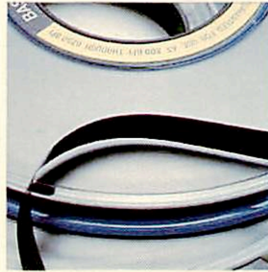
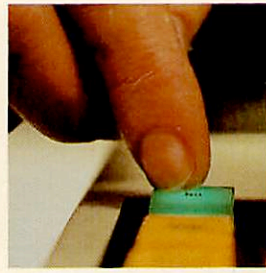
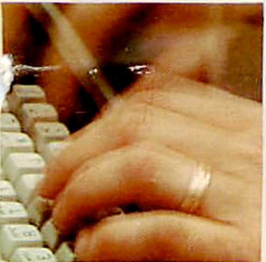
Prime Computer, Inc.
Reference Guide
FDR3057-101B

PRIME

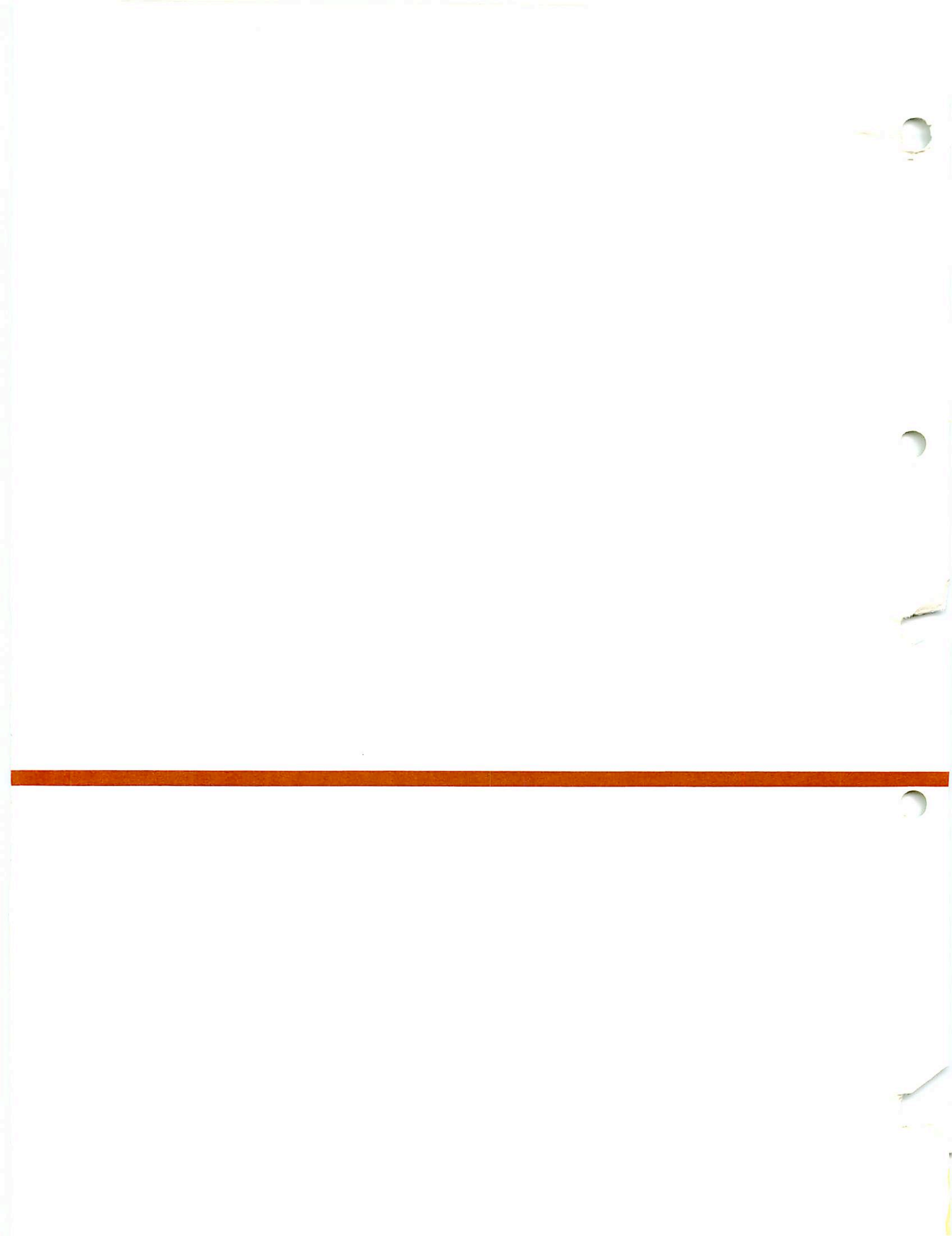
FORTRAN
Revision 17







The FORTRAN Reference Guide



FORTRAN Reference Guide

by Anthony R. Lewis

with Update Pages for Rev. 19, July 1982

by Carol A. Pryor

COPYRIGHT INFORMATION

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

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Copyright © 1982 by
Prime Computer, Incorporated
500 Old Connecticut Path
Framingham, Massachusetts 01701

PR1ME and PRIMOS are registered trademarks of Prime Computer, Inc.

