minutes and seconds). This does not include disk I/O time. Disk I/O time (in minutes and seconds) is the accumulated time for disk input and output. Disk I/O includes paging I/O time generated on the user's behalf. All times include system supervisor overhead caused by user requirements.

The TIME command can be given before and after executing a program. The time differences can be used to benchmark the program and measure efficiency as the program is optimized.

Example: the command input file C_BENCH07 contains the following:

```
COMO O_BENCH07
/* TIMING TEST OF BENCH07 PROGRAM
DATE
/* GET START TIME VALUES
TIME
SEG #FTN.TEST
/* GET STOP TIME VALUES
TIME
COMO -END
CO -TTY
```

The command CO C_BENCH07 executes this command file. Upon completion, the output file O_BENCH07 contains the following:

```
OK, /* TIMING TEST OF BENCH07 PROGRAM
DATE
GO

Wednesday, June 7, 1978   9:59 AM

OK, /*
/* GET START TIME VALUES
/*
TIME
  1'12   0'03   0'03
OK, /*
SEG #FTN.TEST
GO
THIS IS A TEST

OK, /*
/* GET STOP TIME VALUES
/*
TIME
  1'12   0'04   0'05
OK, /*
COMO -END
```

PHANTOM USERS

The phantom user feature allows command file processing without tieing up a terminal. Once a phantom process has been initiated, it is treated by PRIMOS as a separate process that is not associated with a terminal. The terminal is then made available for other uses.

The command file run by the phantom process specifies the commands and their sequence, program invocations and necessary input data required to complete a particular job. Phantoms are used for long compilations, loadings, and executions that are debugged and require no interactive terminal input. Certain PRIMOS system utilities (e.g., FAM, SPOOL) are implemented as phantom processes.

Using PHANTOM

A phantom user process is initiated by the command:

PHANTOM filename [file-unit]

filename is the name of a command input file, and **file-unit** is the PRIMOS file unit number on which the command file is to be opened. If omitted, file unit 6 is used.

The PHANTOM command checks for available phantom processes. The number varies with each installation. The message

NO FREE PHANTOMS

is returned if no processes are available. Control is then returned to PRIMOS. When a phantom process is available, the message

PHANTOM USER IS user-number

is returned and the phantom user is logged in (under the same login-name as the invoker). The home and current directories of the phantom are set to the current directory of the originating user. **User-number** is the number assigned by PRIMOS to the PHANTOM process. Control returns to PRIMOS, the terminal is freed for other use, and the phantom command file is opened on the specified (or default) unit. PRIMOS then reads all further commands for the phantom user from the command file.

Phantom operation

Phantom processes should not execute programs which require input from an actual terminal. Such an instruction will abort and log out the phantom process. This logout information is printed only at the supervisor terminal.

While a phantom process is in operation, generated output is suppressed unless a command output file has been opened by a COMOUTPUT command in the phantom command file. Output is then written to the COMOUTPUT file.

It is possible to initiate another phantom from a running phantom, in a manner similar to chained COMINPUT files. However, there is no guarantee that a phantom user process will be available when the process is requested by a command file.

The final command in the last executed phantom command file should be LOGOUT.

Phantom logout

At the completion of a job process, phantom users are automatically logged out. To cancel a phantom user process before completion, use the command:

LOGOUT -usernumber

usernumber is the PRIMOS-assigned phantom user number

Any phantom can be logged out from the supervisor terminal. From a user terminal, a phantom can be logged out only if the terminal has the same login UFD as that which initiated the phantom.

Phantom STATUS information

The STATUS USER command (discussed in Section 2), provides a list of all the users in the system, their login numbers, assigned line numbers, etc. Phantom users are distinguished by the line number 77 in a STATUS list. In the following example, the phantom users are numbers 57 and 58, as indicated by LIN = 77. These phantom processes were initiated by users logged into SYSTEM and FAM respectively.

Example:

```
OK, STATUS USER

USER   NO LIN PDEVS
TEKMAN 43 51  50460
SILVA  44 52  10460
BD     45 53  61060
SYSTEM 57 77   460
FAM    58 77   460 (2)
SYSTEM 59 77   61060

OK,
```

Example of phantom command file

The phantom command file PH.TEST contains the following commands:

```
/* BEGIN TEST OF PHANTOM
COMOUTPUT O_PH.TEST
DATE
/* COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
/* LOAD THE PROGRAM
SEG
VLOAD #FTN.TEST
LO B_FTN.TEST
LI
SA
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
QU
/* PHANTOM TEST COMPLETED
DATE
LOGOUT
```

When a phantom is invoked at the terminal by PH PH.TEST, the terminal interactive dialog is:

```
OK, PH PH.TEST
PHANTOM IS USER 61
OK,
```

The contents of the command file, O_PH.TEST created by the phantom are:

```
OK, DATE
GO

Wednesday, June 7, 1978   3:27 PM

OK, /* COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
GO
0000 ERRORS [<.MAIN.>FTN-REV15.1]

OK, /* LOAD THE PROGRAM
SEG
GO
# VLOAD #FTN.TEST
$ LO B_FTN.TEST
$ LI
LOAD COMPLETE
$ SA
$ MAP M_LOADTEST 7
$ MAP M_UNSATISFIED 3
$ QU

OK, /* PHANTOM TEST COMPLETED
DATE
GO

Wednesday, June 7, 1978   3:28 PM

OK, LOGOUT
TEKMAN (61) LOGGED OUT AT  15'28 060778
TIME USED=  0'01   0'04   0'10
```

SEQUENTIAL JOB PROCESSOR (CX)

The CX utility handles the queuing of jobs for sequential execution as phantoms. Jobs may be:

- Run simultaneously (multiple job streams).
- Queued according to priority (priority levels).
- Restricted to a specified amount of CPU time.

Using CX

Jobs are submitted by passing the name of a command input file to the CX queue manager. The command format is:

CX pathname [-PRIORITY level] [-CPULIMIT cpu-seconds]

pathname	The name of the command input file from which the CX processor will read commands.
-PRIORITY level	Optional assignment of job priority. Standard priority range is 0–7. Default is level 3 (median).
-CPULIMIT	Time allowed for job to run, in CPU seconds. A number from 1 to 2147483647 (0 is illegal). Default = NONE (no time limit). Job is logged out after the limit is reached.

The range of priority levels and CPU-limits are installation configurable. Check with your System Administrator for the range of values implemented on your system.

Job file number

When the CX command and options have been specified, the system responds with the following message:

YOUR JOB FILE IS CX##queue-number

queue-number is a 2-digit number identifying the job in the CX queue. This number is assigned by the CX queue manager.

Job ID

A CX job may be given an ID for ease of identification in the CX queue. The first six characters immediately following the first * (the second symbol of a comment indicator) occurring in the command file are taken as the job ID label. This ID label is printed in STATUS interrogation requests. (see below).

Example:

```
/* CXTEST IS JOB ID
```

The letters CXTEST will be taken as the job ID of the command file called CXTEST.

CX command files

A CX command file is a command input file. It is the same as a command file used for PHANTOM execution with the following exceptions:

- The first executable command must be an ATTACH to the desired working directory. (CX initially logs in the job in its own UFD.)
- The last command may be either CO -TTY, LOGOUT, or CX -E. A job will be listed as ABORTED in the queue if it terminates without one of these commands.

CX queue information

The status of the CX queue can be determined by issuing the CX command, followed by one of these options:

-A	Lists entire activity file.
-P	Lists all jobs belonging to user.
-Q	Prints job queue.
-Snn	Prints status of job **nn**.

For example,

```
OK, CX -A
GO

CX JOB FILE LISTING   78/06/08   2:07 PM

 FILE     ID     OWNER     STATE          DATE/TIME
------   ------  ------  ----------      -----------

CX##07  .TIMDA   LSMITH  COMPLETED  78/04/18   1:47 PM
CX##06  .TIMDA   LSMITH  ABORTED    78/04/18   1:38 PM
CX##05  .TIMDA   LSMITH  COMPLETED  78/04/18   1:35 PM
CX##04  .TIMDA   LSMITH  COMPLETED  78/04/18   1:26 PM
CX##03  .TIMDA   LSMITH  ABORTED    78/04/18   1:18 PM
CX##02  .TIMDA   LSMITH  COMPLETED  78/04/14   5:23 PM
CX##01  .TIMDA   LSMITH  COMPLETED  78/04/14   5:15 PM
```

Dropping jobs from the queue

A job waiting in the CX queue may be dropped by the command:

CX -Dnn

nn specifies which job is to be dropped from the waiting queue.

A job *cannot* be dropped from the queue if the job is executing. A user may only drop from the queue jobs which were entered under the user's login name. For example:

```
OK, CX -D01
GO

?CAN'T - NOT YOUR JOB

OK,
```

Example of CX usage

The command file CXTEST in UFD=TEKMAN>PDR3057>FDR3057 contains the following commands:

```
/* CXTEST IS JOB ID
A TEKMAN>PDR3057>FDR3057
COMOUTPUT O.CXTEST
DATE
/* COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
/* LOAD THE PROGRAM
SEG
VLOAD #FTN.TEST
LO B_FTN.TEST
LI
SA
MAP M_LOADTEST 7
MAP M_UNSATISFIED 3
QU
CX -Q
/* CX TEST COMPLETED
DATE
COMOUTPUT -END
CX -E
```

When the CX processor is invoked the interactive session at the terminal is:

```
OK, CX CXTEST
GO
YOUR JOB FILE IS CX##24

[CX, REV 16.0]

OK,
```

The command output file created by the CX command file is:

```
OK, DATE
GO

Thursday, June 8, 1978   2:09 PM



OK, /* COMPILE THE PROGRAM IN 64V MODE
FTN FTN.TEST -64V
GO
0000 ERRORS [<.MAIN.>FTN-REV16]

OK, /* LOAD THE PROGRAM
SEG
GO
# VLOAD #FTN.TEST
$ LO B_FTN.TEST
$ LI
LOAD COMPLETE
$ SA
$ MAP M_LOADTEST 7
$ MAP M_UNSATISFIED 3
$ QU

OK, /*
CX -Q
GO
CX JOB QUEUE LISTING   78/06/08   2:09 PM

   FILE    OWNER     ID        DATE/TIME
  ------  ------   ------     -----------

* CX##25   TEKMAN   CXTEST     78/06/08   2:08 PM

[CX, REV 16]

OK, /* CX TEST COMPLETED
DATE
GO

Thursday, June 8, 1978   2:09 PM

OK, COMOUTPUT -END
```

(* denotes execution)

MAGNETIC TAPE UTILITIES

The Prime magnetic tape utilities allow the duplication of magnetic tapes, the transfer of files from disk to tape and vice-versa, and the transfer and translation of tapes in non-Prime format to and from PRIMOS disk files.

Duplicating magnetic tapes

The following utilities are available for duplicating magnetic tapes:

MAGNET

- Copying from tape to tape.
- Translating from EBCDIC or BCD to ASCII.
- Copying binary files.

MAGSAV

- Archiving Prime-format files, directory-trees or disk volumes to tape.

MAGRST

- Restoring Prime-format files, directory-trees or disk volumes from tape.

Copying tapes with MAGNET: If there are two tape drives at the programmer's disposal, the COPY option of MAGNET can be used to generate duplicates of magnetic tapes. This option copies one tape directly to another. The MAGNET utility may be used for tapes in Prime or non-Prime format.

The essential steps in the copy procedure are:

1. Assign two magnetic tape drive units from terminal
2. Mount the FROM tape (original) and TO tape (blank) on their respective drive units.
3. Use COPY option of MAGNET: supply FROM and TO tape unit numbers and starting file number and number of files to be copied, as requested.
4. Dismount both tapes and unassign tape drives when EOT (end of tape) message is returned.

For information on MAGNET, see Reference Guide, PRIMOS Commands.

Copying tapes with MAGRST/MAGSAV: When only one tape drive is available, the MAGSAV/MAGRST utilities can be used to duplicate tapes as follows:

1. Assign a tape drive unit from the terminal.
2. Mount FROM (original) tape on drive unit.
3. Copy tape to files on disk using MAGRST.
4. Remove FROM tape and replace with TO (blank) tape on drive unit.
5. Transfer files from disk to TO tape using MAGSAV.
6. Dismount tape and unassign drive unit from terminal.

A summary of MAGRST and MAGSAV dialogues follows.

MAGRST dialogue summary

The Magnetic Tape Restore Utility restores files, directory, trees and partitions from a magnetic tape (7- or 9-track) to a disk.

The command format is:

MAGRST [-7TRK] *(option specifies 7-track tape: default = 9)*

The MAGRST utility asks the user a series of questions which are summarized, along with appropriate response, below.

Question	**Response**
TAPE UNIT:	Specify number from 0–7.
ENTER LOGICAL TAPE #	Specify number from 1 to n: tape is rewound and positioned. Enter 0 if tape already rewound and positioned.
READY TO RESTORE	Enter YES to restore entire tape: NO causes request for new tape drive number and logical tape number. PARTIAL restores part of tape as defined by the following: **$I [filename] n** – print index to **n** levels to terminal or optional filename. **NW [n]** – print **n-level** index at terminal but do not update disk.
TREENAME:	Requested when PARTIAL restore is specified. Enter pathname(s) for file(s) to be restored.

MAGSAV dialogue summary

The Magnetic Tape Save Utility writes PRIMOS files from disk to a 7- or 9-track magnetic tape. Several options may be specified with the MAGSAV command:

-LONG	Sets record size to 1024-words (Default = 512). Useful for long files.
-UPDT	Indicates update. DUMPED switch set for files and directories saved from disk to tape.
-INC	Indicates incremental dump. Only files and directories with DUMPED switch set to 0 will be saved. (Default = save all)

The MAGSAV dialogue is summarized below. Advisable user responses are indicated.

Question	**Response**
TAPE UNIT:	Specify physical tape unit number from 0–7.
ENTER LOGICAL TAPE #:	Specify number from 1 to n: rewinds and positions tape. Specify 0 if tape is already rewound and positioned.
TAPE NAME:	Positions tape. Specify 0 if tape is already rewound and positioned.
DATE:	Specify date in format: mm dd yy. Default (CARRIAGE RETURN only) is system date.
REV. NO.	Enter appropriate REV. number.
NAME:	Possible responses include: **$A** – change home UFD **$Q** – terminate logical tape and return to PRIMOS **$R** – do $Q and rewind tape **$I [filename] n** – saves index to **n** levels **MFD** – save entire disk * – save current directory

USING PRIMOS WITH NETWORKS

Many Prime installations contain two or more processors connected in a network—a combination of communications hardware and PRIMOS software called PRIMENET. On a system using PRIMENET, the following operations are possible:

- LOGIN to a UFD on a remote system and use that CPU for 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; the local CPU does the processing.)
- Make sure a spool file is printed on the local spool queue (if more than one processor is running a spool queue).

In a network, the processor the user terminal is connected to is the **local** processor, while all other processors are considered **remote**. Each processor in the system is assigned a **nodename** during system configuration. Disks connected to remote processors are identified by local logical disk numbers. (These are assigned by the local system operator during system configuration.) To determine the nodename and logical disk numbers for remote processors use the STATUS command (described later in this section).

For more information on the inner workings of PRIMENET, see the System Administrator's Guide. PRIMENET also supports network-primitive subroutine calls for program-level communication between processes running on different processors. These subroutines are described in Reference Guide, PRIMOS Subroutines.

Remote login

The LOGIN command accepts a nodename argument that enables login 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.

Network status

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
```

```
RING  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) within the pathname of an ATTACH command, as in:

```
ATTACH <3>JONES
```

which attaches to a UFD=JONES on logical disk number 3.

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 the local spooler, use the -HOME argument in the SPOOL command:

SPOOL filename [-HOME]

FILE COPYING, DELETING, AND LISTING (FUTIL)

FUTIL is a file utility command for copying, deleting, and listing files and directories. FUTIL is most often used for copying files and directories from one directory to another. It is also useful for deleting groups of files and entire directories. Its list option allows the user to examine file and directory properties and to keep track of the contents of directories involved in the copy or delete processes. FUTIL allows operations on files within user file directories (UFDs) and segment directories.

Invoking FUTIL

To invoke FUTIL, type FUTIL. When ready, FUTIL prints the prompt character >, and waits for a command string from the user terminal. FUTIL accepts either upper or lower case input, but passwords must be entered *exactly* as they have been created, i.e., in UPPER CASE for almost all instances. (Most other commands will convert passwords to upper case before attempting the match. FUTIL does not.) To abort long operations (such as LISTF), type BREAK, and restart FUTIL by typing S 1000.

To use FUTIL, type a command followed by a carriage return, and *wait* for the prompt character before issuing the next command. The erase (") and kill (?) characters are supported in both command and subcommand lines.

FUTIL commands

Although only the major FUTIL commands are discussed in detail here, the following figures illustrate the function of all available FUTIL commands. Figure 10-1 is an overview of all FUTIL commands. Figure 10-2 outlines the COPY, DELETE and PROTECtion commands and how they operate on files and directories. A typical FROM and TO directory outline is included to show how commands move files and directories from one location to another.

The following are examples of the most commonly used FUTIL commands. An overview of FUTIL commands appears at the end of this section.

For complete details on all FUTIL commands, listed at the end of this section, see Reference Guide, PRIMOS Commands.

Copying

FUTIL provides several commands which allow the user to copy files, and formats are listed below:

COPY	Copies files (as many as will fit on line).
TRECPY	Copies directory trees.
UFDCPY	Copies entire UFD structure (complete with all files).
TO	Specifies directory to which file(s) or directories are to be copied. Accepts a pathname. Default is home directory.
FROM	Specifies directory from which files or directories are to be copied. Accepts a pathname. Default is home directory.

The general format of these commands are:

COPY pathname [new-name], [pathname new-name] . . .

TRECPY pathname

UFDCPY

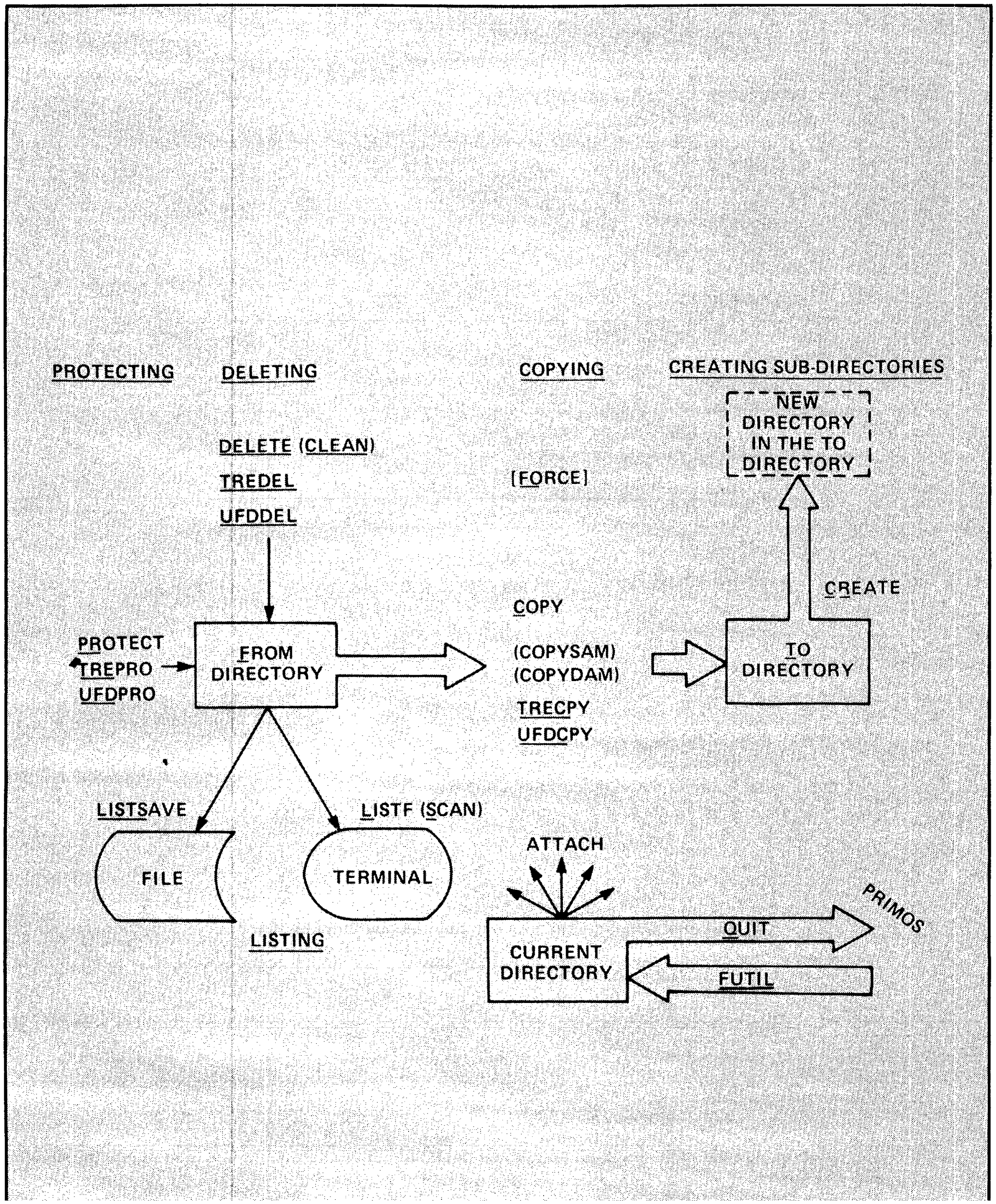
To copy a file, the user must have read access rights. The name of a file may be changed by indicating the desired new name immediately after the current name has been specified. Filename pairs are separated by commas on the command line.

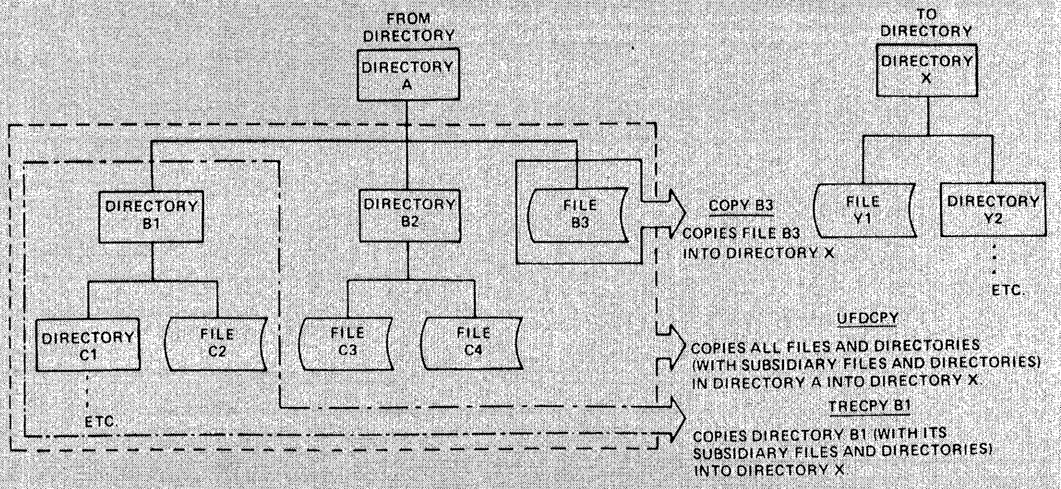
Example: The tree structures in figure 10-3 illustrates the positions of various files and directories which will be copied from one point to another in the following examples.

Situation 1: Suppose we want to copy the files HITS and MISSES from the directory NAUTILUS, into our current directory, SECRETS. We are currently attached to SECRETS and have also set it as home. The pathname of SECRETS is represented as follows: <*> SECRETS. In pathnames, <*> represents the current disk. In this case, it represents disk 2. This pathname can also be represented as <MONITOR>SECRETS. MONITOR is the volume-name of the logical device, whereas 2 is the volume-number. Similarly, in figure 10-3, the volume-name and number of the other logical disk are 1 and NAVY respectively. The volume-name and number can be used interchangeably in a pathname, and both appear in the following examples. Any directory subordinate to SECRETS would be described by a relative pathname, as in, *>DOMESTIC. In relative pathnames, the use of * indicates the home directory.

We do the following:

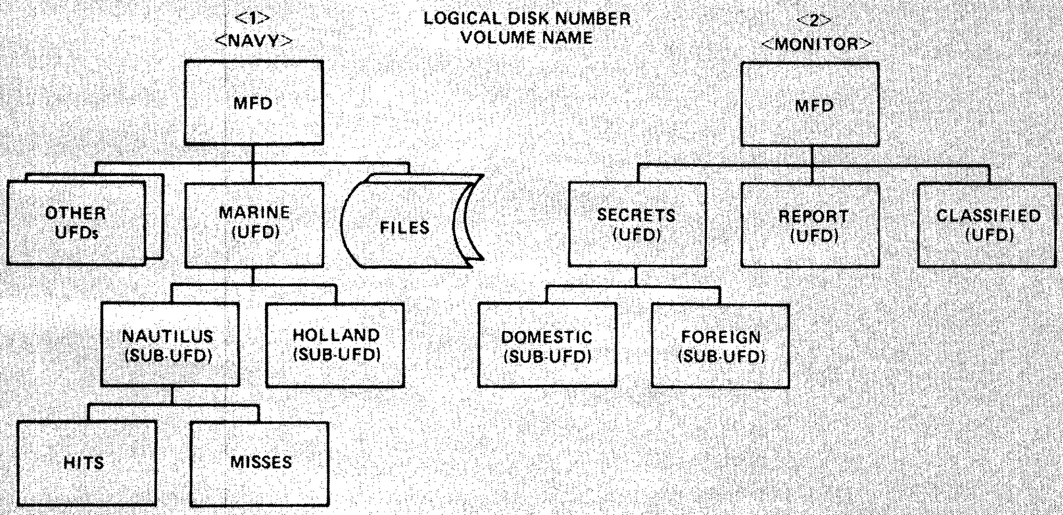
1. Invoke FUTIL.
2. Define the FROM directory.
3. Define the files to be copied and indicate new filenames (optional).


PROTECTING
DELETING
COPYING
CREATING SUB-DIRECTORIES
NEW DIRECTORY IN THE TO DIRECTORY
DELETE (CLEAN)
TREDEL
UFDDEL
[FORCE]
CREATE
COPY
PROTECT
TREPRO
UFDPRO
FROM DIRECTORY
(COPYSAM)
(COPYDAM)
TRECPY
UFDCPY
TO DIRECTORY
LISTSAVE
LISTF (SCAN)
FILE
TERMINAL
LISTING
ATTACH
PRIMOS
QUIT
CURRENT DIRECTORY
FUTIL


FROM DIRECTORY
DIRECTORY A
TO DIRECTORY
DIRECTORY X
DIRECTORY B1
DIRECTORY B2
FILE B3
COPY B3
COPIES FILE B3 INTO DIRECTORY X
FILE Y1
DIRECTORY Y2
ETC.
DIRECTORY C1
FILE C2
FILE C3
FILE C4
UFDCPY
COPIES ALL FILES AND DIRECTORIES (WITH SUBSIDIARY FILES AND DIRECTORIES) IN DIRECTORY A INTO DIRECTORY X.
ETC.
TRECPY B1
COPIES DIRECTORY B1 (WITH ITS SUBSIDIARY FILES AND DIRECTORIES) INTO DIRECTORY X.

COPYING

COPY B3	COPIES FILE B3 INTO DIRECTORY X
TRECPY B1	COPIES DIRECTORY B1 (WITH ITS SUBSIDIARY FILES AND DIRECTORIES) INTO DIRECTORY X
UFDCPY	COPIES ALL FILES AND DIRECTORIES (WITH SUBSIDIARY FILES AND DIRECTORIES) IN DIRECTORY A INTO DIRECTORY X


<1>
<NAVY>
LOGICAL DISK NUMBER
VOLUME NAME
<2>
<MONITOR>
MFD
MFD
OTHER UFDs
MARINE (UFD)
FILES
SECRETS (UFD)
REPORT (UFD)
CLASSIFIED (UFD)
NAUTILUS (SUB-UFD)
HOLLAND (SUB-UFD)
DOMESTIC (SUB-UFD)
FOREIGN (SUB-UFD)
HITS
MISSES

The FUTIL dialogue is as follows:

```
OK, FUTIL
GO
> FROM <1>MARINE>NAUTILUS
> COPY HITS, MISSES ZEROES
> QUIT
OK,
```

The files HITS and ZEROES (formerly MISSES) are now in our home directory SECRETS, as well as in the FROM directory NAUTILUS. (The file MISSES is not renamed in the FROM directory.)

Situation 2: Suppose we want to copy all the contents of the directory HOLLAND to another directory CLASSIFIED, on the current disk. The files and directories contained in HOLLAND are called a directory tree. The FUTIL dialogue would be as follows:

```
OK, FUTIL
GO
> FROM <1>MARINE
> TO <*>CLASSIFIED
> TRECPY HOLLAND
```

This copies the directory HOLLAND (with its subordinate files and directories) to the directory CLASSIFIED. the <*> indicates the current disk. HOLLAND is now a subdirectory in CLASSIFIED.

Situation 3: Suppose we wish to copy the entire directory tree MARINE into the UFD REPORTS. The FUTIL dialogue would be:

```
OK, FUTIL
GO
> FROM <NAVY>MARINE
> TO <MONITOR>REPORTS
> UFDCPY
> QUIT
```

The entire batch of files and directories listed under the UFD MARINE are now listed as a subdirectory under the UFD REPORTS.

Situation 4: We can also copy files from our home (current) directory to another. It is not necessary to specify a FROM name. Simply specify the directory TO which the files are to be copied. If we want to copy the directory REPORTS to the directory MARINE, the following dialogue results. If the files to be copied are located in the home directory, the FROM name need not be defined. When the TO directory alone is defined, FUTIL assumes the FROM directory to be the home directory. If the FROM directory specification is omitted, the home directory is assumed, and FUTIL searches for the files to be copied in the home directory. This is the default. Note that both a TO and FROM directory must be given in this case. REPORTS is not our home or current directory.

```
OK, FUTIL
GO
> FROM <*>REPORTS
> TO <1>MARINE
> UFDCPY
> QUIT
```

Deleting files

Commands for deleting files, directory trees and UFDs are:

DELETE	Deletes specified files from FROM directory
TREDEL	Deletes specified directory trees from FROM directory.
UFDDEL	Deletes entire specified UFD.

The user must have read, write, and delete/truncate access rights to delete any file.

Examples:

Situation 1: In order to delete the file HITS from the subUFD NAUTILUS, the following dialogue could be used:

```
OK, FUTIL
GO
> FROM <NAVY>MARINE>NAUTILUS
> DELETE HITS
> QUIT
```

Situation 2: If we wanted to delete the directory tree rooted in the subUFD HOLLAND, we would do the following:

```
OK, FUTIL
GO
> FROM <1>MARINE
> TREDEL HOLLAND
> QUIT
OK,
```

This deletes the directory HOLLAND and its entry in MARINE.

Situation 3: To delete the contents of CLASSIFIED appearing on our current disk, 2, the following dialogue could be implemented:

```
OK, FUTIL
GO
> FROM <*>CLASSIFIED
> UFDDEL
> QUIT
OK,
```

This effectively deletes the entire UFD CLASSIFIED and all of its subordinate directories and files.

Listing contents of a directory

The LISTF command in FUTIL displays a list of all the files and directories in the FROM directory. It also displays the FROM directory pathname and the TO directory pathname (default). The various options of the LISTF command provide information on all the files contained in the FROM directory.

FUTIL Command Summary

ATTACH pathname

Changes working directory to **pathname.**

CLEAN prefix [level]

Deletes files beginning with **prefix,** for indicated number of **levels** (default=1).

COPY from-name [to-name] [,from-name [to-name]] . . .

Copies named files from FROM directory to TO directory. If **to-names** are omitted, copies have same names as originals.

COPY (from-position) [(to-position)]

Copies from one segment directory to another. If **to-position** is omitted, copy goes to same position as original.

COPYDAM

Same as COPY but sets file type of copy to DAM.

COPYSAM

Same as COPY but sets file type of copy to SAM.

CREATE directory [owner-password [non-owner-password]]

Creates **directory** in current TO directory (with optional passwords).

$$\textbf{DELETE} \left\{ \begin{array}{l} \textbf{file-a [file-b]} \ldots \\ \textbf{(position-a) [(position-b)]} \ldots \end{array} \right\}$$

Deletes from FROM directory, named files or, in segment directories, deletes files at specified positions.

$$\textbf{FORCE} \left\{ \begin{array}{l} \textbf{ON} \\ \textbf{[OFF]} \end{array} \right\}$$

ON forces read-access rights in FROM directory for LISTF, LISTSAVE, SCAN, UFDCPY, and TRECPY. OFF stops FORCE action (default).

FROM pathname

Defines FROM directory for subsequent commands such as COPY, LISTF, etc.

LISTF [level] [FIRST] [SIZE] [PROTEC] [RWLOCK] [TYPE] [DATE] [PASSWD] [LSTFIL]

Lists files and attributes at terminal (and into optional file called **LSTFIL**).

LISTSAVE filename [level] [FIRST] [SIZE] [PROTEC] [RWLOCK] [TYPE] [DATE] [PASSWD]

Same as LISTF, with the LSTFIL option specified, but writes output to **filename**.

PROTEC filename [owner-access [non-owner-access]]

Sets protection attributes for **filename**.

SCAN filename [level] [FIRST] [LSTFIL] [SIZE] [PROTEC] [RWLOCK] [TYPE] [DATE] [PASSWD]

Searches FROM directory tree for all occurrences of specified **filename** and prints requested attributes.

SRWLOC filename lock-number

Sets per-file read/write lock.

TO pathname

Defines TO directory for subsequent commands such as CREATE and all copying commands.

TRECPY directory-a [directory-b] [,directory-c [directory-d]] . . .

Copies directory tree(s) in FROM directory into TO directory.

TREDEL directory-a [directory-b] . . .

Deletes directory tree(s) in FROM directory.

TREPRO pathname [owner-access [non-owner-access]]

Sets protection rights for directory and contents (default 1 0).

TRESRW pathname lock-number

Sets per-file read/write lock for all files in **pathname**.

UFDCPY

Copies entire FROM directory into TO directory.

UFDDEL

Deletes entire FROM directory.

UFDPRO [owner-access [non-owner-access [level]]]

Sets protection attributes for entire FROM directory.

UFDSRW lock-number n-levels

Sets per-file read/write lock for **n-levels** in FROM directory.

Lock-number	Meaning	Code
0	Use system read/write lock	SYS
1	n readers OR 1 writer	W/NR
2	n readers AND 1 writer	1WNR
3	n readers AND n writers	NWNR

FILE MANIPULATION

PRIMOS provides utilities for comparing, merging, and sorting files.

File comparison

The PRIMOS command CMPF permits the simultaneous comparison of up to five ASCII files of varying lengths. The format is:

CMPF file-1 file-2 [. . . . .file-5] [options]

The first file, **file-1**, is treated as the original file against which the other files are compared. The CMPF command produces output indicating which lines have been added, changed or deleted in the other files.

The options which may be specified are:

-BRIEF	Suppresses the printing of differing lines of text of files being compared. Only identification letters and line numbers are printed.
-MINL number	Sets the minimum **number** of lines that must match after a discrepancy between files is found. Needed in order to resynchronize file comparison. Default = 3 lines.
-REPORT filename	Produces a file with specified **filename**, containing the differences found between compared files (in lieu of displaying them at the terminal during the comparison process).

After a difference between the original file and another specified file has been discovered, CMPF attempts to resynchronize the files for comparison. This occurs only when a certain number of lines match in all the files being compared. The default value is 3, but can be changed in the -MINL option. The comparison process continues until another difference is found.

When line differences are reported, either at the terminal or in a report file, each line from the original file is indicated by the letter A, followed by the line number of the line containing discrepancies. The corresponding lines of other files are indicated in the same manner using letters B through E respectively.

Example: Consider the following two files:

```
FILEA     FILEB

The       The
quick     swift
brown     red
fox       fox
jumps     jumps
over      over
the       the
lazy      dog
dog
```

A CMPF comparison of these two files works as follows:

```
OK, CMPF FILEA FILEB
GO

A2          quick
A3          brown
CHANGED TO
B2          swift
B3          red


A8          lazy
DELETED BEFORE
B8          dog.

COMPARISON FINISHED.
2 DISCREPANCIES FOUND.


OK,
```

Merging text files

The MRGF command merges up to five ASCII files. The format is:

MRGF file-1 [file-2 . . .file-5] outfile [options]

The first file specified is treated as the original file, and it is assumed that changes have been made to this file to produce the other files. Pathnames may be used to specify files to be merged. Unchanged lines of text and nonconflicting changes between files are automatically copied to the output file, **outfile**. When corresponding lines of text in the files differ, the user is asked by the MRGF program to solve the conflicts. This is done via a series of questions to which the user must respond appropriately.

The **options** taken by the MRGF command are similar to those for the CMPF command. There is an additional option, -FORCE, which causes **file-2** to be the preferred file if conflicts exist between several files. No MRGF interactive dialogue will be generated when conflicts arise if the -FORCE option is used. File-2 is assumed 'correct' and the other files forced to comply with it.

Sorting files

The SORT command sorts various file types (default is ASCII), in ascending or descending order. Lower case characters are sorted as upper case characters but are printed out as lower case after being sorted. Information required by the SORT program is the following:

1. INPUT TREENAME—OUTPUT TREENAME
2. NUMBER OF PAIRS OF STARTING AND ENDING COLUMNS (one pair per line, separated by spaces)
3. SPECIFIC DATA TYPE - code entered at end of last line of column keys. Up to 10 keys may be specified.

Codes

'A'	ASCII
'I'	Single Precision Integer
'F'	Single Precision Real
'D'	Double Precision Real
'J'	Double Precision Integer

Default is ASCII

Example:

```
OK, SORT
GO
SORT PROGRAM PARAMETERS ARE:
  INPUT TREE NAME -- OUTPUT TREE NAME FOLLOWED BY
  NUMBER OF PAIRS OF STARTING AND ENDING COLUMNS.
INFILE OUTFILE 3
  INPUT PAIRS OF STARTING AND ENDING COLUMNS
  ONE PAIR PER LINE--SEPARATED BY A SPACE.
  FOR REVERSE SORTING ENTER "R" AFTER DESIRED
  ENDING COLUMN--SEPARATED BY A SPACE.
  FOR A SPECIFIC DATA TYPE ENTER THE PROPER CODE
  AT THE END OF THE LINE--SEPARATED BY A SPACE.
     "A" - ASCII
     "I" - SINGLE PRECISION INTEGER
     "F" - SINGLE PRECISION REAL
     "D" - DOUBLE PRECISION REAL
     "J" - DOUBLE PRECISION INTEGER
  DEFAULT IS ASCII.
1 5
15 25
30 35

BEGINNING SORT



 PASSES         2            ITEMS        401

[SORT-REV16.0]

OK,
```

Options

Several options may be specified with the SORT command. BRIEF suppresses SORT program messages. SPACE eliminates blank lines in the output. MERGE allows the merger of up to ten files. The R option (for reverse sorting) is placed after the column pairs which are to be sorted in descending order.

SETTING TERMINAL CHARACTERISTICS

Terminal characteristics may be set with the TERM command. These characteristics remain in effect until you reset them or until you log out. The commonly used TERM options are listed below. Typing TERM with no options returns the full list of TERM options available. The format is:

TERM options

The options are:

-ERASE character	Sets user's choice of erase **character** in place of the default, ".
-KILL character	Sets user's choice of kill **character** in place of default, ?.
-XOFF	Enables X-OFF/X-ON feature, which allows programs to halt without returning to PRIMOS command level. Programs may be resumed at point of halt by typing CONTROL-Q. Programs are halted by typing CONTROL-S. Also sets terminal to full duplex (default value.)
-NOXOFF	Disables X-OFF/X-ON feature (default).
-DISPLAY	Returns list of currently set TERM characters. Also displays current Duplex, Break and X-ON/X-OFF status.

11

Extended segmented program techniques

ADVANCED FEATURES OF LOAD SUBPROCESSOR

Relative segment 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 SEG's loader for this purpose: relative segment assignment and absolute segment assignment.

Relative segment assignment assigns reference numbers to SEG's default segments; these reference numbers 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 0 1 2
```

The numbers 1, 2, and 3 are relative segment reference numbers. The 0 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 0 0 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.

Another form of the COMMON command:

COMMON REL segno

allows the user to establish a reference number for segments into which COMMON will be loaded. **segno** is the segment number into which COMMON will be loaded. It is always a small octal number.

Example:

```
CO REL 1
```

If data segment was assigned a relative value of 1, 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.)

Load placement control

The Loader's family of Load commands, LOAD, LIBRARY, and RL, has optional numeric arguments for load placement control:

LOAD filename [addr psegno lsegno]

LIBRARY [filename] [addr psegno lsegno]

RL filename [addr psegno lsegno]

addr is the starting point for procedure in the segment specified by **psegno**. If addr is 0, the current PBRK for that segment is used (TOP+1). Users ordinarily specify 0 for this parameter. psegno is a relative segment assignment number to be used in loading procedure (the code). **lsegno** is 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 0, 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 0 1 1
LO B_SUB1 0 1 1
     .
     .
     .
LO B_SUBLAST 0 1 1
LI 0 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.

Implicit parameter assignment: the D/ prefix

The D/ modifier tells the loader to use the same numeric parameters as were used for the preceding load family command. The example in the preceding paragraph is equivalent to:

```
CO REL 3
LO B_MAIN 0 1 1
D/LO B_SUB1
    .
    .
    .
D/LO B_SUBLAST
D/LI
```

The commands:

```
LO B_MAIN
LO B_SUB1 0 1 1
D/LO  B_SUB2     (or LO B_SUB2 0 1 1)
LI
```

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

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.

Specific segment assignment: the S/ prefix

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 [pathname] addr psegno lsegno

xx is LO, LI, RL, PL, or IL. If LO or RL is used **pathname** is mandatory; if LI is used pathname is optional (omission loads PFTNLB). If PL or IL is used pathname should be omitted.

addr is the starting load address in the procedure segment. An addr of 0 is interpreted as start loading at the current pointer position in the procedure segment. This is the usual value. **psegno** is the procedure segment number. **lsegno** is the data linkage segment number.

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

As with relative segment assignment commands, the segments will be created if they do not already exist. If a segment runs out of room the next segment in sequence is 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 is created as a procedure segment and the Load will proceed 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.

Examples:

```
S/LO B_JUNK 0 2000 4002
```

This loads object file B_JUNK with its procedure beginning at the current load pointer location in segment '2000 and its data linkage areas beginning at the current load pointer in segment '4002. (Previously COMMON was located with a CO ABS command.)

```
S/IL 0 4000 4000
```

This loads 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 0 2000 4000
D/PL
```

is equivalent to

```
S/LO B_BENCH 0 2000 4000
S/PL 0 2000 4000
```

> **CAUTION**
>
> When using the S/ modifier, some of SEG's checking mechanisms are overridden. Therefore, the user must carefully examine the loadmap to make sure there is no inconsistency or confusion. The S/ modifier may *not* be combined with the D/ modifier either as D/S/xx or S/D/xx.

Forceloading (The F/ Modifier)

When a file is loaded, normally only those routines referenced by previously loaded modules (or by routines in the library) are selected. 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/ modifier as in:

F/xx [filename] [addr psegno lsegno] *Form 1*

or

F/S/xx [filename] [addr psegno lsegno] *Form 2*

xx is one of the loading commands, LO, LI, RL, PL, or IL. **filename** is the filename (or pathname) 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.

Form 1 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.

Examples:

F/LOB_THINGS	Forceload all modules in B_THINGS in default segment.
F/LI	Forceload *all* the FORTRAN library in default segments

Form 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
```

This forceloads *all* of the procedure of the FORTRAN library PFTNLB beginning at location '4000 in segment '2000 with linkages area in segment '4002.

S/F/xx is identical to F/S/xx, and the D/ prefix may be combined with F/.