PRIMENET, RINGNET, and THE PROGRAMMER'S COMPANION are trademarks of Prime Computer, Inc.

PRINTING HISTORY — FORTRAN Reference Guide

Edition	Date	Number	Documents Rev.
*First Edition	November 1977	PDR3057	14
*Update	July 1978	PTU2600-047	15
*Second Edition	January 1979	FDR3057	16.3
Third Edition	March 1980	FDR3057	17.2
Update Package 1	May 1981	COR3057-001	18.1
Update Package 2	July 1982	COR3057-002	19.0

*These editions are out of print.

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 x2053, 2054

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000, x4837

Customers Outside U.S.

Contact your local Prime subsidiary or distributor.

INFORMATION Systems

Contact your Prime INFORMATION system dealer.

CONTENTS

PART I — OVERVIEW

1 OVERVIEW OF PRIME'S FORTRAN

Introduction 1-1
FORTRAN Under PRIMOS 1-2
System Resources Supporting FORTRAN 1-5

PART II — LANGUAGE-SPECIFIC SYSTEM INFORMATION

2 COMPILING

Introduction 2-1
Using the Compiler 2-1
End of Compilation Message 2-1
Compile Error Messages 2-2
Prime FORTRAN Compiler Parameters 2-2

3 DEBUGGING

Introduction 3-1
Source Level Debugger 3-1
Coding Strategy 3-1
Compiler Usage 3-2

4 OPTIMIZATION AND OTHER HELPFUL HINTS

Introduction 4-1
DO Loops 4-1
Statement Numbers 4-2
Multi-Dimensioned Arrays 4-3
Load Sequence and Memory Allocation 4-3
Function Calls 4-4
V-Mode vs. R-Mode Compilation 4-4
64V-Mode Common 4-4
IF Statements 4-4
Input/Output 4-4
Statement Sequence 4-5
Parameter Statements 4-5
Inefficient Library Calls 4-5
Statement Functions and Subroutines 4-5
Integer Divides 4-6
Logical vs. Arithmetic IF 4-6
Use of the Compiler's-DYNM Option 4-6
Conclusion 4-7
Request for Contributions to this Section 4-7

PART III — LANGUAGE REFERENCE

5 FORTRAN LANGUAGE ELEMENTS

Legal Character Set 5-1
Line Format 5-1
Operands 5-2
Generalized Subscripts 5-5
Operators 5-6
Program Composition 5-7

6 FORTRAN STATEMENTS

Implemented Statements 6-1
Header Statements for Subprograms 6-3
Specification Statements 6-4
Storage Statements 6-5
External Procedure Statements 6-9
Compilation and Run-Time Control Statements 6-9
Assignment Statements 6-10
Control Statements 6-11
Input/Output (I/O) Statements 6-14
Coding Statements 6-19
Format Statements 6-20
Device Control Statements 6-26
Function Calls 6-26
Subroutine Calls 6-27

7 FORTRAN FUNCTION AND SUBROUTINE STRUCTURE

Functions 7-1
Subroutines 7-2

8 FORTRAN FUNCTION REFERENCE

FORTRAN Function Library 8-1

APPENDICES

A COMPILER ERROR MESSAGES

B SYSTEM DEFAULTS AND CONSTANTS

C ASCII CHARACTER SET

PRIME Usage C-1
Keyboard Unit C-1

D PRIME MEMORY FORMATS OF FORTRAN DATA TYPES

Introduction D-1
Data Types D-2

I

OVERVIEW

1
Overview of Prime's FORTRAN

INTRODUCTION

This document contains the information specific to Prime's FORTRAN IV language and its compiler (FTN). General program development information: getting on the system, entering programs, loading, and execution are treated in the Prime User's Guide. We assume that you have read the Prime User's Guide and are familiar with the FORTRAN language, but not necessarily with its implementation 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 book documents Prime's FORTRAN IV and its compiler (FTN) at software revision level 17 (Rev. 17). Together with our new book, the Prime User's Guide, it replaces the FORTRAN Programmer's Guide, FDR3057. The language-specific material in the Programmer's Guide has been restructured in this language reference guide, while language-independent material (on PRIMOS utilities and commands) has been expanded and placed in the Prime User's Guide. (Details on the use of subroutines remain in the Subroutine Reference Guide).

18

This restructuring represents another stage in the continuing evolution of Prime's documents. First, it reflects the growing number of Prime's compiled languages (FORTRAN IV, FORTRAN 77, COBOL, PL/I subset G, PASCAL and RPG II, as of Rev. 18). Second, it points up the compatibility, at object code level, of program modules written in these languages. For example, a FORTRAN subroutine can be called from an RPG program module, or a PL/I subset G subroutine from a COBOL program. Third, it recognizes that program development is basically identical in all high-level languages (with a few exceptions, such as loading libraries). Thus, applications programmers can use the Prime User's Guide as their tutorial for PRIMOS, and the language reference guides, such as this book, as reference works.