```
S/LO B_BENCH 0 2001 4002
F/S/PL 0 2001 4002
```

is equivalent to

```
S/LO B_BENCH 0 2001 4002
F/D/PL
```

Relocating uninitialized COMMON

COMMON blocks which are not initialized by a DATA statement or a BLOCK DATA subprogram may be relocated in the load with the SYMBOL command. This process reduces the number of segments used by the load and, therefore, decreases the time to restore prior to execution. The format is:

SYMBOL [sname] segno addr

sname is the symbol name; here, it is the name of the COMMON block. **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.

Examples:

```
SY CYMBAL 4001 12000
```

Locates the COMMON block 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.

Example of Use: A program BENCH has 3 large (over 33K) COMMON blocks. It is desired to reduce time required 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. 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.

Initializing the load

The load subprocessor's INITIALIZE 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** (pathname is allowed)

Replacing modules

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

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. However, the old module still occupies space in the runfile. Overuse of the RL command may significantly increase runfile size as well as restore and execution times.

Example: RL B_MODULE

This 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.

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.

Altering stack size

The STACK command changes the amount of space required for the stack. The size parameter is the minimum required stack size in words (octal).

Example: ST 100000

This reserves 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) subprocessor.

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 (Modification sub-processor) and by the SPLIT command (Loader sub-processor). SK and SPLIT perform their normal function if no extension parameters are supplied.

When extension stack segments are specified, 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. For complete details, see Reference Guide, LOAD and SEG.

THE MODIFICATION SUBPROCESSOR

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

MODIFY [filename] or **SAVE [filename]**

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

The command invokes the modification subprocessor, which 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.

SHARED CODE

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 administrator the segment numbers which are to be used for the particular program being loaded. Public shared segments are a large but finite resource. Their allocation will be made only for those programs which will benefit by being loaded as shared procedure. Currently, segments '2000 to '2017 are reserved for Prime-supplied shared subsystems (Shared Editor, FORMS, etc.). Segments '2030 to '2037 are available as public shared segments.

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 the PRIMOS SHARE command. This is done by the System Operator at the Supervisor Terminal.

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.

Since the main program is labelled as subroutine MAIN, no other subroutine may have that name. There is no subroutine or function named MAIN 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 producing code to be loaded with SEG. If an array or COMMON block exceeds 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. Details of over 64K COMMON are treated elsewhere in this section. Extension stacks, which may be necessary in certain cases of recursive subprograms or if programs are chained, are also discussed in this section.

Loading

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

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

```
OK, SEG                 Invoke SEG.
#VL #BENCH              Establish runfile and access Loader.
$LO B_BENCH             Load main program.
$LI                     Load FORTRAN library.
LOAD COMPLETE           Load is complete.
$SA                     Save result.
$MA MAPFIL              Generate a map in file MAPFIL to be examined.
$QU                     Return to PRIMOS.
OK,
```

At this point the program will be executed and, if necessary, debugged. The number of segments used can be decreased by moving the location 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.
LOAD COMPLETE              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.

Loading for shared code

Loading for shared code requires the separation of the procedure frame from the linkage frames. This capability exists in the advanced 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.

SEG's 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=CMDNC0. Splitting is accomplished by the SPLIT command, which breaks a segment into procedure (lower) and data (upper) portions. This operation conserves segments. It also allows the loading of RUNIT as an aid to creating shared programs:

SPLIT segno addr *Form 1*

or

SPLIT addr *Form 2*

segno is the absolute octal segment number. addr is the location of the split in the segment. **addr** must be a multiple of '4000.

Form 1 splits the segment into procedure and data portions to decrease number of segments used. Example:

```
SP 4000 10000
```

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

Form 2 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.

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.

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 [runfile]

runfile is the pathname 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: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.

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 0 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 (renamed if desired) from PRIMOS command level either from CMDNC0 or by a PRIMOS RESUME command.

Example:

Programmer has been assigned segment '2031 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 0 2031 4000   Load the procedure portion of the user
                             program into segment '2031; load link
                             frames into '4000.
$ D/PL                       Load the pure FORTRAN library with the
                             same parameters.
$ S/IL 0 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 BE2031
# 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 supervisor terminal by the System Operator. See System Administrator's Guide for details.

COMMON BLOCKS OVER 64K WORDS LONG

The size of COMMON blocks and the arrays within them are limited only by the number of segments available to the user. A total of 256 segments are available for assignment to users. 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

Any named COMMON or blank COMMON may be over 64K; no special syntax is required. The only indication that a COMMON block is over 64K is in the concordance, generated with the compiler's -XREFL option. 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.

Arrays in a COMMON block over 64K are treated as if they spanned a segment boundary regardless of their size. Code normally generated for array references will not work for these arrays. Programs (and subprograms) referencing these arrays 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 (see below).

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).

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. This method cannot be used when there are dummy arguments or COMMON dimensions. Example:

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

2. Compile the subprogram with the -BIG option. All dummy argument arrays will be treated as arrays spanning segment boundaries. -BIG also forces compilation in 64V mode. Example:

   ```
   FTN SUB -BIG
   ```

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.

- An array may span segment boundaries, but 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 0 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(1000,44)

Number of Words = 1000 x 44 x 2 = 88000

12

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.
- 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).

Requirements

The MIDAS system requires the 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 Figure 12-1).

- 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 deleted in part or full (KIDDEL).

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 or restructure 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.

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.

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. Certain restrictions exist in modifying parameters, especially in converting to long indices.

An example of the template creation dialogue is shown in Figure 12-2.

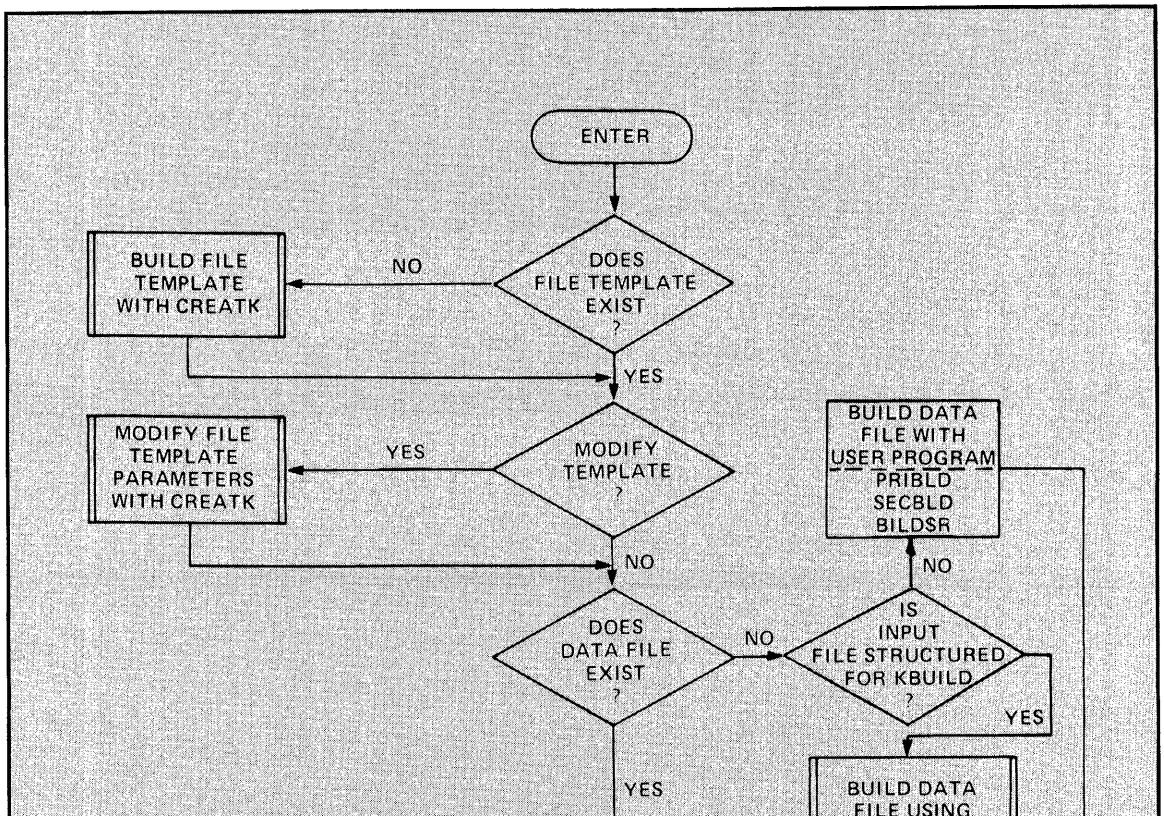
Building the data sub-file

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 the last milestone report. Milestone reports are also generated at the start and end of file processing.


ENTER
BUILD FILE TEMPLATE WITH CREATK
NO
DOES FILE TEMPLATE EXIST ?
YES
MODIFY FILE TEMPLATE PARAMETERS WITH CREATK
YES
MODIFY TEMPLATE ?
NO
BUILD DATA FILE WITH USER PROGRAM
PRIBLD
SECBLD
BILD$R
NO
DOES DATA FILE EXIST ?
NO
IS INPUT FILE STRUCTURED FOR KBUILD ?
YES
YES
BUILD DATA FILE USING

```
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 = :  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.?  (CR)                            RETURN indicates no more indices

OK,
```

Figure 12-2. Sample of CREATK dialogue

The MIDAS file created by KBUILD has fixed-length records and completely sorted indexes. The user may alter these records to variable-length data records by the use of CREATK.

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.

```
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 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 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 index 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 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 keys. 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 12-3. 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

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, or
- the entire file.

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 FORTRAN declarations and statements by the FORTRAN DML Preprocessor. The output of the preprocessor is the source input for the FORTRAN compiler.

See: Reference Guide For DBMS Schema Data Definition Language (DDL), and the FORTRAN Reference Guide For DBMS.

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 the 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.

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
```

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 the user 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.

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

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 Assembly Language Programmer's Guide

13

Optimization and other helpful hints

INTRODUCTION

This section presents some programming hints for improving the performance of FORTRAN routines. Some of them are merely reminders of good coding practice; others take advantage of implementation techniques in the FTN compiler. All offer some speedup in program execution.

DO LOOPS

1. Remove **invariant expressions** from DO loops. For example,

```
      DO 10 I = 1, 50
         A = 3.01
             .
             .
   10 CONTINUE
```

should be changed to:

```
      A = 3.01

      DO 10 I = 1, 50
             .
             .
   10 CONTINUE
```

2. Optimize unnecessary **subscript calculations**. The first source code sequence is more efficient than the second one below.

```
      SUM = 0

      DO 10 I = 1, 90
         SUM = SUM + ARRAY (I)
   10 CONTINUE

      ARRAY(N) = ARRAY(N) + SUM

      - - - - - - - - - - - - -

      DO 10 I = 1, 90
         ARRAY(N) = ARRAY(N) + ARRAY (I)
   10 CONTINUE
```

3. Minimize **DO Loop Setup Time**. When nesting DO Loops (also any hand-coded control structures), order the loops so that the fewer iteration count loops are on the outside, and the higher iteration count loops are on the inside.

Example 1:

```
      DO 20 I = 1, 5
        DO 10 J = 1, 100
              .
              .
              .
            loop-body
              .
              .
              .
 10   CONTINUE
 20   CONTINUE
```

Example 2:

```
      DO 20 J = 1, 100
        DO 10 I = 1, 5
              .
              .
              .
            loop-body
              .
              .
              .
 10   CONTINUE
 20   CONTINUE
```

Example 1 is the preferred control structure for the following reasons. The execution time for a DO loop consists of three major items:

1. Setup time (Ts)—the time required to initialize the index.
2. Increment and test time (Ti)—the time taken *each* time the flow of control hits the bottom of the loop.
3. Time to execute the body of the loop (Tb).

For examples 1 and 2 above, the time required to execute the DO 10 loops is:

1. Time(1) = 5 × (Ts + 100Ti + 100Tb)
2. Time(2) = 100 × (Ts + 5Ti + 5Tb)

which yields:

1. Time(1) = 5Ts + 500Ti + 500Tb
2. Time(2) = 100Ts + 500Ti + 500Tb

Time (1) is smaller, making it the preferred structure.

4. Use CONTINUE Statements. Always end DO loops with a CONTINUE statement. This is a special case of statement number usage, described below.

STATEMENT NUMBERS

Eliminate all unnecessary statement numbers, i.e., those that program control will never access. Most optimizations are performed between statement numbers; therefore the fewer statement numbers, the more optimization possible. For example,

```
  IF (I .EQ. O) J = K
```

can be more efficient and is easier to read than:

```
     IF (I .NE. O) GOTO 10
     J = K
  10 next-statement
```

MULTI-DIMENSIONED ARRAYS

Reference memory as sequentially as possible. For multi-dimensioned arrays, the leftmost subscript varies the fastest in FORTRAN, so when addressing large portions of an array, paging and working set can be significantly reduced by indexing the leftmost subscript the fastest (e.g., in the innermost loop). Thus,

```
     DO 20 I = 1, 100
        DO 10 J = 1, 100
          ARRAY (J, I) = 3.0
  10      CONTINUE
  20    CONTINUE
```

is more efficient than accessing the structure as ARRAY (I, J) = 3.0.

If the program can be coded CLEANLY without multiple-dimension structures, memory addressing can be more efficient. For each dimension over one, this saves one ‘multiply’ per effective address calculation; i.e., number-of-multiplies = number-of-dimensions – 1. For instance, the example above could be written as:

```
     DIMENSION JUNKARRAY  (1)
     EQUIVALENCE (ARRAY(1,1), JUNKARRAY(1))

     DO 10 I = 1, 10000
        JUNKARRAY(1) = 3.0
  10 CONTINUE
```

saving considerable CPU time.

LOAD SEQUENCE AND MEMORY ALLOCATION

Paging time can be significantly reduced by ordering routines by frequency of use (rather than, say, alphabetically). The Main routine must always be loaded first for LOAD or SEG to work properly.

A suitable loading scheme would allocate memory as:

```
MAIN
.
.
END

.
.      most common subroutines
.


.
.      occasionally used subroutines
.


.
.      infrequently used subroutines
.
```

Paged memory fragmentation can be reduced by loading routines on page boundaries using SEG's P/LO command.

In subroutine libraries, the top down tree structure must be preserved if 'reset force load' is in use.

This ordering method may also be used to order COMMON blocks in memory by frequency of use. See Section 11 for details.

FUNCTION CALLS

Eliminate **redundant function calls** with equal arguments. For example:

```
TEMP = SIN(X)
A = TEMP * TEMP
```

is *significantly* faster than:

```
A = SIN(X) * SIN(X)
```

Make sure that the function has no side effects which might modify the argument(s) or anything else in the environment.

V-MODE VS. R-MODE COMPILATION

In almost all cases, V-mode code executes faster than R-mode code. If a V-mode program plus data is less than 64K words, and the routine is *not* to be shared, use the MIX command of SEG to compact the memory image.

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.

IF STATEMENTS

Minimize **compound logical connectives** within an IF statement when possible. For example,

```
IF (A.EQ.B .OR. C.EQ.D) GOTO 10
```

has the same effect as, but is *slower* than:

```
IF (A.EQ.B) GOTO 10
IF (C.EQ.D) GOTO 10
```

INPUT/OUTPUT

Significant speed improvement in raw data transfers can be achieved by using the equivalent IOCS or file system routine instead of formatted input/output. For example,

```
      INTEGER TEXT(40)
      READ (5, 20, END = 99) TEXT
   20 FORMAT(40A2)
```

is *slower* than

```
INTEGER TEXT(40)
CALL RDASC(5, TEXT, 40, $99)
```

but the *fastest* yet is. . .

```
 INTEGER TEXT(40), CODE
 CALL RDLIN$(1, TEXT, 40, CODE)

 IF(CODE .NE. 0) /* Any error?
X GOTO 99        /*Yes, go process error.
```

There are also routines for reading/writing octal, decimal, and one-unit hexadecimal numbers from/to the terminal. For example, CALL TIHEX(N), will read a hexadecimal integer from the terminal into the 16-bit integer named N. For printing out text efficiently, use the TNOU/TNOUA routines. See the Reference Guide, PRIMOS Subroutines for more specific information about these lower level routines.

STATEMENT SEQUENCE

The compiler can do register tracking, but cannot reorder statements. For example, given the sequence:

```
A = B
X = Y
R = B
```

the generated code is

```
LDA B
STA A
LDA Y        (6 instructions long)
STA X
LDA B
STA R
```

If the source had been rearranged to

```
A = B
R = B
X = Y
```

the generated code is reduced to:

```
LDA B
STA A
STA R        (5 instructions long)
LDA Y
STA X
```

PARAMETER STATEMENTS

Initializing named constants via PARAMETER statements allows the compiler to perform constant-folding optimizations. The compiler does not fold normal variables initialized by DATA statements into constants.

INEFFICIENT LIBRARY CALLS

Some of the library routines are not optimized for time-critical operations. The get and store character routines (GCHR$A, etc.) are convenient, but comparatively slow. Some of the APPLIB routines are by definition slow. Avoid using the MAX and MIN calls especially in V-mode. It may be more efficient to code it yourself.

Remember the 80/20 rule, which states: "80 percent of a program's time is spent in 20 percent of the code" (*exact numbers subject to debate*). Therefore, standard library routines are adequate in the non-time-critical 80 percent of the program.

STATEMENT FUNCTIONS AND SUBROUTINES

Use statement functions instead of formal FUNCTION subprograms when practical. In V-mode this eliminates a lengthy PCL/PRTN sequence. Try to minimize the number of arguments passed to and from a function or subroutine regardless of whether it is a statement function or a separate function subprogram.

INTEGER DIVIDES

When dividing a *non-negative* integer by a power of two, use the RS (right shift) binary intrinsic function. For example:

```
I = RS(J, 3)
```

Is much faster than:

```
I = J/8
```

LOGICAL VS. ARITHMETIC IF

Logical IFs are preferred to arithmetic IF statements. Many FORTRAN programs have sections which look like:

```
        IF (I - J) 1, 2, 1
1       next-statement
        .
        .
        .
2       some-other-statement
```

A more optimal code sequence would be:

```
        IF (I. EQ. J) GOTO 2
1       next-statement
        .
        .
        .
2       some-other-statement
```

which is also more readable.

USE OF THE COMPILER'S -DYNM OPTION

V-mode programs run faster, better, and cleaner if local variables are placed in the stack through the -DYNM option. These variables are not guaranteed to be valid after a return. For example:

```
      INTEGER COUNT
      DATA COUNT /0/

      IF(COUNT .NE. 12) GOTO 1

      CALL TONL
      COUNT = 0

1        COUNT = COUNT + 1
      some-more-code
      RETURN
      END
```

The above example would *not* work if compiled with the -DYNM option, because the value of COUNT would not be saved after execution of the RETURN statement.

CONCLUSION

These are some of the more common guidelines to keep in mind while programming in Prime FORTRAN. If you keep these ideas in mind while writing, or while 'tweaking' FORTRAN programs, your programs will be generally smaller and faster. Some of these rules are not necessarily permanent. As Prime FORTRAN evolves more and more optimizations, the user will have more freedom to choose coding styles.

Generally it is easier to apply these techniques at initial coding time, as opposed to 'going back and optimizing'. While some of these changes can be done easily with a few Editor tricks, others may require extensive changes to source code. Many other useful examples of good FORTRAN programming practice appear in the following text:

Kernigan and Plaugher, *The Elements of Programming Style*, McGraw-Hill, 1974

REQUEST FOR CONTRIBUTIONS TO THIS SECTION

If you have optimizing techniques in Prime FORTRAN that you would like to share with future readers, please submit them to: Technical Publications, Prime Computer, Inc., 145 Pennsylvania Avenue, Framingham, MA 01701.

4

FORTRAN LANGUAGE REFERENCE

14

FORTRAN language elements

LEGAL CHARACTER SET

The characters allowed in Prime FORTRAN are:

- The 26 upper-case 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.
- The 10 digits: 0,1,2,3,4,5,6,7,8,9.
  Letters and digits together are called alphanumeric characters.
- These 12 special characters:

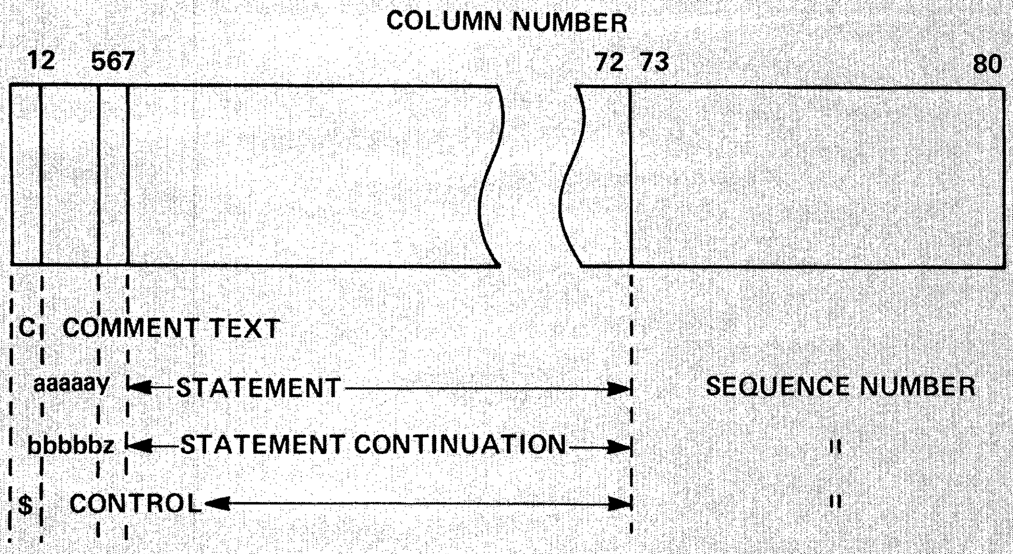
=	equals
'	single quote (apostrophe)
:	colon
+	plus
–	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
$	dollar sign

- Blanks or spaces.

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 can 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. These are three types of lines: Comments, statements (and their continuations), and control statements. (See Figure 14-1.)



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 14-1. 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

See appendix D for details of constant storage.

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	2 x 2	same 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 5O13752 or -3O156

(The O notation is obsolete. It is 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 octal digits plus a sign.

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

Integer constants are treated as short integers unless:

- Their magnitude exceeds 32767 or :177777 (octal).
- 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 5 and 17 for details.)

Long integers may be used in the FORTRAN program anywhere that short integers are used. This includes 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 17 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 2 digits can be printed from 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:

   ```
   23HTHIS IS AN ASCII STRING
   ```

2. The string enclosed in single quotes:

   ```
   'THIS IS AN ASCII STRING'
   ```

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 15).

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 15, 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 (I1,I2,. . .,In).

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.

GENERALIZED SUBSCRIPTS

There is no syntactical limitation on subscript expressions. The FORTRAN compiler allows any integer-valued expression as an array subscript.

Use of generalized subscripts

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

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
```

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. For example, if, 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).

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

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

	P	
Q	.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 generally proceeds from left to right. However, the compiler takes advantage of groupings of elements (in accordance with mathematical rules) and, as a result of this, evaluation may sometimes not be strictly left-to-right (See note below). Expressions within parentheses are evaluated before operations outside the parentheses are performed.

Note

When two elements are combined by an operator, the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The results of different permissible orders of combination even though mathematically identical need not be computationally identical. See: Section 6.4, para. 2, ANSI X3.9–1966

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 14-2. 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 15.

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 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 14-2. Source Program Composition

15

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
READ	Input/Output
REAL	Specification
REAL*4	Specification
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:

1. **Header Statements:**
    - BLOCK DATA
    - FUNCTION
    - SUBROUTINE
2. **Specification Statements:**
    - IMPLICIT
    - mode: COMPEX, LOGICAL, DOUBLE PRECISION, REAL, REAL*4, REAL*8, INTEGER, INTEGER*2, INTEGER*4.
    - PARAMETER
3. **Storage Statements:**
    - COMMON
    - DIMENSION
    - EQUIVALENCE
    - SAVE
4. **External Statements:**
    - CALL
    - EXTERNAL
5. **Data Definition Statements:**
    - DATA
6. **Compilation and Run-Time Control Statements:**
    - FULL LIST
    - LIST
    - NO LIST
    - TRACE
    - $INSERT
7. **Assignment Statements**
8. **Control Statements**
    - ASSIGN
    - CONTINUE
    - DO
    - END
    - GO TO
    - IF

- PAUSE
- RETURN
- STOP

9. **Input/Output Statements:**
    - PRINT
    - READ
    - WRITE
10. **Coding Statements:**
    - DECODE
    - ENCODE
11. **Format Statements:**
    - FORMAT
12. **Device Control Statements:**
    - BACKSPACE
    - ENDFILE
    - REWIND
13. **Functions**
14. **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

[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).

SUBROUTINE statements

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 expressed.

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
```

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 5.

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 5 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=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 FORTRAN 77 standard. 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 or array names, 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 11.

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 FORTRAN 77 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 17.

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 350 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 (9) and the Compiler Section (5).

FULL LIST statement

Causes a listing of subsequent source code with a symbolic listing. Overridden by compiler parameters.

INSERT statement

See **$INSERT**.

LIST statement

Causes a listing of subsequent source code with no symbolic listing. Overridden by compiler parameters.

NO LIST statement

Causes a cessation of subsequent source code listing and of symbolic listing. Overridden by compiler parameters.

FULL LIST, LIST, and NO LIST may be used anywhere in the source program.

Item TRACE statement

TRACE V1, V2, . . . Vn

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.

Area TRACE statement

TRACE n

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 5). It is possible to have the TRACE output written into a file instead of at the user 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 a 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 Section 10.

$INSERT statement

$INSERT insert-file

Insert into the program, at compilation time, the file whose pathname is **insert-file**. 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.
- Parameter definitions, especially for the file management system, applications library, MIDAS, etc.

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 PRECISION
- 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 15-1. 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]

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.

Unconditional GO TO statement

GO TO k

Transfers control to statement labelled k.

Table 15-1. Data Mode Rules for Assignment Statements (A=B)

	To A (left-hand-side)				
FROM B (right-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.
Fi Truncate fractional part and transform integral part to integer

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.

Arithmetic IF statement

IF (e) k1, k2, k3

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 **k2**, and if e>0 (positive), control is transferred to statement labelled **k3**.

Logical IF statement

IF (e) statement

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

RETURN

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 15-2 for list of FORTRAN device units.

Direct access READ and WRITE statements

The FORTRAN compiler and run-time library 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.

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, PRIMOS Subroutines 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** below 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.

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	The FORTRAN I/O unit number. This is the number used in READ and WRITE statements (1=terminal, 2=paper tape punch/reader, etc. (See table 15-2.)
device	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	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	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	An arbitrary name.
size	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.

PRINT statement

PRINT f [,list]

Prints the **list** of elements at the user 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	A list of variables and array names (separated by commas) into which data are read.

Table 15-2. Devices and Their Default 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

r	The long or short integer expression whose value is the record number to be accessed.
f	The statement number of the format specifier (optional).
b	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.

Binary READ statement

READ (u [, END = a] [, ERR = b] list

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.

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.

List-directed READ statement

READ (u,* [, END = a] [, ERR = b]) list

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.

Example 1:

Source line: READ(1,*)A,B,C
Input Data: 151,,2E2
Result: A=151. B is unchanged. C=2.E2

Example 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 a 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:

Example 1:

Source:	INTEGER*2 IBUF(2) READ(1,*) IBUF
Input Data:	'ABC'
Result:	IBUF(1)=AB. IBUF(2)=C.

Example 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	A list of variables and array names (separated by commas) from which data are printed.

Formatted WRITE statement

WRITE (u,f [,ERR=b]) list

Causes data in the **list** to be written out on FORTRAN unit **u** according to the format statement **f**.

Direct-access WRITE statements

WRITE(u'r,f,ERR=b) list *IBM format*

WRITE(u,f,REC=r,ERR=b) list *ANSI format*

u	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 WRITE statements.

r	The long or short integer expression whose value is the record number to be accessed.
f	The statement number of the format specifier (optional).
b	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.

Binary WRITE statement

WRITE (u [,ERR=b]) list

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 device. 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)

Formatted DECODE statement

DECODE (c,f,a[, ERR=sn]) list

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**.

List-directed DECODE statement

DECODE (c, *, a [, ERR=sn]) list

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.
F1, etc.	A format field description.
d	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 15-3 and 15-4 summarize the field descriptors available in Prime FORTRAN, where **n** (positive integer constant) is the number of times the basic field descriptor is 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 15-3 and 15-4 (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 parentheses 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 permissible.

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 is 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.

Table 15-3. Results of Formats in Output Statements

<table>
<tr><th>FORMAT</th><th>OUTPUT</th></tr>
<tr><td>snFw.d
Floating</td><td>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.</td></tr>
<tr><td>snEw.d
Exponential</td><td>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.</td></tr>
<tr><td>snGw.d
General</td><td>Prints Real or Double Precision numbers in F or E format according to the magnitude of the number and the decimal place specifier - d.
<table>
<tr><th>Magnitude</th><th>Effective Format</th></tr>
<tr><td>0.1 to 1.0</td><td>F(w-4) .d,4X</td></tr>
<tr><td>1.0 to 10.0</td><td>F(w-4).(d-1), 4X</td></tr>
<tr><td>.</td><td>.</td></tr>
<tr><td>.</td><td>.</td></tr>
<tr><td>.</td><td>.</td></tr>
<tr><td>10**(d-2) to 10**(d-1)</td><td>F(w-4) .1, 4X</td></tr>
<tr><td>10**(d-1) to 10**d</td><td>F(w-4) .0, 4X</td></tr>
<tr><td>Outside Range</td><td>Ew.d</td></tr>
</table>
Truncation is performed as for E and F formats.</td></tr>
<tr><td>snDw.d
Double Precision</td><td>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.</td></tr>
<tr><td>wX
Space</td><td>Writes w spaces into the output record (negative w backspaces for replacing).*</td></tr>
</table>

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 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 and 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 with the format specifier itself, but the format specifier may be included in a group repetition.*

Formats as variables: It is possible to enter format statements at run time by any method of building this format 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 statements that handle 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 parentheses 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

Table 15-4. Results of Formats in Input Statements

FORMAT	INPUT
snFw.d **Floating** **snEw.d** **Exponential** **snGw.d** **General** **snDw.d** **Double-Precision**	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 there is 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. All numbers are initially converted internally to double-precision numbers; if entered in E,F, or G format, they are truncated.
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.
nLw **Logical**	Stores true/false in internal representation based upon first *non-space* characters in the input data (all others ignored). If T it is set to +1; if F it is set to 0; if anything else it is set to 0 and the error flag is set (use OVERFL to look at error flag).
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 with the format specifier itself but the format specifier may be included in a group repetition.*

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 suppression).

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).

Zed (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 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 15-5 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 15-3 and 15-4)

Where n, the scale factor, is an integer constant with an optional minus sign. Once a scale factor has been specified, it appies 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.

Table 15-5. 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

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.

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. In 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 has 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-hand side is an expression in 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)]

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 where a value is to be returned to the calling program. This will cause the constant to be altered and produce undesirable results.

16

FORTRAN function and subroutine structure

FUNCTIONS

There are four types of functions; all are called in the same manner (see Section 15).

Prime FORTRAN library functions

These library subprograms (see Section 18) which are called automatically by the compiler as required 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 in logical processing of integers. They are automatically inserted in the program by the compiler as required.

User-defined function subprograms

FUNCTION subprograms can be created by the user 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 calcu-
lations, 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 of the variables supplied as the arguments 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 19 and Reference Guide, PRIMOS Subroutines).

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 19 and Reference Guide, PRIMOS Subroutines).

FORTRAN math subroutines

These handle mathematical calculations such as matrix multiply and inversion permutations, etc. (See Section 19).

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 calculations,
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 the number of characters, but the first letter does not set the data mode of the result. 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.

5

UTILITY REFERENCE

17

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 administrator may have changed the defaults; if so, the programmer should obtain a list of the installation-specific defaults. (See figure 17-1).

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 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 11 for more information on this requirement.

B[INARY] $\begin{Bmatrix} \textbf{pathname} \\ \text{YES} \\ \text{NO} \end{Bmatrix}$

Specifies the binary (object) output file. If **pathname** 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 17-1.)

DCLVAR

Flags undeclared variable. 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.**

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 64).

DYNM

Enables local storage in Stack Frame (Prime 350 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.

Table 17-1. Compiler File Specifications

Compiler Mnemonics	INPUT or SOURCE	LISTING	BINARY
pathname	looks for file named **pathname** as source file	opens file named **pathname** as listing file	opens file named **pathname** as (object) file.
YES	*not applicable*	uses default filename for listing file. L_PROGRAM	uses default filename for binary file. B_PROGRAM
NO	*not applicable*	no listing file.	no listing file.
TTY	compiles program as entered from the terminal.	prints listing on user terminal.	*not applicable*
SPOOL	*not applicable*	spools 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 17-2 for A- and B-register settings.

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

Generate instructions from the floating-point skip set when testing the result of a floating-point operation.

I[NPUT] pathname

Specifies the **pathname** of the input source program (See Table 17-1). 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. 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., RESU$$) in a program compiled with the LONG INTEGER default options.

Example:

```
CALL RESU$$('AUDIT YEAR', INTS(10))
```

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.

$$\textbf{L[ISTING]} \begin{Bmatrix} \textbf{pathname} \\ \textbf{YES} \\ \textbf{NO} \\ \textbf{TTY} \\ \textbf{SPOOL} \end{Bmatrix}$$

Specifies the listing device/filename:

pathname	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**.

NOCLVAR *Default*

Suppresses undeclared variable flagging. Does not generate error messages when undeclared variables are detected. See **DCLVAR**.

NOERRTTY

No terminal error messages. Suppresses the printing of error messages on the users terminal. See **ERRTTY**.

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**.

OPT *FTNOPT only*

Optimizes all DO loops that do not contain GO TO expressions. The loops are optimized by removal of invariant expressions and by strength reduction of expressions involving the DO-loop index. Strength reduction can be done if the loop index is altered in the normal loop increment only and if the loop increment is invariant within the loop. See **UNCOPT**.

PBECB

Generates code to load Entry Control Blocks (ECBs) into procedure frame. For 64V-mode subroutines only. See **64V**.

SAVE *Default*

Local storage allocation. Performs local storage allocation statically. See **DYNM**.

S[OURCE]

Same as **I[NPUT]**. See **INPUT**.

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**.

UNCOPT *FTNOPT only*

Unconditionally optimizes all DO loops. The optimization is performed in the same manner as for the OPT option. If the loop GO TO statements transfer control within the loop or simply exit the loop, then the code generated by the compiler will execute correctly. However, if any loop contains a GO TO statement that exits to a code sequence which transfers control back inside the loop, then the optimized code will most likely not execute correctly. This is especially true if the code sequence modifies any operands invariant within the loop or modifies the loop index or loop index increment. It is the programmer's responsibility to insure that these operations are not performed if the UNCOPT option is to be used. See **OPT**.

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 256 segments of 128K bytes each. It may be run on any Prime 350 (or higher system).
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 350 (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 pathname [**1/a-register**] [**2/b-register**]

where **pathname** is the pathname 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 pathname	*default A and B registers*
FTN pathname 1/a-reg	*default B register*
FTN pathname 2/b-reg	*default A register*

For non-default values include both parameters:

FTN pathname 1/a-reg 2/b-reg

or

FTN pathname 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 17-2.

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 17-3.

Default		**A Register Bit**	**Reset (0)**	**Set (1)**
0	0	1		
0	0	2	**LIST**	**EXPLIST**
	0	3	**LIST**	**ERRLIST**
	0	4	**NOTRACE**	**TRACE**
1	0	5	**32R**	**64R**
	0	6		**DEBASE**
	1	7	**NOERRTTY**	**ERRTTY**
7	1	8		**INPUT**
	1	9		**SOURCE**
	1	10		
0	0	11		**LISTING**
	0	12		
	0	13		
7	1	14		**BINARY**
	1	15		
	1	16		
		B register Bit		
0	0	1		
0	0	2		**PBECB**
	0	3	**SAVE**	**DYNM**
	0	4		
0	0	5		**OPT** (*FTNOPT only*)
	0	6		**UNCOPT** (*FTNOPT only*)
	0	7		
	0	8	**NOBIG, 32R**	**BIG, DYNM, 64V**

Table 17-2. 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 17-3); PRIMOS BINARY command
DCLVAR	B(16) = 1
DEBASE	A(6) = 1
DYNM	B(3,8) = 1
ERRLIST	A(3) = 1
ERRTTY	A(7) = 1; *default*
EXPLIST	A(2) = 1
FP	B(15) = 0; *default*
I[NPUT]	A(8,9,10) = input file definition (See Table 17-3)
INTL	B(10) = 1
INTS	B(10) = 0; *default*
L[ISTING]	A(11,12,13) = listing file definition (see Table 17-3); PRIMOS LISTING command
NOBIG	B(8,9) = 0; *default*
NODCLVAR	B(16) = 0
NOERRTTY	A(7) = 0
NOFP	B(15) = 1
NOTRACE	A(4) = 0; *default*
NOXREF	B(12,13) = 0; *default*
OPT	B(5) = 1; (*FTNOPT only*)
PBECB	B(2) = 1
SAVE	B(3) = 0; *default*
S[OURCE]	A(8,9,10) = input file definition (see Table 17-3); same as I[NPUT]
TRACE	A(4) = 1
UNCOPT	B(6) = 1; (*FTNOPT only*)
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 17-3. 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

The PRIMOS commands

LISTING pathname-2 opens a listing file with the specified name pathname-2 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 pathname
 .
 .
 .
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, **pathname**, will contain the concatenation of all listing outputs from **source-1,...**, **source-n** (for those compilations wherein listings were specified).

BINARY pathname-3 opens a binary (object) file with the specified name pathname-3 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 typing:

```
                  2             3
C[LOSE]     pathname-2    pathname-3

or

C[LOSE]     ALL
```

18

FORTRAN function 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 the Reference Guide, PRIMOS Subroutines.

V-Mode FORTRAN library

Certain single-argument scientific subroutines in the V-mode FORTRAN library will be automatically replaced *by the compiler* with their short call versions, identified by the suffix $X. These $X versions execute faster than their regular counterparts.

The $X versions are not directly accessible to the FORTRAN programmer (and have different calling sequences). They will only be noticeable at the load-map level.

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 to be 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 15.).

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)**
AMAX1	Finds the maxmum value in a variable list of single-precision floating-point numbers. **SP = AMAX1 (SP1,SP2,. . .,SPn)**
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)**
AMIN1	Finds the minimum value in a variable list of single-precision floating-point numbers. **SP = AMIN1 (SP1,SP2,. . .,SPn)**

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)**
DMIN1	Finds the minimum value among a variable list of double-precision floating-point numbers. **DP = DMIN1 (DP1,DP2,. . .,DPn)**
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 sign of a second (DP2). **DP = DSIGN (DP1,DP2)**
DSIN	Computes the sine 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 conversions 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	$0 \leqslant I2 \leqslant 32767$
<0	Initializes the random number generator and returns the first random number	$0 \leqslant I2 \leqslant 32767$

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.

$$I = LOC \left\{ \begin{array}{l} \textbf{constant} \\ \textbf{variable name} \\ \textbf{array name} \\ \textbf{array element} \end{array} \right\}$$

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, 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**) **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. **I = MAX1 (SP1,SP2,. . .,SPn)**
MIN0	Finds the minimum value among a variable list of integers. (see **AMIN0**). **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**) **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 zeros. **I2 = RS (I1,IP)** where IP is the number of bits to be shifted to the right. 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, 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 determines 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)**

19

Libraries reference

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, whose name is MATHLB.

For further details on the COMB and PERM subroutines, see "Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," Gideon Ehrlich, Journal of the ACM, **20** No. 3 (July 1973) pp. 5000–5113.

Matrix operations subroutines

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.) COMB is not loopless.

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.

CALL $\left\{\begin{array}{l}\textbf{CLINEQ}\\ \textbf{DLINEQ}\\ \textbf{LINEQ}\end{array}\right\}$ (xvect, yvect, cmat, work, n, npl, ierr)

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** (np1 = n+1). The integer error flag **ierr** returns one of three possible values.

ierr	
0	solution found
1	coefficient matrix singular
2	np1 ≠ 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	np1,np1	*work*
np1	Integer			*passed* (=n+1)
ierr	Integer			*returned*

* all of the same mode which determine the subroutine used.

$$\textbf{CALL}\left\{\begin{array}{l}\textbf{CMADD}\\ \textbf{DMADD}\\ \textbf{IMADD}\\ \textbf{MADD}\end{array}\right\}\textbf{(mats, mat1, mat2, n,m)}$$

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.

$$\textbf{CALL}\left\{\begin{array}{l}\textbf{CMADJ}\\ \textbf{DMADJ}\\ \textbf{IMADJ}\\ \textbf{MADJ}\end{array}\right\}\textbf{(mato, mati, n, iw1, iw2, iw3, iw4, ierr)}$$

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*
iw1	*	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.

$$\text{CALL} \left\{\begin{matrix}\text{CMCOF}\\ \text{DMCOF}\\ \text{IMCOF}\\ \text{MCOF}\end{matrix}\right\} \textbf{(cof, mat, n, iw1, iw2, iw3, iw4, i,j, ierr)}$$

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 $j \neq 0$ or $i \neq 0$ 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*
iw1	*	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.

$$\text{CALL} \left\{\begin{matrix}\text{CMCON}\\ \text{DMCON}\\ \text{IMCON}\\ \text{MCON}\end{matrix}\right\} \textbf{(mat, n, m, con)}$$

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.

$$\text{CALL}\left\{\begin{array}{l}\textbf{CMDET}\\ \textbf{DMDET}\\ \textbf{IMDET}\\ \textbf{MDET}\end{array}\right\}\textbf{(det, mat, n, iw1, iw2, iw3, iw4, ierr)}$$

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*
iw1	*	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.

$$\text{CALL}\left\{\begin{array}{l}\textbf{CMIDN}\\ \textbf{DMIDN}\\ \textbf{IMIDN}\\ \textbf{MIDN}\end{array}\right\}\textbf{(mat, n)}$$

Sets the nxn matrix **mat** equal to the nxn identity matrix. That is,

mat (i,j) = 0 if i ≠ j
mat (i,j) = 1 if i = j

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.

$$\text{CALL}\left\{\begin{array}{l}\textbf{CMINV}\\ \textbf{DMINV}\\ \textbf{MINV}\end{array}\right\}\textbf{(mato, mati, n, work, np1, npn, ierr)}$$

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 np1 x npn scratch matrix work, where np1 = 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	np1 ≠ 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	np1,npn	*work*
np1	Integer			*passed*
npn	Integer			*passed*
ierr	Integer			*returned*

$$\text{CALL}\left\{\begin{matrix}\textbf{CMMLT}\\ \textbf{DMMLT}\\ \textbf{IMMLT}\\ \textbf{MMLT}\end{matrix}\right\}\textbf{(matp, matl, matr, n1, n2, n3)}$$

Multiplies the n1xn2 matrix **matl** (on the left) by the n2xn3 matrix **matr** (on the right) and stores the resulting n1xn3 product matrix in **matp.**

Note

matp must be distinct from mat1 and matr, although mat1 and matr may be the same. For example:

```
CALL MMLT (A, B, C, N1, 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	n1,n3	*returned*
matl	*	2	n1,n2	*passed*
matr	*	2	n2,n3	*passed*
n1	Integer			*passed*
n2	Integer			*passed*
n3	Integer			*passed*

* are of the same mode which determines which subroutine is used.

$$\text{CALL}\left\{\begin{matrix}\textbf{CMSCL}\\ \textbf{DMSCL}\\ \textbf{IMSCL}\\ \textbf{MSCL}\end{matrix}\right\}\textbf{(mato, mati, n, m, scon)}$$

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.

$$\textbf{CALL}\left\{\begin{array}{l}\textbf{CMSUB}\\ \textbf{DMSUB}\\ \textbf{IMSUB}\\ \textbf{MSUB}\end{array}\right\}\textbf{(matd, mat1, mat2, n,m)}$$

Subtracts the nxm matrix **mat2** from the nxm matrix **mat1** and stores the difference in the nxm matrix **matd.**

Argument	**Mode**	**Subscript(s)**	**Dimension(s)**	**Comments**
matd	*	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 determine the subroutine to be used.

$$\textbf{CALL}\left\{\begin{array}{l}\textbf{CMTRN}\\ \textbf{DMTRN}\\ \textbf{IMTRN}\\ \textbf{MTRN}\end{array}\right\}\textbf{(mato, mati, n)}$$

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)**	**Comments**
mato	*	2	n,n	*returned*
mati	*	2	n,n	*passed*
n	Integer			*passed*

* all of the same mode which determines the subroutine used.

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)*

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, PRIMOS Subroutines.

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*

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 multiple-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

ptable	Pointer to first word of table of entries. Example: the table is an array ITABLE(I), 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 field. 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 address for an error 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 entries must be 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)**

tarray has dimension **nkwrds.**
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)**

tarray has dimension **nwords**

INSERT - *straight insertion sort*

CALL INSERT **(ptable,nentry,nwords,fword,nkwrds,npass,altbp,incr)**

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)**

tarray has dimension **nwords**

RADXEX- *radix exchange sort*

CALL RADXEX **(ptable,nentry,nwords,fword,fbit,nbit,tarray,npass,altbp)**

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

tarray has dimension **2*nbit**

SHELL - *diminishing increment sort*

CALL SHELL **(ptable,nentry,nwords,fword,nkwrds,npass,altpb)**

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 and 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 the applications library subroutines must define the values of the keys used in these routines. This definition is performed by placing the instruction $INSERT SYSCOM >A$KEYS in each module which uses any of these subroutines.

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, PRIMOS subroutines.

The applications library subroutines may be grouped by their functions:

File System	TEMP$A, OPEN$A, OPNP$A, OPNV$A, OPVP$A, CLOS$A, RWND$A, GEND$A, TRNC$A, DELE$A, EXST$A, UNIT$A, RPOS$A, POSN$A, TSCN$A.
String Manipulation	FILL$A, NLEN$A, MCHR$A, GCHR$A, TREE$A, TYPE$A, MSTR$A, MSUB$A, CSTR$A, CSUB$A, LSTR$A, LSUB$A, JSTR$A.
User query	YSNO$A, RNAM$A, RNUM$A.
System Information	TIME$A, CTIM$A, DTIM$A, DATE$A, EDAT$A, DOFY$A.
Conversions	ENCD$A, CNVA$A, CNVB$A.
Mathematical Routines	RNDI$A, RAND$A.
Parsing	CMDL$A.

A brief description of these routines follows, in alphabetical order.

CLOS$A *LOGICAL*

Attempts to close a file by the file unit number on which it was opened. Reports on success or failure of attempt.

CMDL$A *LOGICAL*

Parses a PRIMOS-like command line and returns information for each -keyword (and optional argument) entry in the line (one entry per call).

CNVA$A *LOGICAL*

Converts an ASCII digit string to a numerical value for octal, decimal, and hexadecimal numbers. Reports whether the conversion was made successfully or not.

CNVB$ *INTEGER*2*

Converts a binary number (INTEGER*4) to an ASCII digit string for decimal, octal, and hexadecimal numbers. The function value is the number of digits in the string (or 0 if the conversion is unsuccessful).

CSTR$A *LOGICAL*

Compares two character strings for equality and returns .TRUE. as the function value if they are equal.

CSUB$A *LOGICAL*

Compares two substrings of character strings for equality and returns .TRUE. as the function value if they are equal.

CTIM$A *REAL*8*

Returns the CPU time since login in centiseconds (argument returned) and in seconds (function value).

DATE$A *REAL*8*

Returns the system date as DAY MON DD 19YR (argument returned) and as MM/DD/YY (function value).

DELE$A *LOGICAL*

Attempts to delete a file specified by the filename. If successful the function is .TRUE., otherwise .FALSE..

DOFY$A *REAL*8*

Returns the day of the year as a 3-digit number (argument returned) and as YR.DDD (function value). The latter is suitable for printing in FORMAT F6.3.

DTIM$A *REAL*8*

Returns disk time since login in centiseconds (argument returned) and in seconds (function value).

EDAT$A *REAL*8*

Returns the date as DAY, DD MON 19YR (argument returned) and as DD/MM/YR (function value). This is the European/military format.

ENCD$A *LOGICAL*

Encodes a value in FORTRAN floating-point print format (Fw.d) and reports whether the encoding was successful or not.

EXST$A *LOGICAL*

Checks for the existence of a file specified by name and reports whether the file exists or not.

FILL$A *INTEGER*

Fills a buffer with a specified ASCII character.

GCHR$A *INTEGER*

Accesses a character in a specified array position. The function value is the character in FORTRAN A1 FORMAT (right padded with blanks).

GEN$A *LOGICAL*

Positions a file pointer opened on a specified file unit to the End-of-File. The function value tells whether the positioning was successful or not.

JSTR$A *LOGICAL*

Right- or left-justifies a string and reports whether the operation is successful.

LSTR$A *LOGICAL*

Locates a string within another string. The function value reports on whether the substring was found or not.

LSUB$A *LOGICAL*

Locates one substring within another substring. The function value reports on whether the substring was found or not.

MCHR$A *INTEGER*

Replaces a character in one array with a specified character from another. The function value is the character moved in FORTRAN A1 FORMAT, right padded with blanks.

MSTR$A *INTEGER*

Moves one string to another string. The function value is equal to the number of characters moved.

MSUB$A *INTEGER*

Moves a substring into a substring in another string. The function value is equal to the number of characters moved.

NLEN$A *INTEGER*2*

Returns the operational length of string in a buffer, excluding trailing blanks.

OPEN$A *LOGICAL*

Opens a file on a specified file unit. The function value reports whether the operation was successful or not.

OPNP$A *LOGICAL*

Gets a filename from the user terminal and opens that file on a specified file unit. The function value reports whether the operation was successful or not.

OPNV$A *LOGICAL*

Opens a file on a specified file unit, verifies the operation. If the file is in use the operations are re-tried. The function value reports on the ultimate success of the operations.

OPVP$A *LOGICAL*

Gets a file name from the user terminal and opens that file on a specified file unit. The operations are verified. If the file is in use the operations are re-tried. The function value reports on the ultimate success of the operations.

POSN$A *LOGICAL*

Positions the pointer in the file open on a specified file unit. The function value reports on the success of the operation.