Organization

The guide is composed of three parts:

Part I. 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 this information (Section 1).

Part II. Language-specific system information. This part of the book includes complete details on the use of the FORTRAN IV compiler, FTN (Section 2). Suggestions to the programmer for debugging (Sec-

tion 3) and optimization (Section 4) are presented along with the locations of additional information.

Part III. **FORTRAN language reference.** Sections 5-8 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.

Appendices A complete list of compiler 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

Prime User's Guide
PRIMOS Commands Reference Guide
PRIMOS Subroutines Reference Guide

Software Subsystem Reference

The FORTRAN Programmer's Companion
The New User's Guide to EDITOR and RUNOFF
LOAD and SEG Reference Guide
Source Level Debugger Guide
MIDAS User's Guide
Reference Guide for DBMS Schema DDL
FORTRAN Reference Guide for DBMS
FORMS Programmer's Guide

18

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

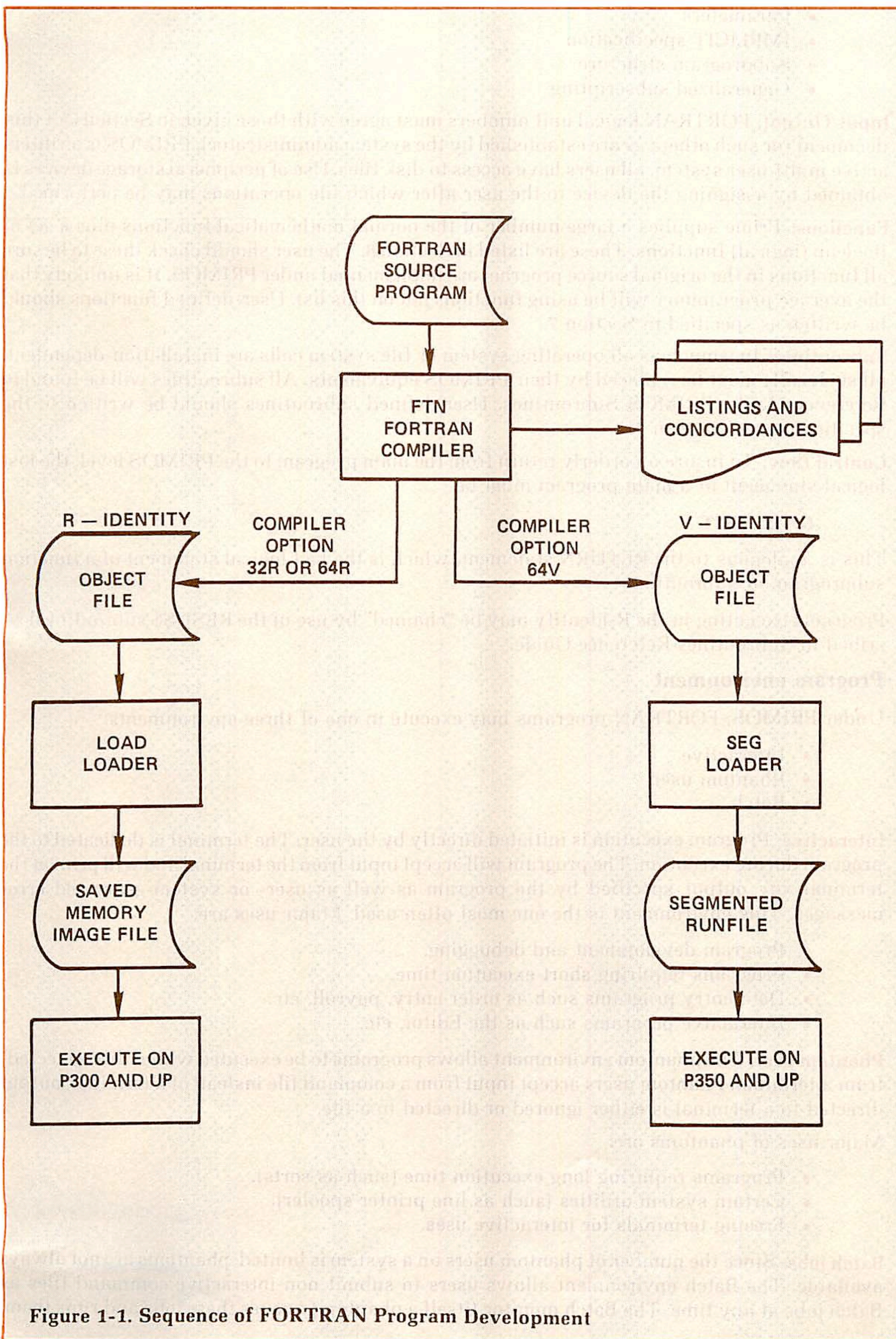


Figure 1-1. Sequence of FORTRAN Program Development

- Parameters
- IMPLICIT specification
- Subprogram structure
- Generalized subscripting

Input/Output: FORTRAN logical unit numbers must agree with those given in Section 6 of this document (or such others as are established by the system administrator). PRIMOS is an interactive multi-user system, all users have access to disk files. Use of peripheral storage devices is obtained by assigning the device to the user 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 8. 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 7.