RAND$A *REAL*8*

Updates the seed of a random number generator. The old seed is passed and a new seed returned. The function value is a random number between 0.0 and 1.0.

RNAM$A *LOGICAL*

Prints a prompt message at the terminal and accepts a name from the terminal. The function value reports on the validity of the name.

RNDI$A *REAL*8*

Generates the initializing seed for a random number generator. The information returned is time of day in centiseconds (argument returned) and in seconds (function value).

RNUM$A *LOGICAL*

Prints a prompt message at the terminal and accepts a number (octal, decimal, or hexadecimal) string from the terminal. If successful the value is returned in one of the subroutine arguments and the function value is .TRUE..

RPOS$A *LOGICAL*

Returns the current absolute position of the pointer in the file opened on a specified file unit. The function value reports on the success of the operation.

RWND$A *LOGICAL*

Rewinds the file opened on the specified file unit. The function value reports on the success of the operation.

TEMP$A *LOGICAL*

Opens a temporary file with a unique name in the current UFD for reading and writing on a specified file unit. The name is returned as an argument in the subroutine call. The function value reports on the success of the operation.

TIME$A *REAL*8*

Returns the time of day as HR:MN:SC (argument returned) and in decimal hours (function value).

TREE$A *LOGICAL*

Scans a string to check whether it is a valid pathname and, if so, locates the final part of the name in the string. The function value reports whether the test is successful or not.

TRNC$A *LOGICAL*

Truncates the file opened on a specified file unit. The function value reports on the success of the operation.

TSCN$A *LOGICAL*

Scans the file system tree-structure (starting with the home directory) to read UFDs and segment directory entries. Each call returns the next file on the current level or the first file on the next lower level. The function value is .TRUE. until an error occurs or an end of file is reached.

TYPE$A *LOGICAL*

Tests a character string to see whether it can be interpreted as a number (octal, decimal, or hexadecimal) or a name. The function value reports whether the string meets the specified criterion.

UNIT$A *LOGICAL*

Tests whether any file is open on a specified file unit. The function value reports whether the unit is in use or not.

YSNO$A *LOGICAL*

Prints a question at the user terminal which can be answered YES (or OK) or NO. The function value is .TRUE. for YES (or OK) and .FALSE. for NO. Any other answer causes the question to be repeated.

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, PRIMOS Subroutines.

File access

Files are structured to be accessed in either of two ways: SAM, or Sequential Access Method, and DAM, or Direct Access Method. SAM files are the most common type of file created and processed by PRIMOS. Most files likely to be dealt with by the user are SAM files.

SAM files: A SAM file consists of records threaded together with forward and backward pointers. Each record in the file contains a pointer to the beginning record address (BRA) of the file. The beginning record of the file contains a pointer to the file directory in which it is listed. Since records are strung together in this manner, they can only be accessed sequentially; the entire file must be searched from the beginning in order to find a record. This is time consuming when many random accesses must be done. However, SAM files are more compact and require less disk storage space than DAM files. SAM files are accessed by PRIMOS commands such as ED, etc.

DAM files: DAM files have a multi-level index containing pointers to every record on the file. If the file is short, the record address pointers point directly to records containing data. If the file is long, these pointers reference other records containing a lower level index. Those indices in turn have pointers to records containing data.

DAM structure is more suitable to rapid, random access of data than SAM structure. Each individual record can be referenced by a unique pointer connecting the record and a pointer index at the beginning of the file. Searching the pointer index for a particular record is quicker than hunting through each entire record in sequence.

DAM files are less compact than SAM files. The MIDAS subsystem or user applications programs must be used to access them. DAM files occur in the MIDAS and SEG subsystems.

Names

In the file system calls, 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)

```
ILLEGAL NAME.  OPENING NEWFILE (NEWWRT)
```

These errors are included in the list of run-time errors in Appendix A. They are labelled as File System errors.

Operating System Subroutines

A list of all operating system subroutines with a brief description of their function is given below. Subroutines marked with a bullet (•) are described in detail following this list.

• **ATCH$$**	Attaches to a UFD and optionally makes it the home UFD.
• **CNAM$$**	Changes a filename.
COMI$$	Switches command input stream from terminal to command file and vice-versa.
COMO$$	Switches output stream from terminal to file and vice-versa.
CREA$$	Creates a sub-UFD in the current UFD.
ERKL$$	Reads or sets the erase and kill characters.
GPAS$$	Returns passwords of sub-UFD in the current UFD.
NAMEQ$	Compares filenames for equivalence.
• **PRWF$$**	Reads, writes, and positions pointer in a SAM or DAM file.
RDEN$$	Reads entry in 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$$**	Restores an *R-mode* memory image from a file, sets initial values, and begins execution. An error in this call causes an error message to be printed automatically and then returns command to PRIMOS.
SATR$$	Sets attributes (protection, date, time, etc.) in a UFD entry.
SAVE$$	Saves an R-mode memory image in user memory by writing it into a disk file.
SGDR$$	Positions and reads segment directory entries.
SPAS$$	Sets the passwords in the current UFD.
• **SRCH$$**	Opens or closes a file.
TEXTO$	Checks the validity of a filename.
• **TSRC$$**	Opens or closes a file anywhere in the PRIMOS file structure.
WTLIN$	Writes a line of ASCII characters to a disk file in compressed format.

ATCH$$

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	**Action**
K$IMFD	Attach to ufd-name in MFD on logical-disk.
K$ICUR	Attach to ufd-name in current UFD.

set-key	**Action**
K$SETH	Set current UFD to home after attaching.

code	Returns integer-valued error code.

CNAM$$

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.

PRWF$$

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.
K$POSR	Performs **read-write-key** operation then move file pointer of **file-unit position-value** words relative to current position.
K$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$CONV	Read/write convenient number of words up to **number-of-words.** See Reference Guide, PRIMOS subroutines for a discussion of "convenient".
K$FRCW	Perform write to disk from buffer before executing next instruction in the program. Increases disk I/O time.
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; other keys leave this parameter unmodified. (INTEGER*2).
code	Returns integer-valued error code.

RESU$$

CALL RESU$$ (filename,name-length)

filename	Name of the file containing the memory image.
name-length	Number of characters in filename.

SRCH$$

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 the **action** on the file that is a segment directory entry in the directory which is open on **filename.**
K$CACC	Change access rights of file open on file-unit to **action.**
K$GETU	Open **filename** on an unused file-unit selected by PRIMOS. The unit number is returned in **file-unit.**
new-file	Specifies type of file to create if file-name 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.

Note

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.

TSRC$$

CALL TSRC$$ (action+new-file,pathname,file-unit,character-position, code)

action	Action to be taken (*mandatory*).
K$READ	Open **pathname** for reading on **file-unit.**
K$WRIT	Open **pathname** for writing on **file-unit.**
K$RDWR	Open **pathname** for reading and writing on **file-unit.**
K$DELE	Delete file **pathname.**
K$EXST	Check on existence of **pathname.**
new-file	Specifies type of file to create if **pathname** does not already exist.
K$NSAM	SAM file (this is the default).
K$NDAM	DAM file.
K$NSGS	SAM segment directory.
K$NSGD	DAM segment directory.
pathname	A specification of any file in any directory or subdirectory stored in array **pathname** 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 **pathname** (count starts at 0) returns: one past the last character that was part of the pathname. word 2 – the number of characters in the **pathname.**
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

Note

TSRC$$ always closes the file unit, then attaches to the user's home UFD before attempting any action.

APPENDICES

A

Error messages

INTRODUCTION

Error messages are given in the following order:

1. FORTRAN Compiler Error Messages
2. Loader Error Messages
3. SEG Loader Error Messages
4. 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.

COMPILER ERROR MESSAGES

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.

ARRAY NESTING OVFLO

Use of arrays as subscripts in other arrays exceeds allowable nesting limit (32).

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 where only a constant or parameter is allowed (i.e., DIMENSION 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.

END/REC PROHIBITED

The END=statement-number expression cannot be used in a direct access READ or WRITE statement.

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 a FUNCTION subprogram.

GBL MDE/IMPL CNFLCT

IMPLICIT statement and global mode specification may not be used in the 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 within the context of the program; for example, RETURN in a main program, SUBROUTINE not the first subprogram 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 will 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.

INTERNAL ERROR

Some combination of source code statements has generated an unresolvable error. The programmer should never see this error.

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.

PAREN NESTING>31

Nesting of parentheses (syntactical, array, or function reference) in expressions may not exceed 31.

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).

TOO FEW SUBSCRIPTS

Number of subscripts used in an array is fewer than the number originally declared in a DIMENSION or mode specification statement.

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.

LOADER ERROR MESSAGES

ALREADY EXISTS!

An attempt is being made to define a new symbol; however, the symbol name is already a defined symbol in the symbol table.

BAD OBJECT FILE

The object text is not recognizable. This usually occurs when an attempt is made to load source code or when the object text was compiled or assembled for segmented loading.

BASE SECTOR 0 FULL

All locations in the sector zero base area have been used. Use the AU command to generate base areas at regular intervals, or use the SETB or LOAD commands to specifically place base areas.

CAN'T DEFER COMMON, OLD OBJECT TEXT

The Defer Common command has been given and a module created with a pre-Rev. 14 compiler or assembler has been encountered. It is not possible to defer Common in this case. The module must be recreated with a Rev. 15 or later compiler or assembler.

CAN'T - PLEASE SAVE

The EXecute command has been given for a run file which has required virtual loading. SAve the runfile and give the EXecute command.

CM$

Command line error. Unrecognized command given. Not fatal.

COMMON OUT OF REACH

COMMON above '100000 is out of reach of the current load mode (16S, 32S or 32R). Use the MOde command to set the load mode to 64R.

COMMON TOO LARGE

Definition of this COMMON block causes COMMON to wrap around through zero. Moving the top of COMMON - with the COMMON command - may help.

sname ILLEGAL COMMON REDEFINITION

An attempt is being made to redefine COMMON block **sname** to a longer length. The user's program should be examined for consistent COMMON definitions. At the very least the longest definition for a COMMON block should be first.

xxxxxx MULTIPLE INDIRECT

A module loading in 64R mode requires a second level of indirection at location **xxxxxx**. This message usually results when an attempt is made to load code compiled or assembled for 32R mode in 64R mode. It can also happen if code has accidentally been loaded into base areas as the result of a bad load command sequence.

sname xxxxxx NEED SECTOR ZERO LINK

At location **xxxxxx** a link is required for desectoring the instruction. No base areas are within reach except sector zero. The last referenced symbol was **sname**. This message is only generated when the SZ command has been given. Sname may be the name of a COMMON block, the name of the routine to which the link should be made, or the name of the module being loaded.

xxxxxx NO POST BASE AREA, OLD OBJECT TEXT

A post base area has been specified for module which was created with a pre-Rev.14 compiler or assembler. No base area is created. Recreate the object text with a Rev. 15 or later compiler or assembler. This is not a fatal error.

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 move COMMON up higher.

PROGRAM TOO LARGE

The program has loaded into the last location in memory and has wrapped around to load in Location 0. The program size must be decreased. Alternatively, compile in 64V mode and use SEG.

REFERENCE TO UNDEFINED COMMON

An attempt is being made to link to a COMMON name which has not been defined. This usually happens to users creating their own translators.

SECTORED LOAD MODE INVALID

A module compiled or assembled to load in R mode has been loaded in S mode. Use the MOde command to reset the load mode. It might be a good idea to be sure that all modules are correctly written, since the default load mode is 32R.

SYMBOL NOT FOUND

An attempt is being made to equate two symbols with the SYmbol command and the old symbol does not exist.

SYMBOL TABLE FULL

The symbol table has expanded down to location '4000. The last buffer cannot be assigned to the symbol table. Rebuild LOAD to load in higher memory locations, or reduce the number of symbols in the load.

SYMBOL UNDEFINED

An attempt is being made to equate two symbols; however, the old symbol is an undefined symbol in the symbol table.

64R LOAD MODE INVALID

A module compiled or assembled to run in only 32K of memory is being loaded in 64R mode. Recompile or reassemble or change the load mode with the loader's MOde command.

SEG LOADER ERROR MESSAGES

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 error, 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 that the average applications programmer will encounter this. Fatal error, 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 error, 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 error, abort load.

ILLEGAL SPLIT ADDRESS

Incorrect use of the Loader's SPLIT command. Segments may be split 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 program wherein COMMON would be allocated to an illegal segment number. Recompile in 64V mode. Fatal error, 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 error, 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 error, 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 error, 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 error, abort load.

RUN-TIME ERROR MESSAGES

ACCESS VIOLATION **64V mode**

Attempt to perform operations in segments to which user has no right.

******AD** **R-mode function**

Overflow or underflow in double-precision addition/subtraction (A$66,S$66).

ALL REMOTE UNITS IN USE **File System**

Attempt made to assign a remote unit when none are available. (Network error) [E$FUIU]

****** ALOG/ALOG 10 - ARGUMENT <=0** **V-mode function**

Argument not greater than zero used in logarithm (ALOG, ALOG 10) function.

filename ALREADY EXISTS **Old file call**

Attempt to create a file or UFD with the name of one already existing. [CZ]

ALREADY EXISTS **File System**

Attempt made to create, in the UFD, a sub-UFD with the same name as one already existing. (CREA$$) [E$EXST]

******AT** **R-mode function**

Both arguments are zero in the ATAN2 function.

****** ATAN2 - BOTH ARGUMENTS = 0** **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 **File System**

The DAM file specified has been corrupted - either by the programmer or by a system problem. (PRWF$$, SRCH$$). [E$BDAM]

BAD FAM SVC **File System**

System problem; will not be seen by applications programmer. [E$BFSV]

BAD KEY **File System**

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 **File System**

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 **File System**

Error generated in accessing segment directory, i.e., PRIMOS file unit specified is not a segment directory. (SRCH$$) [E$BSUN]

BAD SEGMENT NUMBER **File System**

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 **File System**

Error encountered in truncating segment directory. (SGDR$$) [E$BTRN]

BAD UFD **File System**

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 **File System**

PRIMOS file unit number specified is invalid - outside legal range. (PRWF$$, RDEN$$, SRCH$$, SGDR$$). [E$BUNT]

BEGINNING OF FILE **File System**

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 **File System**

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 **File System**

Attempt was made to ASSIGN a device currently assigned to another user. [E$DVIU]

DEVICE NOT ASSIGNED **File System**

Attempt was made to perform I/O operations on a device before assigning that device. [E$NASS]

DEVICE NOT STARTED **File System**

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 **File System**

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 **File System**

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 **File System**

A read/write error was encountered in accessing disk. Returns immediately to PRIMOS level. Not correctable by applications programmer. (ATTCH$$, CREA$$, GPAS$$, PRWF$$, RDEN$$, SATR$$, SRCH$$, SGDR$$). [E$DISK]

DISK WRITE-PROTECTED **File System**

An attempt has been made to write to a disk which is WRITE-protected. [E$WTPR]

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< = 0** **V-mode function**

Argument not greater than zero was 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 <0** **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 **File System**

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 **File System**

System error. [E$FABT]

FAM - BAD STARTUP **File System**

System error. [E$FBST]

FAM OP NOT COMPLETE **File System**

Network error. [E$FONC]

****FE **R-mode function**

Error in FORMAT statement. FORMAT statements are not completely checked at compile time. (F$IO)

FILE IN USE **File System**

Attempt made to open a file already opened or to close/delete a file opened by another user, etc. (SRCH$$) [E$FDEL]

FILE OPEN ON DELETE **File System**

Attempt made to delete a file which is open. (SRCH$$) [E$FDEL]

FILE TOO BIG **File System**

Attempt made to increase size of segment directory beyond size limit. (SGDR$$) [E$FITB]

****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 has 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 **File System**

Illegal name specified for a file or UFD. (CREA$$, SRCH$$) [E$BNAM]

ILL REMOTE REF **File System**

Attempt to perform network operations by user not on network. [E$IREM]

ILLEGAL SEGNO **64V mode**

Program references a non-existent segment or a segment number greater than those available to the user.

ILLEGAL TREENAME **File System**

The string specified for a treename is syntactically incorrect. [E$ITRE]

******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 to close/delete a file opened by another user, etc. [SI]

INVALID FAM FUNCTION CODE **File System**

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 **File System**

No more users may access the network. [E$TMRU]

NAME TOO LONG **File System**

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 PHANTOMS AVAILABLE **File System**

An attempt has been made to spawn a phantom. All configured phantoms are already in use. [E$NPHA]

NO RIGHT **File System**

User does not have access right to file, or does not have write access in UFD when attempting to create a sub-UFD. (CREA$$, GPAS$$, SATR$$, SRCH$$, SGDR$$) [E$NRIT]

NO ROOM **File System**

An attempt has been made to add to a table of assignable devices with a DISKS or ASSIGN AMLC command and the table is already filled. [E$ROOM]

NO TIME **File System**

Clock not started. System error. [E$NTIM]

NO UFD ATTACHED **Old file call**

User not attached to a UFD [AL, SL]. Usually occurs after attempt to attach with a bad password.

NO UFD ATTACHED **File System**

User not attached to a UFD. (ATCH$$, CREA$$, GPAS$$, SATR$$, SRCH$$). [E$NATT] Usually occurs 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 0
2. Trap to location 0
3. SVC switch on, SVC trap and location '65 is 0.

NOT A SEGDIR **File System**

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 **File System**

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 **File System**

File specified in subroutine call not found. (ATCH$$, GPAS$$, SATR$$, SRCH$$) [E$FNTF]

filename NOT FOUND IN SEGDIR **File System**

Filename specified in subroutine call not found in specified segment directory. (SRCH$$, SGDR$$) [E$FNTS]

NULL READ UNIT **PRIMOS**

Program has attempted to read with a bad unit number. This may be caused by a program overwriting itself (array out of bounds).

OLD PARTITION **File System**

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 Administrator 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).

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 **File System**

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 **File System**

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 function**

Device error or end-of-file in READ statement on FORTRAN logical unit n.

****SE **R-mode function**

Single precision exponent overflow.

SEG-DIR ER **Old file call**

Error encountered in segment directory operation. [SQ]

SEGDIR UNIT NOT OPEN **File System**

Attempt has been made to reference a segment directory which is not open. (SRCH$$) [E$SUNO]

SEM OVERFLOW **File System**

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 function.

**** SQRT - ARGUMENT<0 **V-mode function**

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 **File System**

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 **File System**

UFD has no room for more files and/or sub-UFD's. Occurs only in old file partitions. (CREA$$, SRCH$$) [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 **File System**

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 **File System**

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.

B

System defaults and constants

TERMINAL

full duplex
X-ON/X-OFF disabled

EDITOR (ED)

INPUT (TTY)
LINESZ 144
MODE NCOLUMN
MODE NCOUNT
MODE NNUMBER
MODE NPROMPT
MODE PRALL
VERIFY

SYMBOLS

BLANKS	#
COUNTER	@
CPROMPT	$
DPROMPT	&
ERASE	"
ESCAPE	^
KILL	?
SEMICO	; end of line or command
TAB	\
WILD	!

VIRTUAL LOADER (LOAD)

Memory Location: '122770 – '144000
Loading address: current *PBRK value
Library: FTNLIB FORTRAN library
MODE: D32R
Sector Zero Base Area:
  Base start at location '200
  Base range '600 words
COMMON: Top = '077777

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	CONTROL-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	

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	('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 line kill
\	('334)	is interpreted as a logical tab (Editor)

C ASCII CHARACTER SET

Table C-1. ASCII Character Set (Non-Printing)

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
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 on 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	Synchronocity (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

Table C-2. ASCII Character Set (Printing)

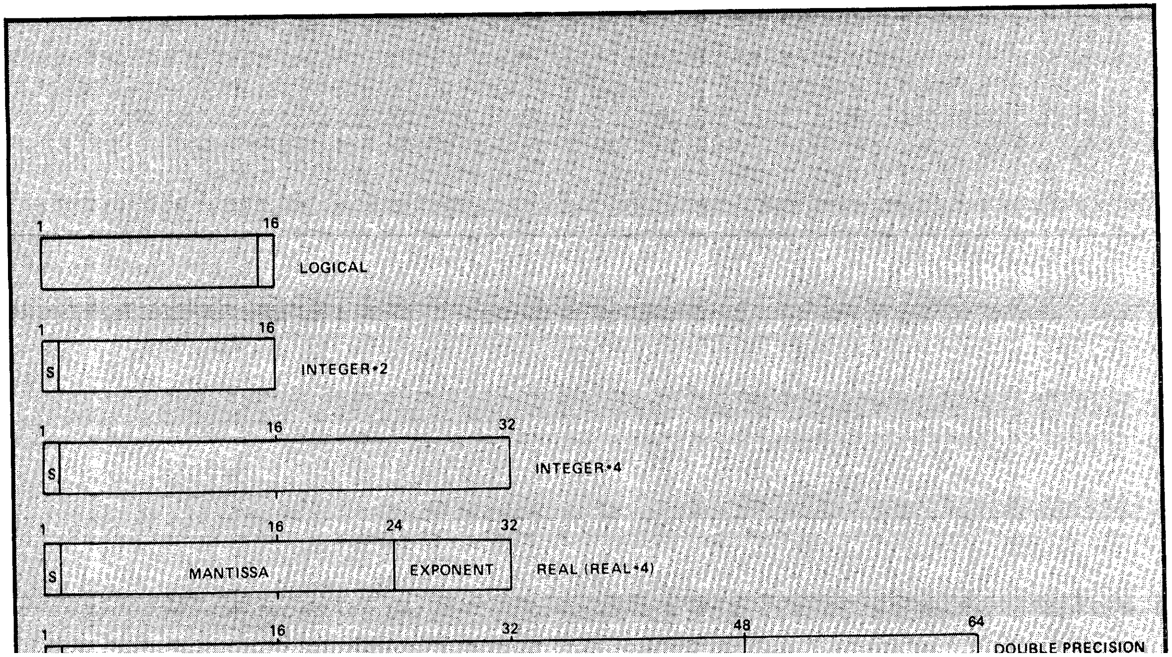
Octal Value	ASCII Character	Octal Value	ASCII Character	Octal Value	ASCII Character
240	.SP. (1)	300	@	340	`(9)
241	!	301	A	341	a
242	"(2)	302	B	342	b
243	# (3)	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' (4)	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	312	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

D

Prime memory formats of FORTRAN data types

INTRODUCTION

Prime machines use a 16-bit memory word which is addressable by word. Prime's FORTRAN data types depart slightly from the ANSI standard which states that LOGICAL, INTEGER, and REAL items occupy one storage unit each. If a storage unit is 32 bits (4 bytes =2 words), then the requirements of ANSI are met except for the LOGICAL type which is only 16 bits. Below is a representation of the sizes of data entities, for the purposes of EQUIVALENCE statements, used by Prime. Detailed descriptions of each type are presented separately.


1
16
LOGICAL
1
16
S
INTEGER*2
1
16
32
S
INTEGER*4
1
16
24
32
S
MANTISSA
EXPONENT
REAL (REAL*4)
1
16
32
48
64
DOUBLE PRECISION

DATA TYPES

LOGICAL 16 bits. Bits 1–15=0, Bit 16=0=.FALSE., 1=.TRUE.

These values are equivalent to INTEGER*2 values of 0 and 1 respectively. Any other values are illegal for LOGICAL variables.

INTEGER*2 16 bits. Bit 1=sign bit. INTEGER numbers are in 2's complement representation with a value range of -32768 to 32767. These numbers in octal are '100000 and '077777 respectively. Note that -0=0, and -(-32768)=-32768.

Integer arithmetic is always exact. Integer division truncates, rather than rounds.

INTEGER*4 32 bits. Bit 1=sign bit. Integer numbers are in 2's complement representation with a value range of -2147483648 to 2147483647. These numbers, in octal (word 1, word 2) are ('100000, '000000) and ('077777, '177777) respectively. Note that -0=0 and -(-2147483648)= -2147483648.

Integer arithmetic is always exact. Integer division truncates, rather than rounds.

CAUTION

Explicit use of DBLE (FLOAT(I*4)) can cause the loss of the low-order 8 bits of precision. Mixed mode arithmetic, however, will not lose precision.

REAL*4 32 bits. Bit 1=sign bit. Bits 2–24=mantissa. Bits 25–32=exponent. The mantissa and sign are treated as a 2's complement number and the exponent is an *unsigned*, excess 128, binary exponent. In general, any floating point number is represented as:

N=M * 2**(E–128)

where

$-1 \leqslant M < -1/2$ or $1/2 \leqslant M < 1$

$0 \leqslant E \leqslant 255$

Zero is represented as M=0, E=0.

The value range, in octal (word1, word2) is:

('100000, '000377) [See Note] to ('077777, '177777)
corresponding to -1*2**(127) and (1-e)*2**(127).

The values closest to zero, in octal are:

('137777, '177400) and ('040000, '000000) [See Note]
corresponding to (-1/2+e)*2**-128 and 1/2*2**-128

Normalization ensures that bits 1 and 2 are different and is achieved by shifting left 1 bit at a time. Hence, the effective precision is between 22 and 23 bits.

Note

These numbers will cause exponent overflow if negated due to the asymmetry of 2's complement notation.

DOUBLE PRECISION 64 bits. Bit 1 = sign bit. Bits 2–48 = mantissa. Bits 49–64 = exponent. The mantissa and sign are treated as a 2's complement number and the exponent is a *signed*, excess 128, binary exponent. In general, any double precision floating point number is represented as:

$N = M * 2\ (E-128)$

where

$-1 \leqslant M < -1/2$ or $1/2 \leqslant M < 1$

$-32768 \leqslant E \leqslant 32767$.

Zero is represented as $M = 0$, $E = 0$.

The value range, in octal (word1, word2, word3, word4) is:

('100000, '000000, '000000, '077777) [See Note] to

('077777, '177777, '177777, '077777)

corresponding to -1*2**32639 and (1-e)*2**32639

The values closest to zero, in octal, are:

('137777, '177777, '177777, '100000) and
('040000, '000000, '000000, '100000) [See Note]

corresponding to (-1/2+e)*2**-32896 and 1/2*2**-32896

Normalization ensures that bits 1 and 2 are different and is achieved by shifting left 1 bit at a time. Hence, the effective precision is between 46 and 47 bits.

Note

These numbers will cause exponent overflows if negated due to the asymmetry of 2's complement notation.

COMPLEX 64 bits. A complex number is defined as two REAL*4 entities (see above) representing the real and imaginary parts.

CHARACTERS Prime uses ASCII as its standard internal and external character code. It is the 8-bit, marking variety (parity bit always on). Thus, Prime's code set is effectively a 128-character code set. (ASCII spacing representation, parity bit always off, can be entered into the system, but most system software will fail to recognize the characters as their terminal printing equivalent.)

Characters packed into numeric items will always be negative numbers if accessed numerically. Also, if the data item is not completely filled (e.g., A2 format into a REAL*4 item), the item will be right padded with blanks (ASCII '240).

The positions of the exponents for REAL and DOUBLE PRECISION items precludes sorting character data as REAL items, but is quite legitimate in integer items. However, EQUAL comparisons in REAL items are valid.

A

A input format 15-22
A output format 15-21
A register 17-6
A register defaults 17-6
A$KEYS file 19-9
Abbreviations, command 2-1
Access, file 19-14
Access, file, controlling 3-8
Accessing PRIMOS 3-1
Accessing the system 3-2
ADD1$, MIDAS subroutine 12-5
Addition, matrix, subroutine 19-2
Address constants 14-6
Address, call by 15-4
Addressing mode 6-7
Adjoint, matrix, subroutine 19-2
Advanced features, SEG LOAD subprocessor 11-1
Advanced SEG features 7-7
Advantages of MIDAS 12-1
Advantages of shared procedures 11-7
Altering stack size 11-6
AND truth table 14-7
Angle brackets, convention 2-4
ANSI standard data storage D-1
APPLIB 19-9
Application library subroutines 16-3
Applications library 19-9
Applications library subroutines, list 19-10
Area TRACE statement 15-8
Areas, base 6-4
Areas, base (SEG) 7-6, 7-8
Arguments, function 16-2
Arguments, subroutine 16-3
Arithmetic
  IF statement 15-12
  mixed mode 15-9
  operators 14-7
  vs. logical IF 13-6
Arrays 14-5
  in over 64K word COMMON 11-12
  dummy argument, over 64K word COMMON 11-12
  segment-spanning 5-10
ASCII card decks, reading 4-2
  character set C-1
  character strings 14-4
  characters, non-printing C-2
  characters, printing C-3
  data storage format D-3
  keyboard input C-1
  magnetic tape, reading from 4-3
  parity C-1
  Prime usage C-1
Assembly language, interface to 12-8
ASSIGN (PRIMOS command) 4-1
ASSIGN option -WAIT 4-1
ASSIGN statement 15-10
Assigned GO TO statement 15-11
Assigned segments, releasing 7-7
Assigning a device 4-1
Assigning directory passwords 3-3
Assignment statements 15-9
  statements, data mode rules, table 15-11
  device, queuing 4-1
  parameter, implicit 11-3
  segment 7-4
  segment, relative 11-1
  segment, specific 11-3
ATCH$$ subroutine 19-16
ATTACH (PRIMOS command) 3-2
Attaching to remote directories 10-18
ATTDEV subroutine 15-14
Attn key 2-5
Audience 1-1
Automatic logout 3-9

B

B format, details 15-21
B output format 15-21
B register 17-6
B register defaults 17-6
Backslash, usage 2-5
BACKSPACE statement 15-25
Base area error messages 6-7
Base areas 6-4
  (SEG) 7-6, 7-8
  conservation 5-10
BCD card decks, reading 4-2
  magnetic tape, reading from 4-3
BIG (compiler option) 5-10, 17-1
BILD$R, MIDAS subroutine 12-2
BINARY (compiler option) 5-3, 17-1
  (PRIMOS command) 17-10
Binary file, compiler (unit 3) 5-3, 17-9
  file, definition 2-1
  files, concatenating 17-10
  magnetic tape, reading from 4-3
  READ statement 15-16
  search subroutine 19-9
  WRITE statement 15-18
Bit-device correspondence, compiler 17-9
Bit-mnemonic correspondence, A register 17-7
Bit-mnemonic correspondence, B register 17-7
Blank COMMON 15-6
Blank COMMON, relocating 11-5
BLOCK DATA statement 15-3
Block data subprogram 15-3
BLOCKDATA statement 15-3
BNSRCH subroutine 19-9
Braces, convention 2-4
Brackets, convention 2-4
Break key 2-5
BUBBLE subroutine 19-8
Building MIDAS data subfile 12-2
Byte, definition 2-1

C

Call by address 15-4
Call by name 15-4
CALL EXIT 1-4
CALL statement 15-7, 15-25
CANCEL (SPOOL option) 3-7
Cancelling spool request 3-7
Card decks, ASCII, reading 4-2
  BCD, reading 4-2
  EBCDIC, reading 4-2
Cards, punched, reading 4-2
Caret, usage 2-5
Central processor unit, definition 2-1
Chain, directory 2-7
Chaining command files 10-3
Change I/O unit physical device correspondence 15-13
Changing
  directory names 3-5
  editor modes 4-4
  file names 3-5
  file names during copying 10-19
  record size 15-3
  working directory 3-2
CHARACTER data storage format D-3
Character set, ASCII C-1
  legal 14-1
Character string input, list directed 15-17
Characters, special terminal 2-5
Circular reasoning see proof by assumption
CLINEQ subroutine 19-1
CLOSE (PRIMOS command) 17-10
CLOSE ALL 17-10
Closing command input files 10-4
Closing command output files 10-6
Closing deck image files 4-2
Closing files 17-10
CM error 6-1
CMADD subroutine 19-2
CMADJ subroutine 19-2
CMCOF subroutine 19-3
CMCON subroutine 19-3
CMDET subroutine 19-4
CMDNC0 8-4
CMDSEG 8-5
CMIDN subroutine 19-4
CMINV subroutine 19-4
CMMLT subroutine 19-5
CMPF (PRIMOS command) 10-27
CMSCL subroutine 19-5
CMSUB subroutine 19-6
CMTRN subroutine 19-6
CNAM$$ subroutine 19-16
CNAME (PRIMOS command) 3-5
CO see COMINPUT
COBOL, interface to 12-8
Code, lines of, modifying 4-5
  moving lines of 4-5
  relative address 5-9
  segmented address 5-9
Codes, concordance 5-10
Codes, error 19-15
Coding statements 15-18
Coding strategy 9-1
Cofactor, matrix, subroutine 19-3
Collating sequence 19-7
Column 6 for continuation 14-2
Columns 73-80 14-2
COMB subroutine 19-1
Combination subroutine 19-1

COMINPUT (PRIMOS command) 10-2
  options 10-2
Command
  abbreviations 2-1
  external, definition 2-1
  file operations 10-1
  file requirements 10-2
  files, chaining 10-3
  files, CX 10-12
  files, errors in 10-4
  format conventions 2-4
  input files, closing 10-4
  internal, definition 2-2
  output files, closing 10-6
  sequences from files 10-1
  summary, editor 4-7
  summary, FUTIL 10-25
  summary, LOAD 6-9
  summary, SEG 7-8
  summary, SEG LOAD subprocessor 7-10
  summary, SEG MODIFY subprocessor 7-13
  UFD, installation of program in 8-4
  UFD, installing R-mode programs in 8-4
  UFD, installing V-mode programs in 8-5
Commands, FUTIL 10-18
Comment lines 14-2
  in SEG 7-1
  loader 6-1
  in command files 10-2
  overlaying 4-5
  use of 9-1
COMMON
  blank, relocating 11-5
  block F$IOBF 15-14
  block LIST 15-6
  blocks 15-6
  blocks (load map) 6-6, 7-6
  blocks over 64K words 11-12
  blocks, reserving space for 11-2
  locating 6-7
  locating (SEG) 7-8
  locating into specified segments 11-1
  location, deferring 6-7
  statement 15-6
  uninitialized, relocating 11-5
Common SEG command parameters 7-8
Common sort parameters 19-7
COMO see COMOUTPUT
COMO, use with TRACE 15-9
COMOUTPUT (PRIMOS command) 10-5
  file options 10-5
  terminal options 10-5
Companion, Programmer's 3-1
Comparing files 10-27
Compilation
  end of, message 5-2
  statements 15-8
  V-mode vs. R-mode 13-4
Compile error message 5-2
Compiler
  binary file (unit 3) 5-3, 17-9
  devices, default 17-9
  -DCLVAR usage 9-2
  -DYNM option, use of 13-7
  error messages A-1
  error messages, print at terminal 5-5
  error messages, suppress printing 5-5
  file specifications, table 17-2
  file unit usage 17-9
  FORTRAN, defaults B-2
  functions 5-3
  global trace 9-2
  input file 5-3
  input/output specifications 5-3
  invoking 5-1
  listing file (unit 2) 5-4, 17-9
  listing, default 5-6
  listing, enable 5-5
  listing, expanded 5-6
  listing, full 5-6
  object file (unit 3) 5-3, 17-9
  operations 5-11
  optimizing 5-12
  optimization 5-12
  option -32R 5-9
  option -64R 5-9
  option -64V 5-9
  option -BIG 5-10
  option -BINARY 5-3
  option -DCLVAR 5-11
  option -DEBASE 5-10
  option -DYNM 5-10
  option -ERRLIST 5-6
  option -ERRTTY 5-5
  option -EXPLIST 5-6
  option -FP 5-11
  option -INPUT 5-3
  option -INTL 5-11
  option -INTS 5-11
  option -LIST 5-6
  option -LISTING 5-4
  option -NOBIG 5-10
  option -NODCLVAR 5-11
  option -NOERRTTY 5-5
  option -NOFP 5-11
  option -NOTRACE 5-9
  option -NOXREF 5-8
  option -OPT 5-12
  option -PBECB 5-11
  option -SAVE 5-10
  option -SOURCE 5-3
  option -TRACE 5-9
  option -UNCOPT 5-12
  option -XREFL 5-8
  option -XREFS 5-8
  parameter combinations, prohibited 5-11
  parameter mnemonics, table 5-4
  parameters 5-3
  reference 17-1
  source file (unit 1) 5-3, 17-9
  syntax 5-1
  syntax checking 9-2
  description 1-7
Compiling 5-1
  for shared procedures 11-8
  from peripheral devices 5-3
  to peripheral devices 5-3
Complete cross reference 5-8
Completing a work session 3-8
COMPLEX data storage format D-3
COMPLEX mode 15-5
Complex numbers 14-4
Composition, program 14-8
Computed GO TO statement 15-11
Concatenating binary files 17-10
Concatenating listing files 17-10
Concepts, glossary 2-1
Concordance address, over 64K word COMMON 11-12
Concordance codes 5-10
Concordance see also cross reference
Concordances, compiler, enable 5-5
CONIOC 15-14
Conserve loader base areas 5-10
Constants 14-3
  address 14-6
  range 14-3
  system B-1
Contents
  directory, listing with FUTIL 10-25
  file, examining 3-6
  of directories 3-4
Continuation lines 14-2
CONTINUE statement 15-10
Control flow, conversion 1-4
  flow, program, monitoring 9-2
  key 2-5
  lines 14-2
  load placement 11-2
  statements 15-10
CONTROL-P, usage 2-6
CONTROL-Q, usage 2-6
CONTROL-S, usage 2-6
Controlling file access 3-8
Conventions,
  command format 2-4
  filename 2-2
  glossary 2-1
Conversion, control flow 1-4
  functions 1-4
  input/output 1-4
  program 1-4
  source language 1-4
  subroutines 1-4
Copies, file, obtaining 3-6
Copying
  directories 10-19
  directories trees 10-19
  files 10-18
  files and changing names 10-19
  tapes with MAGNET 10-15
  tapes with MAGRST/MAGSAV 10-15
  UFDs 10-19
  with FUTIL 10-19
CPU, definition 2-1
CR (in B format) 15-23
CREATE (PRIMOS command) 3-3
Creating MIDAS template 12-2

  new directories 3-3
  new programs 4-4
  R-mode runfiles 11-10
CREATK (MIDAS utility) 12-2
  dialogue 12-5
CRMPC (PRIMOS command) 4-2
Cross reference codes 5-9
  see also concordance
  compiler, enable 5-5
  complete 5-8
  example 5-8
  explanation 5-8
  full 5-8
  partial 5-8
  short 5-8
  suppression 5-8
Current directory, definition 2-1
Current disk 2-10
CX (PRIMOS command) 10-11
  command files 10-12
  options 10-11
  queue information 10-12
  queue, dropping jobs from 10-13
  see also sequential job processor

D

D input format 15-22
D output format 15-20
D/ prefix 11-13
DAM files 19-14
  see also direct access method
Data definition
  mode convention, FORTRAN, overriding 15-4
  mode of function 15-3
  mode rules for assignment statements, table 15-11
  mode typing, parameter 15-6
  modes 15-5
  segment 11-2
  statement 15-8
DATA statement 15-8
Data storage format,
  ASCII D-3
  CHARACTER D-3
  COMPLEX D-3
  DOUBLE PRECISION D-2
  INTEGER*2 D-2
  INTEGER*4 D-2
  LOGICAL D-2
  REAL*4 D-2
Data storage, ANSI standard D-1
Data subfile, MIDAS, building 12-2
Data types 15-5
  FORTRAN, memory formats D-1
Database Management System see also DBMS
  description 1-8
  interface to 12-6
DATE (PRIMOS command) 10-6
Date/time stamping of output files 10-6
DBMS see also Database Management System
DBMS, description 1-8
DCLVAR (compiler option) 5-11, 17-1
DEBASE (compiler option) 5-10, 17-1
Debugging 9-1
DECODE, formatted, statement 15-18
  list directed, statement 15-18
  Decreasing number of segments 11-9
Default
  characters, editor 4-10
  compiler devices 17-9
  compiler listing 5-6
  object code 5-9
  protection keys 3-8
  record size 15-13
Defaults,
  A register 17-6
  B register 17-6
  ED B-1
  editor B-1
  execution B-2
  FORTRAN compiler B-2
  FTN B-2
  LOAD B-1
  loader B-1
  PRIMOS B-2
  SEG loader B-2
  segmented loader B-2
  system B-1
  terminal B-1
DEFER (SPOOL option) 3-7
Deferring COMMON location 6-7
Deferring spool printing 3-7
Definitions 2-1
DELET$, MIDAS subroutine 12-5
DELETE (PRIMOS command) 3-8
Deleting
  directories 3-3, 10-24
  directory trees 10-24
  files 3-8, 10-18
  MIDAS files 12-6
  programs 4-11
  UFDs 10-24
  with FUTIL 10-24
Delimiters, format 15-19
Delimiters, list directed 15-16
DELSEG (PRIMOS command) 7-7
Descriptor repetition 15-19
Details, loading 6-6
Determinant subroutine 19-4
Determining file size 3-6
Development, program 1-3
Device assignment, queuing 4-1
Device control statements 15-25
Device see also disk
Device, assigning 4-1
Device-bit correspondence, compiler 17-9
Devices, compiler, default 17-9
Devices, unassigning 4-1
Dialogue, CREATK 12-5
Dialogue, KBUILD 12-4
DIMENSION statement 15-6
Dimensioning, not allowed in SAVE statement 15-7
Diminishing increment sort subroutine 19-8
Direct access 15-12
  and ATTDEV subroutine 15-13
  and the Editor 15-13
  IBM compatibility 15-13
  READ statements 15-15
  WRITE statements 15-17
  use of 15-13
Direct entry links 7-6
Directories,
  creating 3-3
  deleting 3-3
  remote, attaching to 10-18
Directory
  chain 2-7
  contents 3-4
  contents, listing with FUTIL 10-25
  current, definition 2-1
  definition 2-1
  examining contents 3-4
  file, master, definition 2-3
  home vs. current 2-7
  home, definition 2-2
  name, definition 2-1
  names, changing 3-5
  operations 3-2
  passwords, assigning 3-3
  segment, definition 2-4
  structures 2-6
  user file, definition 2-4
  working, changing 3-2
Disk
  see also device
  current 2-10
  logical, definition 2-3
  physical, definition 2-3
DLINEQ subroutine 19-1
DMADD subroutine 19-2
DMADJ subroutine 19-2
DMCOF subroutine 19-3
DMCON subroutine 19-3
DMDET subroutine 19-4
DMIDN subroutine 19-4
DMINV subroutine 19-4
DMMLT subroutine 19-5
DMSCL subroutine 19-5
DMSUB subroutine 19-6
DMTRN subroutine 19-6
DO loop
  index 15-10
  optimization 5-12, 13-1
  one-trip 15-10
DO loops, implied 15-18
  nesting 15-10
DO statement 15-10
Documents, related 1-2
DOUBLE PRECISION data storage format D-2
DOUBLE PRECISION mode 15-5
Double precision numbers 14-4
DOUBLE PRECISION see also REAL*8
Double-quote, usage 2-5
Dropping jobs from CX queue 10-13
Dummy argument arrays, over 64K word COMMON 11-12
Duplicating magnetic tapes 10-15
  runfiles 11-6
Dynamic allocation of local storage 5-10

- DYNM (compiler option) 5-10, 17-1
  - option, compiler, use of 13-7

E

- E input format 15-22
- E output format 15-20
- EBCDIC card decks, reading 4-2
- EBCDIC magnetic tape, reading from 4-3
- ECBs, load into procedure frame 5-11
- ED (PRIMOS command) 4-4
  - defaults B-1
- Edit mode, editor 4-4
- Editing session, sample 4-5
- Editor
  - command summary 4-7
  - defaults 4-10, B-1
  - description 1-8
  - edit mode 4-4
  - input mode 4-4
  - modes, changing 4-4
  - special characters 4-5
  - symbol names 4-10
  - techniques 4-5
  - usage of " 4-5
  - usage of ; 4-5
  - usage of ? 4-5
  - usage of / 4-5
  - text 4-4
  - use of on direct access files 15-13
- Ellipsis, convention 2-4
- Enable compiler concordances 5-5
- Enable compiler cross references 5-5
- Enable compiler listings 5-5
- Enable flagging of undeclared variables 5-11
- Enable global trace 5-9
- ENCODE statement 15-19
- End of compilation message 5-2
- END statement 15-11
- END= 15-14
- ENDFILE statement 15-25
- Ending main program 1-4
- Entering programs from other media 4-1
- Entering source programs 4-1
- Entry control block 7-6
- Environment, interactive, description 1-5
- Environment,
  - phantom user, description 1-5
  - program, list 1-5
  - sequential job processing, description 1-5
- Equations, linear, subroutine 19-1
- EQUIVALENCE statement 15-7
- ER! prompt 2-6
- ERR= 15-14
- ERRD.F 19-15
- ERRLIST (compiler option) 5-6, 17-2
- Error codes 19-15
- Error handling,
  - file 19-15
  - loader 6-1
  - loader, table 6-9
  - SEG 7-1
- Error message, compiler 5-2
- Error messages A-1
  - base areas 6-7
  - compiler A-1
  - compiler, print at terminal 5-5
  - compiler, print only 5-6
  - file system A-8
  - FORTRAN library A-8
  - loader A-5
  - run-time 8-2, A-8
  - SEG loader A-6
- Errors in command files 10-4
- Errors, file system 8-3
  - FORTRAN function, R-mode 8-2
  - FORTRAN function, V-mode 8-3
- ERRTTY (compiler option) 5-5, 17-2
- Examining file contents 3-6
- Examples, conventions 2-5
- Executing programs 8-1
- Execution
  - defaults B-2
  - from SEG's loader 7-2
  - of R-mode memory images 8-1
  - of R-mode programs 8-1
  - of segmented runfiles 8-2
  - of V-mode programs 8-2
  - program, from the loader 6-2
- Exit, normal 15-10
- Expanded compiler listing 5-6
- EXPLIST (compiler option) 5-6, 17-2
- Extended intrinsic functions 16-1
  - range, optimization 5-12
  - segmented program techniques 11-1
- Extension stack segments 11-6
- Extensions 1-4
- Extent 7-4
- External command, definition 2-1
- External procedure statements 15-7
- EXTERNAL statement 15-7