Subroutines: Inasmuch as all operating system or file system calls are installation-dependent, all such calls must be replaced by their PRIMOS equivalents. All subroutines will be found in Reference Guide, PRIMOS Subroutines. User-defined subroutines should be written to the specifications in Section 7.

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-identify may be "chained" by use of the RESU$$ subroutine described in Subroutines Reference Guide.

Program environment

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

- Interactive
- Phantom user
- Batch

Interactive: Program execution is initiated directly by the user. 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 allows programs to be executed while "disconnected" from a terminal. Phantom users accept input from a command file instead of a terminal; output directed to a terminal is either ignored or directed to a file.

Major uses of phantoms are:

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

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

one to six at a time, as phantoms become free. (See the Prime User's Guide for details).

File system summary

PRIMOS allows the user to access up to 128 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. Fileunits 1-16 opened by these subroutines, may be accessed by FORTRAN I/O statements such as READ, WRITE, ENCODE, DECODE. See Section 6 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

Operation	Data Mode of Argument and Value Integer	Single-Precision	Double-Precision	Complex
Sine	*n/a*	SIN	DSIN	CSIN
Cosine	*n/a*	COS	DCOS	CCOS
Arctangent	*n/a*	ATAN	DATAN	
Arctangent of ratio	*n/a*	ATAN2	DATAN2	
Hyperbolic tangent	*n/a*	TANH		
Log-base e (Ln)	*n/a*	ALOG	DLOG	CLOG
Log-base 2	*n/a*		DLOG2	
Log-base 10	*n/a*	ALOG10	DLOG10	
Exponential	*n/a*	EXP	DEXP	CEXP
Square root	*n/a*	SQRT	DSQRT	CSQRT
Absolute value	IABS	ABS	DABS	CABS
Remainder (modulus)	MOD	AMOD	DMOD	*n/a*
Truncation to Integral value	*n/a*	AINT	DINT	*n/a*
Positive difference	IDIM	DIM		*n/a*
Magnitude of first no. times sign of second	ISIGN	SIGN	DSIGN	*n/a*
Complex conjugate	*n/a*	*n/a*	*n/a*	CONJG
Maximum of List	AMAX0 *(1)*	AMAX1	DMAX1	*n/a*
	MAX0	MAX1 *(1)*		*n/a*
Minimum of List	AMIN0 *(1)*	AMIN1	DMIN1	*n/a*
	MIN0	MIN1 *(2)*		*n/a*

Notes

n/a — Not applicable.

1 — Value mode is single-precision.

2 — Value mode in integer.

Libraries

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).

Table 1-2. Matrix Operations Subroutines

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

Notes

n/a — Not applicable

* — For square matrices only

Compiler

Prime's FORTRAN IV compiler operates on FORTRAN source code to generate highly optimized object code. The user has the option, at compilation time, of generating object code for execution in either the R-identity or V-identity. Additional options control I/O specifications, listings, concordances, memory usage, and other useful operations. The compiler is described in Section 2.

Loader

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

SEG utility

SEG is the V-identity program loading and execution utility. It combines separately compiled program modules, subroutines, and libraries into an executable program. (see the Prime User's Guide). Program modules can be up to 64K words long. All memory management, symbol

tables, linkages, etc., are handled by SEG's loader. Various types of loadmaps may be obtained. The SEG utility has many functions, including loading shared procedures. These are described in LOAD and SEG Reference Guide.

Editor

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

Multiple index direct access system (MIDAS)

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).

18 | Complete documentation is MIDAS User's Guide.

Database Management system (DBMS)

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 Mangement 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.

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. 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 PL/I subset G programs; PL/I subset G subroutines may be called from FORTRAN programs; and FORTRAN subroutines may be called from COBOL programs. Details are in the PMA Programmer's Guide, The COBOL Reference Guide and the PL/I subset G Reference Guide.

II

LANGUAGE-SPECIFIC SYSTEM INFORMATION

2
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), V-identity code is loaded with Prime's segmented-addressing utility (SEG). These loaders are described in the Prime User's Guide and in the LOAD and SEG Reference Guide. Segmented-addressing code can be executed on Prime 50 Series computers. In the V-identity, special code can be generated for use by the source level debugger (DBG) described in the Source Level Debugger Guide.

18

USING THE COMPILER

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

FTN pathname [-options]

18

pathname The pathname of the FORTRAN source program file.

options The mnemonics for the options controlling compiler functions such as I/O device specification, listings, and others.

All mnemonic options must be preceded by a dash "-". The name of the source program file must be specified as the first expression following FTN. For example,

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

18

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

END OF COMPILATION MESSAGE

18

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 option 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] w WARNINGS

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.

w — The number of warnings. If only 1 warning, the final S is not printed. w WARNINGS does not appear if there are no warning messages.

Example:

```
0000 ERRORS [<.MAIN.>FTN-REV18.1]
```

indicates the successful compilation of a main FORTRAN program by the 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. If the message is a warning, the word WARNING will be part of the message. A list of all messages is given in Appendix A. An ERROR message means the program did not compile; a WARNING message means the program did compile, but not necessarily the way you thought it would.

Example:

****** LINE 0010 [WRUT] UNRECOGNIZED STATEMENT**

Note that the name field has been omitted.

PRIME FORTRAN COMPILER OPTIONS

All options are preceded by a dash, "-", in the command line. Options that are the PRIME-supplied default options (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 2-1).

▶ 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 option list. The code generated here will work whether or not the array actually spans a segment boundary. See also **NOBIG, 64V.** See Section 6 for more information on this requirement.

▶ BINARY { pathname | YES | NO }

Specifies the binary (object) output file. If **pathname** is given, then that will be the name of the

18

binary file. If YES is used, the name of the binary file will be PROGRAM.BIN (where PROGRAM.FTN 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 2-1.)

DCLVAR

Flags undeclared variable. If included in the option list, the compiler will generate an error message when a variable is used in the program, but not included in a header, specification, storage, or external 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).

DEBUG

Produces code allowing full use of the source level debugger (DBG). Modules compiled with this option accept statement breakpoints from the debugger; the debugger recognizes their statement numbers and source line numbers. The code so generated is slower and more space-consuming; interstatement optimization is turned off. For 64V mode only. See **NODEBUG, PROD.**

DYNM

Enables local storage in Stack Frame (Prime 50 Series). Allows dynamic allocation of local storage and also sets the compiler to generate 64V mode object code. The DYNM option allows better memory utilization in the 64V mode. It also allows the creation of recursive FORTRAN subroutines (subroutines which call themselves). See **SAVE, 64V.**

18

18

Table 2-1. Compiler File Specifications

Compiler Mnemonics	INPUT or SOURCE	LISTING	BINARY
pathname	First looks for file named **pathname** .FTN; if not found then 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. PROGRAM.LIST	uses default file-name for binary file. PROGRAM.BIN
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 2-2 for A- and B-register settings.

ERRLIST

Prints only error messages in the listing file. See **EXPLIST**, **LIST**.

Note

This option 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 option 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.

FRN

18

Causes all single-precision numbers (REAL*4) to be rounded each time they are moved from a register to main storage. The methods of rounding is: if the last mantissa bit is 1, add a 1 to the second-to-last bit and set the last bit to 0. This rounding reduces loss of accuracy in low-order bits when many calculations are performed on the same number. This slightly increases execution time and should be used only if maximum accuracy is a major consideration. This option has no effect on double-precision numbers. See **NOFRN**.

INPUT pathname

Specifies the **pathname** of the input source program (See Table 2-1). This option *must not* be used if the source filename immediately follows the FTN command; otherwise, it *must* be included in the option 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 option 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 option has no effect unless an output device/file is specified using LISTING.

▶ **LISTING** { **pathname** | **YES** | **NO** | **TTY** | **SPOOL** }

18

Specifies the listing device/filename:

pathname	Opens this file for the listing.
YES	Uses the default name for the listing file PROGRAM.LIST (where PROGRAM.FTN 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 option is omitted from the option list, it is equivalent to the -LISTING NO parameter inclusion (i.e., no listing file is created).

▶ **NOBIG** *Default*

Utilizes relative addressing. This is the usual memory addressing mode. See **BIG**.

▶ **NODCLVAR** *Default*

Suppresses undeclared variable flagging. Does not generate error messages when undeclared variables are detected. See **DCLVAR**.

NODEBUG *Default*

Produces code without debugger information. This is the mode to be used for a completely debugged and tested program. See **DEBUG**, **PROD**.

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 option 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**.

NOFRN *Default*

Does not cause rounding of single-precision numbers. See **FRN**.

18

NOTRACE *Default*

Suppresses global trace. Does not enable the global trace. Does not override **TRACE** statement. See **TRACE**.

NOXREF *Default*

Suppresses concordance. Do not generate any concordance (cross-reference) listing. See **XREFL**, **XREFS**.

OPT

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 **STDOPT**, **UNCOPT**.

PBECB

Generates code to load Entry Control Blocks (ECBs) into procedure frame. For 64V-mode subroutines only. See **64V**.

PROD

Generates code allowing partial use of the source level debugger. Breakpoints can be set at procedure entries and exits, not at individual statements. Variables are as accessible as in DEBUG mode. Code is as optimized as the NODEBUG compiler code. However, storage of extra information increases the size of the object file and thus the size of the runfile. For 64V mode only. PROD may be used with OPT or UNCOPT. See **DEBUG**, **NODEBUG**.