F

- F input format 15-22
- F output format 15-20
- F$IOBF COMMON block 15-14
- F/ modifer 11-4
- FALSE 14-4
- Field descriptor, format 15-19
- File
  - access 19-14
  - access, controlling 3-8
  - action keys 19-15
  - binary, definition 2-1
  - command, operations 10-1
  - comparison 10-27
  - contents, examining 3-6
  - copies, obtaining 3-6
  - copying 10-18
  - definition 2-1
  - deleting 10-18
  - directory, master, definition 2-3
  - directory, user, definition 2-4
  - error handling 19-15
  - hierarchy 2-6
  - listing 10-18
  - manipulation 10-27
  - names 19-14
  - names, changing 3-5
  - object, definition 2-3
  - operations 3-4
  - protection keys, definition 2-3
  - size, determining 3-6
  - specifications, compiler, table 17-2
  - source, definition 2-4
  - structures 2-6
  - system error messges A-8
  - system errors 8-3
  - system summary 1-5
  - types, PRIMOS, table 2-8
  - unit usge, compiler 17-9
- File-unit, definition 2-2
- Filename conventions 2-2
- Filename, definition 2-2
- Files,
  - DAM 19-14
  - deleting 3-8
  - incorporating into shared segments 11-12
  - object (SEG) 7-8
  - printing 3-6
  - SAM 19-14
  - saving 4-5
  - sorting 10-28
  - text, merging 10-28
- FILMEM (PRIMOS command) 6-2
- FIND$, MIDAS subroutine 12-5
- Finding line numbers 4-5
- Floating point skip operations, generate 5-11
- Floating point skip operations, suppress 5-11
- Forceloading 11-4
- FORM (SPOOL option) 3-8
- Format
  - delimiters 15-19
  - field descriptor 15-19
  - lines, rescanning 15-19
  - statement 15-19
  - command, conventions 2-4
  - line 14-1
  - use of parameters not allowed 15-6
- Formats
  - as variables 15-21
  - in input statements, table 15-22
  - in output statements, table 15-20
  - memory, FORTRAN data types D-1
- Formatted
  - DECODE statement 15-18
  - printer control 15-24
  - READ statement 15-15
  - WRITE statement 15-17
- Forms Management System see also FORMS
- Forms management system, interface to 12-6
- FORMS see also Forms Management System

  description 1-8
Forms, special, printing on 3-7
FORTRAN
  compiler defaults B-2
  data mode convention, overriding 15-4
  data types, memory formats D-1
  extensions, Prime 1-4
  function errors, R-mode 8-2
  function errors, V-mode 8-3
  function library 18-1
  function reference 18-1
  functions 16-1
  functions, list 18-2
  language elements 14-1
  language tutorial books 1-1
  library error messages A-8
  library functions 16-1
  library, V-mode 18-1
  math library 19-1
  math subroutines 16-3
  mathematical functions, table 1-6
  matrix library 19-1
  Prime's, overview 1-1
  statements 15-1
  under PRIMOS 1-4
  unit number, physical devices, table 15-15
FP (compiler option) 5-11, 17-2
Frame,
  link 7-6
  procedure 7-6
  stack 7-6
FTN
  (PRIMOS command) 5-1
  (SPOOL option) 4-10
  defaults B-2
  FORTRAN compiler 5-1
FTNLIB 18-1
FTNOPT (PRIMOS command) 5-12
Full compiler listing 5-6
Full cross reference 5-8
FULL LIST statement 15-8
Function
  arguments 16-2
  calls 15-25
  calls, optimization 13-4
  mode 15-3
  mode typing 16-1
  reference, FORTRAN 18-1
  rules 16-2
FUNCTION statement 15-3, 16-1
Function subprograms, user-defined 16-1
Function, structure of 16-1
Functions,
  compiler 5-3
  conversion 1-4
  extended intrinsic 16-1
  FORTRAN 16-1
  FORTRAN library 16-1
  FORTRAN, list 18-2
  statement 16-2
FUTIL
  (PRIMOS command) 10-18
  command summary 10-25
  commands 10-18
  commands, overview, figure 10-20
  prompt character > 10-18
  invoking 10-18

G

G input format 15-22
G output format 15-20
Generalized subscripts 14-5
Generate floating point skip operations 5-11
Global
  mode specification 15-5
  SAVE 15-7
  trace, enable 5-9
  trace, suppress 5-9
Global/IMPLICIT conflict 15-5
Glossary, concepts and conventions 2-1
GO TO,
  assigned, statement 15-11
  computed, statement 15-11
  unconditional, statement 15-11

H

H output format 15-20
HARDWARE (LOAD subcommand) 6-8
Hardware requirements for loading 6-8
Hardware table 6-10
Header statements for subprograms 15-3
HEAP subroutine 19-8
Heapsort subroutine 19-8
Hierarchy of files 2-6
HIGH 7-4
Hollerith constants 14-4
Home directory, definition 2-2
Home spool queue 10-18
Home vs. current directory 2-7
Housekeeping, MIDAS file 12-6
Hyphen, convention 2-4

I

I input format 15-22
I output format 15-21
I/O unit physical device correspondence, change 15-13
IBM compatibility, direct access files 15-13
Identity, definition 2-2
  matrix, subroutine 19-4
IF arithmetic, statement 15-12
  logical vs. arithmetic 13-6
  logical, statement 15-12
  statements, optimization 13-5
IFTNLB 18-1
IMADD subroutine 19-2
IMADJ subroutine 19-2
IMCOF subroutine 19-3
IMCON subroutine 19-3
IMDET subroutine 19-4
IMIDN subroutine 19-4
IMMLT subroutine 19-5
Implementation, over 64K word COMMON 11-14
Implemented statements, list 15-1
Implicit parameter assignment 11-3
IMPLICIT statement 15-4
IMPLICIT/global conflict 15-5
Implied DO loops 15-18
Important load commands 6-2
Important SEG subcommands 7-2
IMSCL subroutine 19-5
IMSUB subroutine 19-6
IMTRN subroutine 19-6
In-line comments, use of 9-2
Including R-mode interlude in SEG runfile 11-10
Incorporating files into shared segments 11-12
Indention, use of 9-2
Index, DO loop 15-10
Information, system, table 3-5
INITIALIZE (SEG's loader) 11-5
Initializing a load 11-5
Initiating phantoms from a phantom 10-8
INPUT (compiler option) 5-3, 17-3
Input file, compiler 5-3
Input
  mode, editor 4-4
  scale factors 15-24
  specifications, compiler 5-4, 17-7
  statements 15-12
  statements, formats in, table 15-22
Input/output optimization 13-5
  for conversion 1-4
INSERT see $INSERT
INSERT subroutine 19-8
Insertion sort subroutine 19-8
Installation of programs in command UFD 8-4
Installing R-mode programs in command UFD 8-4
Installing V-mode programs in command UFD 8-5
Integer division optimization 13-6
INTEGER mode 15-5
Integer random number generator 18-5
INTEGER see also INTEGER*2, INTEGER*4
Integer sign extension 18-1
Integer truncation 18-1
INTEGER*2
  data storage format D-2
  default 5-11
  mode 15-5
  see also INTEGER, INTEGER*4
INTEGER*4
  data storage format D-2
  default 5-11
  mode 15-5
  see also INTEGER, INTEGER*4
Integers 14-3
  in subroutine calls 17-3
  long 14-3
  short 14-3
Interactive environment, description 1-5
Interchange sort subroutine 19-8
Interface
  to assembly language 12-8
  to COBOL 12-8
  to database management system 12-6

  to FORMS management system 12-6
  to other languages 12-1
  to PMA 12-8
  to systems 12-1
Interfaces, languages, description 1-8
Interlude program 8-5
  R-mode, including in SEG runfile 11-10
Internal command, definition 2-2
INTL (compiler option) 5-11, 17-3
Intrinsic functions, extended 16-1
Intrpt key 2-5
INTS (compiler option) 5-11, 17-3
Inversion, matrix, subroutine 19-4
Item TRACE statement 15-8

J
Job file number 10-11
  ID 10-12
  number, definition 3-2

K
KBUILD (MIDAS utility) 12-2
  dialogue 12-4
Keyboard input, ASCII characters C-1
Keys, file 19-15
Keys, file protection, definition 2-3
Keys, protection, default 3-8
Keys, special terminal 2-5
KEYS.F 19-15
KIDDEL (MIDAS utility) 12-6

L
L input format 15-22
L output format 15-21
Language
  elements, FORTRAN 14-1
  interfaces, description 1-8
  source, conversion 1-4
LDEV, definition 2-3
Ldisk, definition 2-3
Legal character set 14-1
Libraries reference 19-1
Libraries, description 1-7
LIBRARY (SEG's loader) 11-2
Library calls
  applications 19-9
  FORTRAN function 18-1
  FORTRAN math 19-1
  FORTRAN matrix 19-1
  functions, FORTRAN 16-1
  optimization 13-6
  operating system 19-14
  search 19-7
  sort 19-7
  subroutines, applications, list 19-10
  subroutines, loading 6-3
  subroutines, loading (SEG) 7-3
Line
  format 14-1
  numbers 4-5
  numbers, finding 4-5
  printer listing of programs 4-10
Linear equations subroutine 19-1
LINEQ subroutine 19-1
LINK FR. 7-6
Link frame 7-6
Link segment 11-2
LIST (COMMON block) 15-6
LIST (compiler option) 5-6, 17-3
LIST (SPOOL option) 3-6
List directed
  character string input 15-17
  DECODE statement 15-18
  delimiters 15-16
  numerical input 15-16
  READ statement 15-16
LIST statement 15-8
List, FORTRAN functions 18-2
LISTF (PRIMOS command) 3-4
LISTING (compiler option) 5-4, 17-4
LISTING (PRIMOS command) 17-10
Listing
  directory contents with FUTIL 10-25
  file, compiler (unit 2) 5-4, 17-9
  file, spooling 5-5
  files 10-18
  files, concatenating 17-10
  programs 4-10
  programs at terminal 4-10
  programs on line printer 4-10
  spool queue 3-6
  compiler, default 5-6
  compiler, expanded 5-6
  full, compiler 5-6
Listings, compiler, enable 5-5
LNUM (SPOOL option) 4-10
LOAD (PRIMOS command) 6-1
  (SEG's loader) 11-2
  command summary 6-9
  commands, important 6-2
LOAD COMPLETE 6-2, 7-2
Load ECBs into procedure frames 5-11
Load map
  (LOAD) 6-3
  (LOAD), example 6-5
  (SEG) 7-3
  (SEG), example 7-5
  (SEG), types 7-4
  types (LOAD) 6-10
  types (SEG) 7-9
Load maps, type 6-4
Load modules, replacing 11-6
Load placement control 11-2
Load sequence 6-2
Load sequence, optimization 13-3
Load state 6-4
LOAD
  subcommand HARDWARE 6-8
  subcommand MAP 6-3
  subcommands, use of pathnames in 6-7
  subprocessor command summary, SEG 7-10
  subprocessor, SEG, advanced features 11-1
  defaults B-1
  error messages A-5
Load, initializing 11-5
Loader commands, SEG's 11-2
Loader
  defaults B-1
  error handling 6-1
  error handling, table 6-9
  error messages A-5
  functions 6-6
  prompt $ 6-1
  conservation of base areas 5-10
  description 1-7
  SEG 7-7
  SEG, defaults B-2
  SEG, error messages A-6
  SEG, execution from 7-2
  segmented, defaults B-2
  usage 6-1
Loading
  details 6-6
  for shared procedures 11-8
  library subroutines 6-3
  library subroutines (SEG) 7-3
  main program 6-3
  modes 6-11
  R-mode programs 6-1
  RUNIT into segment '4000 11-9
  segmented programs 7-1
  sequence (SEG) 7-2
  templates 11-4
  to specific segments 11-3
  UII library 6-8
  V-mode programs 7-1
  normal 6-2
  normal (SEG) 7-2
  order of 6-3
  order of (SEG) 7-3
  virtual 6-6
Loadmap, use of 7-3
Loads, partial 11-4
Local processor 10-17
Local storage, dynamic allocation 5-10
Local storage, static allocation 5-10
Locating
  COMMON 6-7
  (SEG) 7-8
  into specified segments 11-1
  see also relocating COMMON
Location, stack, RUNIT 11-9
LOCK$, MIDAS subroutine 12-5
Log in 3-2
Log out 3-8
Logging in 3-2
Logging out 3-8
Logical
  constants 14-4
  disk definition 2-3
  functions, mixed integers in 18-2
  IF statement 15-12
  mode 15-5
  operators 14-6
  shift operator 18-7
  vs. Arithmetic IF 13-6
LOGICAL, data storage format D-2
LOGIN (PRIMOS command) 3-2
Login, remote 10-17
LOGOUT (PRIMOS command) 3-8

Logout, automatic 3-9
Logout, phantom 10-8
Long and short integers, mixing 18-1
Long integers 14-3
LOW 7-4
Lower case convention 2-4

M

MADD subroutine 19-2
MADJ subroutine 19-2
MAGNET (PRIMOS command) 4-3
MAGNET, copying tapes with 10-15
Magnetic tape
  ASCII, reading from 4-3
  BCD, reading from 4-3
  binary, reading from 4-3
  duplicating 10-15
  EBCDIC, reading from 4-3
  reading from 4-3
  utilities 10-15
MAGRST (PRIMOS command) 10-15
MAGRST dialogue summary 10-15
MAGRST/MAGSAV, copying tapes with 10-15
MAGSAV (PRIMOS command) 10-16
MAGSAV dialogue summary 10-16
MAGSAV/MAGRST, copying tapes with 10-15
Main program, ending 1-4
  loading 6-3
Maintaining MIDAS files 12-5
Manipulating source programs 4-1
MAP (LOAD subcommand) 6-3
MAP (SEG subcommand) 7-3
Map see also load map
Master file directory, definition 2-3
Math library, FORTRAN 19-1
Math subroutines, FORTRAN 16-3
Mathematical functions, FORTRAN, table 1-6
MATHLB 19-1
Matrix
  addition subroutine 19-2
  adjoint subroutine 19-2
  cofactor subroutine 19-3
  identity subroutine 19-4
  inversion subroutine 19-4
  library, FORTRAN 19-1
  multiplication subroutine 19-5
  subroutines, table 1-6
  subtraction subroutine 19-6
  transpose subroutine 19-6
MCOF subroutine 19-3
MCON subroutine 19-3
MDET subroutine 19-4
Memory
  allocation, optimization 13-3
  formats, FORTRAN data types D-1
  images, R-mode, execution of 8-1
  usage 5-9
Merging text files 10-28
Message, end of compilation 5-2
Message, error, compiler 5-2
Messages,
  compiler A-1
  error A-1
  file system A-8
  FORTRAN library A-8
  loader A-5
  run-time 8-2, A-8
  SEG loader A-6
MFD, definition 2-3
MIDAS
  advantages of 12-1
  description 1-8
  data subfile, building 12-2
  file housekeeping 12-6
  files, deleting 12-6
  files, maintaining 12-5
  parameter file PARM.K 12-5
  requirements for 12-1
  see also Multiple Index Direct Access System
  subroutine ADD1$ 12-5
  subroutine BILD$R 12-2
  subroutine DELET$ 12-5
  subroutine FIND$ 12-5
  subroutine LOCK$ 12-5
  subroutine NEXT$ 12-5
  subroutine PRIBLD 12-2
  subroutine SECBLD 12-2
  subroutine UPDAT$ 12-5
  template, creating 12-2
  template, modifying 12-2
  usage 12-1
  utility CREATK 12-2
  utility KBUILD 12-2
  utility KIDDEL 12-6
MIDN subroutine 19-4
MINV subroutine 19-4
Mixed integers in logical functions 18-2
Mixed mode arithmetic 15-9
Mixing long and short integers 18-1
MMLT subroutine 19-5
Mnemonic-bit correspondence, A register 17-8
Mnemonic-bit correspondence, B register 17-8
MODE D64R, preference for 6-7
Mode
  mixing rules 15-9
  of function 15-3
  specification statement 15-5
  specification, global 15-5
  typing, function 16-1
  addressing 6-7
  data see data type
  definition 2-3
Modes, data 15-5
Modes, loading 6-11
Modification subprocessor 11-6
MODIFY (SEG command) 11-6
MODIFY subprocessor command summary, SEG 7-13
Modifying
  lines of code 4-5
  MIDAS template 12-2
  programs 4-4
Modular program structure 9-1
Modules, replacing 11-6
Monitoring program control flow 9-2
Moving lines of code 4-5
MRGF (PRIMOS command) 10-28
MSCL subroutine 19-5
MSORTS 19-7
MSUB subroutine 19-6
MTRN subroutine 19-6
Multi-dimensioned arrays, optimization 13-3
Multiple Index Direct Access System see also MIDAS
Multiplication,
  matrix, subroutine 19-5
  scalar, subroutine 19-5

N

Name, call by 15-4
Name, directory, definition 2-1
Names, file 19-14
Nesting DO loops 15-10
Nesting, not allowed in $INSERT files 15-9
Network status 10-17
Networks 10-17
New programs, creating 4-4
NEXT$, MIDAS subroutine 12-5
NO LIST statement 15-8
NOBIG (compiler option) 5-10, 17-4
NODCLVAR (compiler option) 5-11, 17-4
Nodename 10-17
Nodename, definition 2-3
NOERRTTY (compiler option) 5-5, 17-4
NOFP (compiler option) 5-11, 17-4
Non-owner
  password 3-3
  rights 3-8
  status 3-3
Non-printing ASCII characters C-2
NONOWN 3-4
Normal exit 15-10
Normal loading 6-2
Normal loading (SEG) 7-2
NOT truth table 14-7
NOTRACE (compiler option) 5-9, 17-4
NOXREF (compiler option) 5-8, 17-4
NULL 3-4
Number representations 2-3
Number, job, definition 3-2
Number, user 3-2
Numbers, line 4-5
Numerical input, list directed 15-16

O

Object code 6-7
  generation 5-9
  default 5-9
Object file,
  compiler (unit 3) 5-4, 17-9
  definition 2-3
  (SEG) 7-8
Obtaining file copies 3-6
OK, prompt 2-6

- OK: prompt 2–6
- One-trip DO loop 15–10
- Open, definition 2–3
- Operands 14–2
- Operating system
  - features 10–1
  - library 19–14
  - subroutines, list 19–15
- Operations, directory 3–2
- Operations, file 3–4
- Operator priority 14–7
- Operators 14–6
  - arithmetic 14–7
  - logical 14–6
  - relational 14–7
- OPT (compiler option) 5–12, 17–4
- Optimization 13–1
  - 64V-mode COMMON 13–4
  - compiler 5–12
  - DO loops 5–12, 13–1
  - function calls 13–4
  - IF statements 13–5, 13–6
  - input/output 13–5
  - integer division 13–6
  - library calls 13–6
  - load sequence 13–3
  - memory allocation 13–3
  - multi-dimensioned arrays 13–3
  - parameter statements 13–6
  - statement functions and subroutines 13–6
  - statement numbers 13–3
  - statement sequence 13–5
  - unconditional 5–12
- Optimizing compiler 5–12
- Optimizing load for shared procedures 11–9
- Option, convention 2–5
- Options, compiler see also parameters, compiler
- OR truth table 14–7
- Order of loading 6–3
  - (SEG) 7–3
- Order of statements in a program 14–8
- Ordinary pathname 2–7
- Organization 1–1
- Other languages, interface to 12–1
- Other media, entering programs from 4–1
- Output scale factors 15–24
  - specifications, compiler 5–4, 17–7
  - statements 15–12
  - statements, formats in, table 15–20
  - stream, definition 2–3
- Output/input optimization 13–5
- Over 64K word COMMON blocks 11–12
  - arrays 11–12
  - concordance address 11–12
  - dummy argument arrays 11–12
  - implementation 11–14
  - programming considerations 11–14
  - restrictions 11–13
- Overlaying comments 4–5
- Overriding FORTRAN data mode convention 15–4
- Overview of FUTIL commands, figure 10–20
  - Prime's FORTRAN 1–1
  - PRIMOS 2–1
- OWNER 3–4
- Owner password 3–3
  - rights 3–8
  - status 3–3

- Packname, definition 2–3
- Page, definition 2–3
- Paper tape, punched, reading from 4–4
- Parameter
  - assignment, implicit 11–3
  - combinations, compiler, prohibited 5–11
  - compiler 5–3
  - data mode typing 15–6
  - statement 15–5
  - statements optimization 13–6
  - usage 15–5
- SEG command, common 7–8
  - Parameters 14–5
- compiler see also options, compiler
- not allowed in FORMAT statement 15–6
- Parentheses, convention 2–4
- Parity, ASCII C–1
- PARM.K, MIDAS parameter file 12–5
- Partial cross reference 5–8
- Partial loads 11–4
- Partition exchange sort subroutine 19–8
- Partition, definition 2–3
- PASSWD (PRIMOS command) 3–3
- Password, non-owner 3–3
- Password, owner 3–3
- Passwords 19–14
- Passwords in FUTIL 10–18
- Passwords in pathnames 3–2
- Passwords, assigning directory 3–3
- Pathname vs. filename 2–7
- Pathname, definition 2–3
- Pathname, ordinary 2–7
- Pathname, relative 2–8
- Pathnames 2–7
- Pathnames in LOAD subcommands 6–7
- Pathnames in SEG commands 7–8
- Pathnames with passwords 3–2
- PAUSE statement 15–12
- PAUSE, recovering from 15–12
- PBECB (compiler option) 5–11, 17–4
- PDEV, definition 2–3
- Pdisk, definition 2–3
- Peripheral devices with compiler 5–3
- PERM subroutine 19–6
- Permutation subroutine 19–6
- Petitio principii see circular reasoning
- PFTNLB 18–1
- PHANTOM
  - (PRIMOS command) 10–8
  - Phantom abort 10–8
  - logout 10–8
  - operation 10–8
  - status information 10–9
  - user environment, description 1–5
  - user, definition 2–4
  - users 10–8
- Physical device FORTRAN unit numbers, table 15–15
- Physical device I/O unit correspondence, change 15–13
- Physical disk, definition 2–3
- PMA see also Prime Macro Assembly language
- PMA, interface to 12–8
- PRIBLD, MIDAS subroutine 12–2
- Prime extensions to FORTRAN 1–4
- Prime Macro Assembly language see also PMA
- Prime usage, ASCII C–1
- PRIMOS commands
  - ASSIGN 4–1
  - ATTACH 3–2
  - BINARY 17–10
  - CLOSE 17–10
  - CMPF 10–27
  - CNAME 3–5
  - COMINPUT 10–2
  - COMOUTPUT 10–5
  - CREATE 3–3
  - CRMPC 4–2
  - CX 10–11
  - DATE 10–6
  - DELETE 3–3, 3–8
  - DELSEG 7–7
  - ED 4–4
  - FILMEM 6–2
  - FTN 5–1
  - FTNOPT 5–12
  - FUTIL 10–18
  - LISTF 3–4
  - LISTING 17–10
  - LOAD 6–1
  - LOGIN 3–2
  - LOGOUT 3–8
  - MAGNET 4–3
  - MAGRST 10–15
  - MAGSAV 10–16
  - MRGF 10–28
  - PASSWD 3–3
  - PHANTOM 10–8
  - PROTEC 3–8
  - RESUME 8–1
  - SEG 7–1, 8–2
  - SIZE 3–6
  - SLIST 3–6
  - SORT 10–28
  - SPOOL 3–6
  - START 8–1
  - TERM 10–30
  - TIME 10–6
  - UNASSIGN 4–1
- PRIMOS
  - defaults B–2
  - file types, table 2–8
  - FORTRAN under 1–4
  - II 2–6
  - in networks 10–17
  - overview 2–1
  - system subroutines 16–3
  - tree-structured file system 2–9

- Print compiler error messages at terminal 5-5
- Print only error messages 5-6
- PRINT statement 15-14
- Printer control, formatted 15-24
- Printing
  - ASCII characters C-3
  - deferring 3-7
  - files 3-6
  - on special forms 3-7
- Priority of operators 14-7
- Procedure
  - frame 7-6
  - frames, load ECBs into 5-11
  - segment 11-2
- Program
  - composition 14-8
  - control flow, monitoring 9-2
  - conversion 1-4
  - development 1-3
  - environments, list 1-5
  - execution from SEG's loader 7-2
  - execution from the loader 6-2
  - installation in command UFD 8-4
  - structure, modular 9-1
  - techniques, extended segmented 11-1
  - order of statements in 14-8
- Programmer's Companion 3-1
- Programming considerations, over 64K word COMMON 11-14
- Programs
  - creating 4-4
  - deleting 4-11
  - entering from other media 4-1
  - entry from terminal 4-4
  - executing 8-1
  - in memory, restarting 8-1
  - listing 4-10
  - modifying 4-4
  - R-mode, execution of 8-1
  - R-mode, loading 6-1
  - renaming 4-11
  - segmented, loading 7-1
  - source, entering 4-1
  - source, manipulating 4-1
  - terminal entry 4-4
  - V-mode, execution of 8-2
  - V-mode, loading 7-1
- Prohibited compiler parameter combinations 5-11
- Prompts, system 2-6
- Proof by assumption see petitio principii
- PROTEC (PRIMOS command) 3-8
- Protection keys, default 3-8
- PRWF$$ subroutine 19-17
- Punched cards, reading 4-2
- Punched paper tape, reading from 4-4