SAVE *Default*

Local storage allocation. Performs local storage allocation statically. See **DYNM**.

SOURCE

Same as **INPUT**. See **INPUT**.

STDOPT *Default*

Generates code which does not optimize DO loops. See **OPT**, **UNCOPT**.

TRACE

Enable global trace. When this option 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

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**, **STDOPT**.

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.

An example of the concordance is:

```
OK, FTN POOH -L TTY -NOERRTTY -XREFS
 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)         WRITE (1,110) X
(0011)  110    FORMAT (I5)
(0012)         CALL EXIT
(0013)         END

  A      R            000062   0006M  0009
  B      R            000064   0002M  0006
  C      R            000066   0003M  0006
  EXIT   R EXTERNAL   000000   0012
  I      I            000070   0002   0003   0004M  0005   0007M
  X      R            000071   0001M  0009   0010
  Y      R            000073   0009M
```

```
$100                000041   0005    0009D
$110                000056   0010    0011D
$20                 000022   0005D   0008
$310                000001   0001D   0005
$320                000030   0005    0006D

0000 ERRORS [<.MAIN.>FTN-REV18.1]
0000 ERRORS [<.MAIN.>FTN-REV18.1]
```

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 2-2.

Table 2-2. Concordance Codes

Code	
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.

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 accommodate 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 the Prime 50 Series 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 No debugger code No DO loop optimization

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 options are given in Table 2-3.

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 2-4.

Default	A Register Bit		Reset (0)	Set (1)
0	0	1		
0	0	2		EXPLIST
	0	3		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		
	1	15		BINARY
	1	16		
	B Register Bit			
0	0	1		
	0	2		PBECB
0	0	3	SAVE	DYNM
	0	4	PROD, NODEBUG	DEBUG
	0	5	STDOPT	OPT, UNCOPT
0	0	6	OPT, STDOPT	UNCOPT
	0	7	NODEBUG	DEBUG, PROD
	0	8	NOBIG, 32R	BIG, DYNM, 64V
0	0	9	NOBIG	BIG
	0	10	INTS	INTL
	0	11		
0	0	12	NOXREF	XREFS
	0	13	NOXREF	XREFL, XREFS
	0	14	NOFRN	FRN
0	0	15	FP	NOFP
	0	16	NODCLVAR	DCLVAR

Figure 2-1. Bit-Mnemonic Correspondence (A and B Registers)

18

Table 2-3 A- and B-register Bit Correspondences of Options (PRIME-supplied defaults are indicated)

A(x,y) = 0(or 1): the mnemonic option 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.

Option	Bit correspondence
BIG	B(8,9) = 1
BINARY	A(14,15,16) = object file definition (See Table 2-4); PRIMOS BINARY command
DCLVAR	B(16) = 1
DEBASE	A(6) = 1
DEBUG	B(4,7) = 1
DYNM	B(3,8) = 1
ERRLIST	A(3) = 1
ERRTTY	A(7) = 1; *default*
EXPLIST	A(2) = 1
FP	B(15) = 0; *default*
FRN	B(14) = 1
INPUT	A(8,9,10) = input file definition (See Table 2-4)
INTL	B(10) = 1
INTS	B(10) = 0; *default*
LISTING	A(11,12,13) = listing file definition (see Table 2-4); PRIMOS LISTING command
NOBIG	B(8,9) = 0; *default*
NODCLVAR	B(16) = 0
NODEBUG	B(4,7) = 0; *default*
NOERRTTY	A(7) = 0
NOFP	B(15) = 1
NOFRN	B(14) = 0; *default*
NOTRACE	A(4) = 0; *default*
NOXREF	B(12,13) = 0; *default*
OPT	B(5) = 1; B(6) = 0
PBECB	B(2) = 1
PROD	B(4) = 0; B(7) = 1
SAVE	B(3) = 0; *default*
SOURCE	A(8,9,10) = input file definition (see Table 2-4); same as INPUT.
STDOPT	B(5,6) = 0; *default*
TRACE	A(4) = 1
UNCOPT	B(5,6) = 1
XREFL	B(13) = 1
XREFS	B(12,13) = 1
32R	A(5) = B(8) = 0; *default*
64R	A(5) = 1
64V	B(8) = 1

18 | 18

Table 2-4. 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

File unit usage

Three file units may be active during a compilation:

File Type	PRIMOS file unit
Source	1
Listing	2
Object	3

If the disk is specified as the device for the listing and/or object file, FTN causes these files to be opened on the disk with default names constructed as follows:

If the source file has the pathname

[MFD]>UFD1>. . .>filename

then the listing file and the object file will be opened as L_filename and B_filename respectively in the UFD currently attached to. Upon completion of the FTN command all files are closed and command returns to PRIMOS.

If the user desires the listing or binary files to be opened in UFDs other than the current one, this must be done prior to invoking the FTN command.