Q

- Question mark, usage 2-5
- Queue information,
  - CX 10-12
  - CX, dropping jobs from 10-13
  - spool, listing 3-6
- Queuing device assignment 4-1
- QUICK subroutine 19-8

R

- R-mode
  - FORTRAN function errors 8-2
  - interlude, including in SEG runfile 11-10
  - memory images, execution of 8-1
  - programs, execution of 8-1
  - programs, installation in command UFD 8-4
  - programs, loading 6-1
  - runfiles, creating 11-10
  - vs. V-mode compilation 13-4
- Radix exchange sort subroutine 19-8
- RADXEX subroutine 19-8
- Random number generator, integer 18-5
- Random number generator, real 18-6
- Range of constants 14-3
- READ statements 15-14
  - binary, statement 15-16
  - direct access, statements 15-15
  - formatted, statement 15-15
  - list directed, statement 15-16
- Read/write lock table 10-27
- Reading
  - ASCII card decks 4-2
  - BCD card decks 4-2
  - EBCDIC card decks 4-2
  - from 7-track tape 4-3
  - from 9-track tape 4-3
  - from ASCII magnetic tape 4-3
  - from BCD magnetic tape 4-3
  - from binary magnetic tape 4-3
  - from EBCDIC magnetic tape 4-3
  - from magnetic tape 4-3
  - from punched paper tape 4-4
  - punched cards 4-2
- REAL mode 15-5
- Real numbers 14-4
- Real random number generator 18-6
- REAL see also REAL *4
- REAL *4
  - data storage format D-2
  - mode 15-5
  - see also REAL
- REAL *8 mode 15-5
  - see also DOUBLE PRECISION
- REC= 15-15, 15-17
- Record size
  - changing 15-3
  - default 15-13
  - over 128 words 15-14
- Recovering from PAUSE 15-12
- Recursive subroutines 15-7
- Reference, compiler 17-1
- Related documents 1-2
- Relational operators 14-7
- Relative
  - address code 5-9
  - pathname 2-8
  - segment assignment 11-1
- Releasing assigned segments 7-7
- Relocating
  - blank COMMON 11-5
  - COMMON see also locating COMMON
  - uninitialized COMMON 11-5
- Remote
  - directories, attaching to 10-18
  - login 10-17
  - processor 10-17
- Renaming programs 4-11
- Repetition, field descriptor 15-19
- Replacing modules 11-6
- Representation,
  - ASCII character strings 14-4
  - complex numbers 14-4
  - double precision numbers 14-4
  - real numbers 14-4
- Representations, number 2-3
- Requirements for MIDAS 12-1
- Requirements, command file 10-2
- Rescanning format lines 15-19
- Reserving space for COMMON blocks 11-2
- Resources, system, list 1-5
- Restarting programs in memory 8-1
- Restarting segmented programs 8-2
- Restrictions on over 64K word COMMON 11-13
- RESU$$ subroutine 19-18
- RESUME (PRIMOS command) 8-1
- Return key 2-5
- RETURN statement 15-12
- REWIND statement 15-25
- Rights, non-owner 3-8
- Rights, owner 3-8
- RL (SEG's loader) 11-2, 11-6
- Rubout key 2-5
- Rules
  - for functions 16-2
  - for subroutines 16-3
  - for variables 14-5
  - mode mixing 15-9
- Run-time error messages 8-2, A-8
- Run-time statements 15-8
- Runfile, definition 2-4
  - SEG , including the R-mode interlude 11-10
- Runfiles 6-7
  - duplicating 11-6
  - segmented 7-7
  - segmented, execution of 8-2
- RUNIT program 11-8
- RUNIT stack location 11-9
- Rust color, convention 2-5

S

- S/ prefix 11-3
- SAM files 19-4
- SAM see also sequential access method
- Sample editing session 4-5
- SAVE (compiler option) 5-10, 17-5
- SAVE (SEG command) 11-6
- SAVE statement 15-7
- SAVE statement, dimensioning not allowed in 15-7
- SAVE, global 15-7
- Saving files 4-5
- Scalar multiplication subroutine 19-5

- Scale factors 15-23
- Search library 19-7
- Search, binary, subroutine 19-9
- SECBLD, MIDAS subroutine 12-2
- Second color, convention 2-5
- SEG
  - command parameters, common 7-8
  - command summary 7-8
  - commands, use of pathnames in 7-8
  - error handling 7-1
  - LOAD subprocessor command summary 7-10
  - LOAD subprocessor, advanced features 11-1
  - loader defaults B-2
  - loader error messages A-6
  - loader subprocessor prompt $ 7-1
  - loading sequence 7-
  - modification subprocessor prompt $ 7-1
  - MODIFY subprocessor command summary 7-13
  - (PRIMOS command) 7-1, 8-2
  - prompt# 7-1
  - runfile, including the R-mode interlude 11-10
  - subcommand MAP 7-3
  - subcommands, important 7-2
  - (UFD) 8-5
  - utility, description 1-7
- SEG's loader 7-7
  - commands 11-2
  - execution from 7-2
- SEG, definition 2-4
  - usage 7-1
- Segment '4000,
  - '4000, splitting 11-11
  - assignment, relative 11-1
  - assignment, specific 11-3
  - assignments 7-4
  - directory, definition 2-4
  - usage 7-7
  - data 11-2
  - definition 2-4
  - link 11-2
  - loading RUNIT into 11-9
  - procedure 11-2
- Segment-spanning arrays 5-10
- Segmented
  - address code 5-9
  - loader defaults B-2
  - program techniques, extended 11-1
  - programs, loading 7-1
  - programs, restarting 8-2
  - runfiles 7-7
  - runfiles, execution of 8-2
- Segments,
  - assigned, releasing 7-7
  - decreasing number used 11-9
  - shared, incorporating files into 11-12
  - specific, loading to 11-3
  - splitting 11-9
  - stack, extension 11-6
- Segno, definition 2-4
- Selecting home spool queue 10-18
- Sequence numbers 14-2
- Sequence, load 6-2
- Sequence, loading (SEG) 7-2
- Sequential job processing environment, description 1-5
- Sequential job processor 10-11
- Sequential job processor see also CX
- Setting A register 17-6
- Setting B register 17-6
- Settng tabs 4-5
- Setting terminal characteristics 10-30
- SHARE (SEG command) 11-10
- Shared code 11-7
  - Shared code see also shared procedure
  - procedure see also shared code
  - procedure, advantages of 11-7
  - procedures, loading 11-8
  - procedures, optimizing load 11-9
  - procedures, source code 11-7
  - segments, incorporating files into 11-12
  - procedures, compiling 11-8
- SHELL subroutine 19-8
- Short and long integers, mixing 18-1
- Short call subroutines 18-1
- Short cross reference 5-8
- Short integers 14-3
- Sign extension, integer 18-1
- SIZE (PRIMOS command) 3-6
- Size, file, determining 3-6
- Skip operations, floating point, generate 5-11
- Skip operations, floating point, suppress 5-11
- SLIST (PRIMOS command) 3-6
- SORT (PRIMOS command) 10-28
- Sort characteristics 19-7
- Sort library 19-7
- Sort parameters, common 19-7
- Sorting files 10-28
- SOURCE (compiler option) 5-3, 17-5
  - code for shared procedures 11-7
  - file, compiler (unit 1) 5-3, 17-9
  - file, definition 2-4
  - language conversion 1-4
  - programs, entering 4-1
  - programs, manipulating 4-1
  - Spaces, convention 2-5
  - Spacing, using of 9-2
  - Special characters, editor 4-5
  - Special forms, printing on 3-7
  - Special terminal characters 2-5
  - Special terminal keys 2-5
  - Specific segment assignment 11-
  - Specific segments, loading to 11-
  - Specification statements 15-4
  - SPLIT (SEG's loader) 11-9
  - Splitting out 11-10
  - Splitting segment '4000 11-11
  - Splitting segments 11-9
  - SPOOL (PRIMOS command) 3-6
  - Spool
    - -CANCEL 3-7
    - -DEFER 3-7
    - file with FORTRAN print conventions 4-10
    - -FORM 3-8
    - -FTN 4-10
    - -LIST 3-6
    - -LNUM 4-10
    - printing, deferring 3-7
    - program with line numbers 4-10
    - queue, home 10-18
    - queue, listing 3-6
    - request, cancelling 3-7
- Spooling the listing file 5-5
- SRCH$$ subroutine 19-18
- ST. SIZE 7-6
- Stack 7-8
- STACK (SEG's loader) 11-6
- Stack
  - frame 7-6
  - location 7-4
  - location, RUNIT 11-9
  - segments, extension 11-6
  - size, altering 11-6
- START (PRIMOS command) 8-1
- State,
  - functions 16-2
  - functions and subroutine optimization 13-6
  - load 6-4
  - lines 14-2
  - numbers, optimization 13-3
  - sequence optimization 13-5
  - data definition 15-8
- Statements 15-1
  - assignment 15-9
  - coding 15-18
  - compilation 15-8
  - control 15-10
  - device control 15-25
  - external procedure 15-7
  - grouped, list 15-2
  - header, for subprograms 15-3
  - implemented, list 15-1
  - input 15-12
  - order of in programs 14-8
  - output 15-12
  - READ 15-14
  - run-time 15-8
  - specification 15-4
  - storage 16-6
  - WRITE 15-17
- Static allocation of local storage 5-10
- Status information, phantom 10-9
- Status, network 10-17
- STOP statement 15-12
- Storage format,
  - data, ASCII D-3
  - data, CHARACTER D-3
  - data, COMPLEX D-3
  - data, DOUBLE PRECISION D-2
  - data, INTEGER*2 D-2
  - data, INTEGER*4 D-2
  - data, LOGICAL D-2
  - data, REAL*4 D-2
- Storage statements 15-6
- Storage,
  - ANSI standard D-1
  - local, dynamic allocation 5-10
  - local, static allocation 5-10
  - symbol 6-6

Strategy, coding 9-1
Stream, output, definition 2-3
Structure of function subprogram 16-1
Structure of subroutine subprograms 16-3
Structure, program, modular 9-1
Structures, directory 2-6
Structures, file 2-6
Sub-UFD, definition 2-4
Subdirectory, definition 2-4
Subprocessor, modification 11-6
Subprogram, block data 15-3
Subprograms, function, user-defind 16-1
Subprograms, header statements for 15-3
Subroutine
  arguments 16-3
  calls 15-25
  calls, integers in 17-3
  rules 16-3
SUBROUTINE statement 15-3, 16-3
Subroutine subprogram, structure of 16-3
Subroutine,
  ATTDEV 15-14
  MIDAS, ADD1$ 12-5
  MIDAS, BILD$R 12-2
  MIDAS, DELET$ 12-5
  MIDAS, FIND$ 12-5
  MIDAS, LOCK$ 12-5
  MIDAS, NEXT$ 12-5
  MIDAS, PRIBLD 12-2
  MIDAS, SECBLD 12-2
  MIDAS, UPDAT$ 12-5
Subroutines 16-3
Subroutines
  $X versions 18-1
  application library 16-3
  conversion 1-4
  reference 19-1
  FORTRAN math 16-3
  library, loading 6-3
  library, loading (SEG) 7-3
  matrix, table 1-7
  operating system, list 19-15
  PRIMOS system 16-3
  recursive-15-7
  short call 18-1
  user-defined 16-3
Subscripts, generalized 14-5
Subscripts, maximum number of 14-5
Subtraction, matrix, subroutine 19-6
Summary,
  command, editor 4-7
  commands, FUTIL 10-25
  commands, LOAD 6-9
  commands, SEG-7-8
  commands, SEG LOAD subprocessor 7-10
  commands, SEG MODIFY subprocessor 7-13
Suppress cross reference 5-8
Suppress flagging of undeclared variables 5-11
Suppress floating point skip operations 5-11
Suppress global trace 5-9
Suppress printing of compiler error messages 5-5
SYMBOL (SEG's loader) 11-5
Symbol names, editor 4-10
Symbol storage -6
Symbol table 6-4, 7-4
Symbols 7-6
Symbols, undefined 7-7
Syntax checking, compiler 9-2
Syntax, compiler 5-1
System
  constants B-1
  defaults B-1
  information, table 3-4
  programming features, LOAD 6-8
  prompts 2-6
  resources 1-5
  interface to 12-1

T

T input format 15-22
T output format 15-20
Tab setting 4-5
Table, symbol 6-4, 7-4
Tape, 7-track, reading from 4-3
Tape, 9-track, reading from 4-3
Tape, magnetic, reading from 4-3
Tape, punched paper, reading from 4-4
Techniques, editor 4-5
Template, MIDAS, creating 12-2
Template, MIDAS, modifying 12-2
Templates, loading 11-4
TERM (PRIMOS command) 10-30
Terminal
  characteristics, setting 10-30
  character, special 2-5
  defaults B-1
  entry of programs 4-4
  keys, special 2-5
  listing of programs 4-10
Text editor 4-4
Text files, merging 10-28
TIME (PRIMOS command) 10-6
Time/date stamping of output files 10-6
TOP 7-4
TRACE
  area, statement 15-8
  (compiler option) 5-9, 17-5
  global, compiler 9-2
  global, enable 5-9
  global, suppress 5-9
  item, statement 15-8
  statements, use of 9-2
  use with COMO 15-9
Transpose, matrix, subroutine 19-6
Tree-structured file system, figure 2-9
Treename 2-7
Treename, definition 2-4
TRUE 14-4
Truncation, integer 18-1
Truth tables 14-7
TSRC$$ subroutine 19-19
Tutorial books, FORTRAN language 1-1
Type, data see also data mode
Type-ahead 2-6
Types, data 15-5

U

UFD, definition 2-4
UII handling 6-8
UII library, loading 6-8
UII see also unimplemented instruction interrupt
UII table 6-8
UNASSIGN (PRIMOS command) 4-1
Unassigning devices 4-1
Unconditional GO TO statement 15-11
Unconditional optimization 5-12
UNCOPT (compiler option) 5-12, 17-5
Undeclared variables, enable flagging 5-11
Undeclared variables, suppress flagging 5-11
Undefined symbols 7-7
Underscore, usage 2-6
Unimplemented instruction interrupt see also UII
Uninitialized COMMON, relocating 11-5
Unit, definition 2-4
Unsatisfied reference 6-6, 7-7
UPDAT$, MIDAS subroutine 12-5
Upper case convention 2-4
Usage of over 64K word COMMON 11-12
Usage, segments 7-7
Use of ATTDEV with direct access 15-13
Use of comments 9-1
Use of COMO with TRACE 15-9
Use of compiler -DYNM option 13-7
Use of direct access 15-13
Use of loadmap 7-3
Use of TRACE with COMO 15-9
User file directory, definition 2-4
User number 3-2
User, phantom, definition 2-4
User-defined function subprograms 16-1
User-defined subroutines 16-3
Using MIDAS 12-2
Using PHANTOM 10-8
Using PRIMOS 3-1
Using SEG 7-1
Using the Loader 6-1

V

V-mode FORTRAN function errors 8-3
V-mode FORTRAN library 18-1
V-mode program,
  installation in command UFD, example 8-5
  execution of 8-2
  installation in command UFD 8-5
  loading 7-1
V-mode vs. R-mode compilation 13-4
V-mode, advantages of 7-1
VAPPLB 19-9

Variable rules 14-5
Variables 14-5
Variables, formats as 15-21
Virtual loading 6-6
Volume, definition 2-4
Volume-name, definition 2-4

W

WAIT (ASSIGN option) 4-1
Word, definition 2-4
Work session, completing 3-8
Working directory, changing 3-2
WRITE statements 15-17
  binary, statement 15-17
  direct access, statements 15-17
  formatted, statement 15-17
Write/read lock table 10-27
Writing terminal output to a file 10-5

X

X input format 15-22
X output format 15-20
XREFL (compiler option) 5-8, 17-5
XREFS (compiler option) 5-8, 17-5

Z

Z (in B format) 15-23

SYMBOLS

" (usage in editor) 4-5
  (usage) 2-5
# (in B format) 15-23
  (SEG prompt) 7-1
#### (FORTRAN main program i.d.) 7-6
$ (FORTRAN address constants) 14-6
  (hexadecimal number) 2-3
  (in B format) 15-23
  (LOAD prompt) 6-1
  (SEG loader subprocessor prompt) 7-1
  (SEG modification subprocessor prompt) 7-1
$INSERT statement 15-9
  nesting not allowed 15-9
$X version, subroutines 18-1
' (octal number) 2-3
  (single quote in IBM format direct access READ) 15-15
'' (single quote representation in ASCII string) 14-4
* (in B format) 15-23
  (in MAGSAV dialogue) 10-16
  (in pathnames) 2-8
** (unsatisfied reference) 6-6, 7-7
**** (FORTRAN function error indicator) 8-2
*CMHGH 6-4
*CMLOW 6-4
*HIGH 6-4
*LOW 6-4
*PBRK 6-4
*STACK 7-4
*START 6-4, 7-4
*SYM 6-4, 7-4
*TEST 8-5
*UII 6-4
+(in B format) 15-22
, (in B format) 15-23
  (in FORMAT statement) 15-19
- (in B format) 15-23
-32R (compiler option) 5-9, 17-5
-64R (compiler option) 5-9, 17-5
-64V (compiler option) 5-9, 17-6
-BIG (compiler option) 5-10, 17-1
-BINARY (compiler option) 5-3, 17-1
-CANCEL (SPOOL option) 3-7
-DCLVAR (compiler option) 5-11, 17-1
-DBASE (compiler option) 5-10, 17-1
-DEFER (SPOOL option) 3-7
-DYNM (compiler option) 5-10, 17-1
-DYNM option, compiler, use of 13-7
-ERRLIST (compiler option) 5-6, 17-2
-ERRTTY (compiler option) 5-5, 17-2
-EXPLIST (compiler option) 5-6, 17-2
-FORM (SPOOL option) 3-8
-FP (compiler option) 5-11, 17-2
-FTN (SPOOL option) 4-10
-HOME (SPOOL option) 10-18
-INPUT (compiler option) 5-3, 17-3
-INTL (compiler option) 5-11, 17-3
-INTS (compiler option) 5-11, 17-3
-LIST (compiler option) 5-6, 17-3
-LIST (SPOOL option) 3-6
-LISTING (compiler option) 5-4, 17-4
-LNUM (SPOOL option) 4-10
-NOBIG (compiler option) 5-10, 17-4
-NODCLVAR (compiler option) 5-11, 17-4
-NOERRTTY (compiler option) 5-5, 17-4
-NOFP (compiler option) 5-11, 17-4
-NOTRACE (compiler option) 5-9, 17-4
-NOXREF (compiler option) 5-8, 17-4
-ON (LOGIN option) 10-17
-OPT (compiler option) 5-12, 17-4
-PBECB (compiler option) 5-11, 17-4
-SAVE (compiler option 5-10, 17-5
-SOURCE (compiler option) 5-3, 17-5
-TRACE (compiler option) 5-9, 17-5
-UNCOPT (compiler option) 5-12, 17-5
-WAIT (ASSIGN option) 4-1
-XREFL (compiler option) 5-8, 17-5
-XREFS (compiler option) 5-8, 17-5
. (in B format) 15-23
.AND. truth table 14-7
.FALSE. 14-4
.NOT. truth table 14-7
.NULL. 3-4
.OR. truth table 14-7
.TRUE. 14-4
/ (in FORMAT statement) 15-19
/* (comment line) 10-2
// (blank COMMON) 15-6
32R (compiler option) 5-9, 17-5
64R (compiler option) 5-9, 17-5
64V (compiler option) 5-9, 17-6
64V-mode COMMON, optimization 13-4
7-track tape, reading from 4-3
9-track tape, reading from 4-3
: (FORTRAN octal numbers) 14-3
; (usage in editor) 4-5
<*> (current disk) 2-10
> (FUTIL prompt character) 10-18
  (in pathnames) 2-7
? (usage in editor) 4-5
  (usage) 2-5
/ (usage in editor) 4-5
  (usage) 2-5
∧ (usage) 2-5
_(usage) 2-6

Technical publications requests your comments . . .

Now that you've finished reading this new final documentation release, we're very interested in hearing what you have to say about it. We'd like your comments on any facet of this document — technical content, writing style, graphics, general philosophy, as well as your suggestions for improvements and your editorial additions.

You can write a letter, make a telephone call, send a telex or make an appointment to come in. We'll guarantee that you get a personal response from the writer directly responsible for this document.

Our address and telephone number is: Prime Computer, Inc. 145 Pennsylvania Avenue, Framingham, MA 01701, telephone number (617)-879-2960, TELEX 94-8482, TWX 710-380-6567.

While you're working on detailed comments, we'd like to receive your initial reactions. The postage paid reply card will direct those reactions to the team responsible for this document.

Keep your FDR's current with our new Automatic Updating Service.

Through our unique Automatic Individual Documentation Update Service (AIDUS) we'll keep your FDR's updated for a nominal fee. You'll receive change sheet packages that correct, expand, and update your FDR's, keeping you abreast of changes and improvements in Prime products. Even if we completely rewrite an FDR, you're covered — we'll send you a copy of the new manual. AIDUS is also available to keep your Programmer's Companions updated.

To subscribe to this service, check the box at the bottom of the reply card. We'll send you an order form. If both cards have been used, call us directly.

Here are my immediate reactions to the ______________________
(*Write in document name*)

	Excellent	Good	Fair	Poor
• I thought the overall quality was	☐	☐	☐	☐
• The technical information content was	☐	☐	☐	☐
• The use of color and graphics were	☐	☐	☐	☐
• I think Prime's mixture of "how-to" with reference information is	☐	☐	☐	☐

Name______________________ Title______________________

Company______________________ Dept______________________

Address______________________

City, State, Zip______________________

Country______________________ Telephone______________________

My job function is best described as ______________________

☐ **Please send me an Automatic Update Service order form**

Here are my immediate reactions to the ______________________
(*Write in document name*)

	Excellent	Good	Fair	Poor
• I thought the overall quality was	☐	☐	☐	☐
• The technical information content was	☐	☐	☐	☐
• The use of color and graphics were	☐	☐	☐	☐
• I think Prime's mixture of "how-to" with reference information is	☐	☐	☐	☐

Name______________________ Title______________________

Company______________________ Dept______________________

Address______________________

City, State, Zip______________________

Country______________________ Telephone______________________

My job function is best described as ______________________

☐ **Please send me an Automatic Update Service order form**

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 531 FRAMINGHAM, MA

POSTAGE WILL BE PAID BY ADDRESSEE

PRIME

PRIME Computer, Incorporated
Technical Publications Department
145 Pennsylvania Avenue
Framingham, MA 01701

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 531 FRAMINGHAM, MA

POSTAGE WILL BE PAID BY ADDRESSEE

PRIME

PRIME Computer, Incorporated
Technical Publications Department
145 Pennsylvania Avenue
Framingham, MA 01701

Part One
OVERVIEW OF
PRIME'S FORTRAN

Part Two
USING FORTRAN
UNDER PRIMOS

Part Three
ADVANCED PROGRAMMING TECHNIQUES

Part Four
FORTRAN
LANGUAGE REFERENCE

Part Five
UTILITY REFERENCE

PR1ME

PR1ME Computer, Inc. 145 Pennsylvania Ave., Framingham, Mass. 01701
P/N FDR 3057-101