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:

CLOSE $\left\{ \begin{matrix} \textbf{2} \\ \textbf{pathname-2} \end{matrix} \right\} \left\{ \begin{matrix} \textbf{3} \\ \textbf{pathname-3} \end{matrix} \right\}$
or
CLOSE ALL

3
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).

SOURCE LEVEL DEBUGGER

Prime has available, as a separately-priced software package, an interactive source level debugger (DBG), which interfaces with FORTRAN IV program modules. Major features of this debugger enable you to:

- Set both absolute and conditional breakpoints
- Request the execution of debugger commands (action list) when a breakpoint occurs
- Execute the program step by step
- Call subroutines or functions from debugger command level
- Trace statement execution
- Trace selected variables, printing a message when their value changes
- Print and change variable values
- Print a procedure call/return stack history (traceback)
- Examine the source file while executing within the debugger, eliminating the need for hard-copy listings

If you have not purchased the source level debugger, other debugging aids and techniques are available. They are discussed below.

CODING STRATEGY

Coding strategy involves avoiding traditional errors in order to minimize the need for debugging. (Section 4 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, spacing, and blank lines, 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 6. 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 header, specification, storage, or external statements. This procedure often uncovers minor spelling errors in the source file (e.g., you declared the variable TEMP, but elsewhere typed it as TMEP).

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 the Prime User's Guide for COMOUTPUT information).

See Section 2 for more information on these compiler options.

4
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 1Ø I = 1, 5Ø
         A = 3.Ø1
            .
            .
   1Ø CONTINUE
```

should be changed to:

```
      A = 3.Ø1

      DO 1Ø I = 1, 5Ø
            .
            .
   1Ø CONTINUE
```

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

```
      SUM = Ø

      DO 1Ø I = 1, 9Ø
         SUM = SUM + ARRAY (I)
   1Ø CONTINUE

      ARRAY(N) = ARRAY(N) + SUM

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

      DO 1Ø I = 1, 9Ø
         ARRAY(N) = ARRAY(N) + ARRAY (I)
   1Ø 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 2Ø I = 1, 5
          DO 1Ø J = 1, 1ØØ
```

```
                  .
                  .
                  .
              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 x (TS + 100Ti + 100Tb)
2. Time(2) = 100 x (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-multiples = number-of-dimensions - 1. For instance, the example above could be written as:

```
      DIMENSION TARRAY  (1)
      EQUIVALENCE (ARRAY(1,1), TARRAY(1))

      DO 10 I = 1, 10000
         TARRAY(I) = 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 the LOAD and SEG Reference Guide 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 (see the LOAD and SEG Reference Guide) 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 effficiently, 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 /Ø/

      IF(COUNT .NE. 12) GOTO 1

      CALL TONL
      COUNT = Ø

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

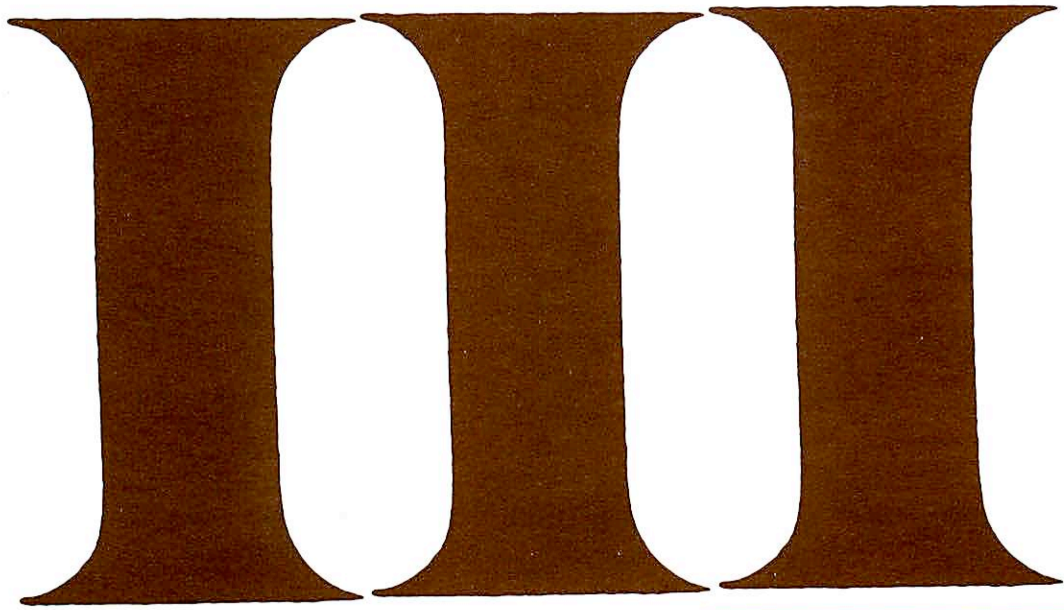
These are some of the more common guidelines to keep in mind while programming in Prime FORTRAN. If you keep these ideas in mind while writing, or while 'tweaking' FORTRAN programs, your programs will be generally smaller and faster. Some of these rules are not necessarily permanent. As Prime FORTRAN 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., 500 Old Connecticut Path, Framingham, MA 01701.



LANGUAGE REFERENCE

5
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 5-1.)

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

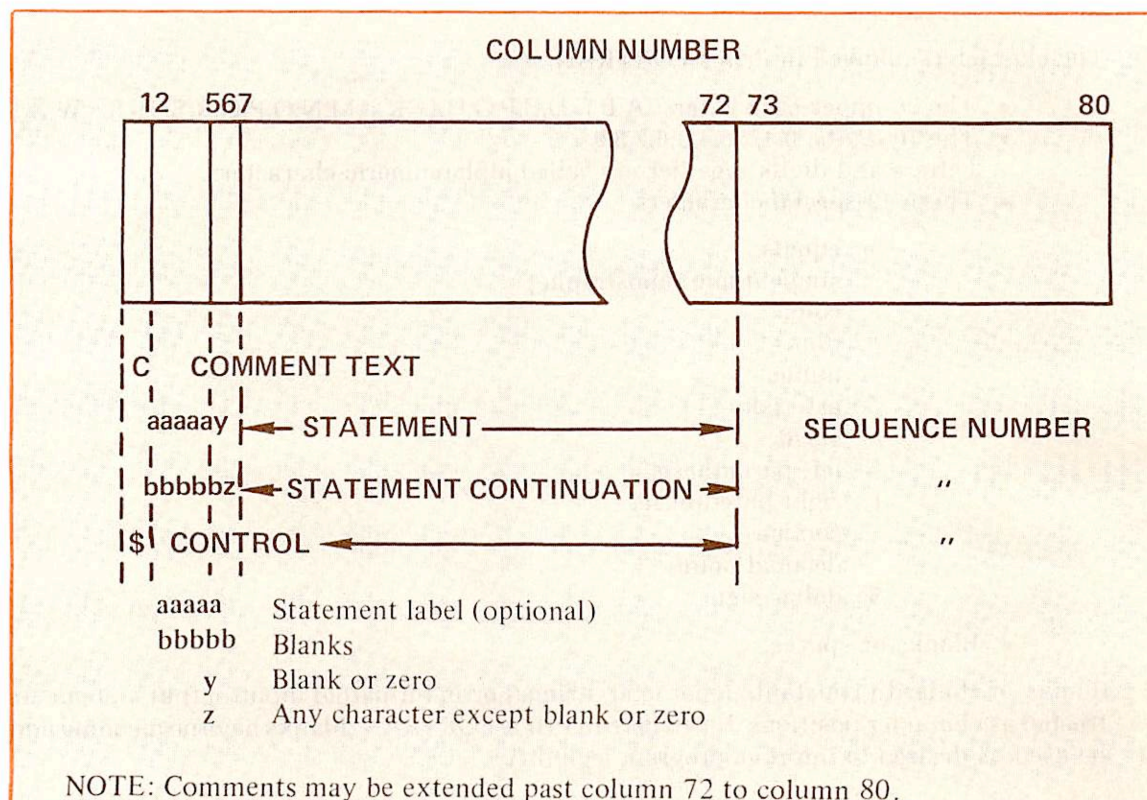
Columns 1-5 are reserved for the numerical statement label, if any. (Blanks and leading zeros are ignored.) Column 6 must be a blank or a zero. Columns 7-72 contain the statement. The statement may begin with leading blanks; this is often done to make the program easier to read, as for indention of nested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

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.



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

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 :077777 (octal). The number -32768 is a long integer in its decimal representation, but a short integer in its octal representation.

Long integers may have any number of digits (plus a sign) - only the magnitude is restricted, for example, 000000000000000000001.

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). The number -2147483648 can be represented in octal but *not* in decimal form.

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 Section 2 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 8 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. The D format is *mandatory* for double-precision numbers. For example:

12345.9253 D-11

The exponent (following D) may take on values from -9902 to +9825. Only 3 digits can be printed from the exponents (see FORMATS, Section 6).

19

18

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 6).

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 6, 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. An array may have from 1 to 7 subscripts.

The total size of all arrays not in COMMON may not exceed one segment (128K bytes). If arrays are larger than one segment, they must be put into COMMON blocks.

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(1ØØ,1ØØ),B(1Ø),Z
      INTEGER G(3,4,5),H(3ØØØ),I,J,K
C
C     ASSIGNMENT
C
      Z=A(G(H(25**K**2),2,RS(I,H(2))),INTS(Z-A(1,1Ø*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 parenthese 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 5-2. 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 6.

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 5-2. Source Program Composition

6

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: COMPLEX, 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

In Prime's FORTRAN, subroutine arguments are passed by address (location) rather than by value. Thus, it is extremely important not to place constants or parameters in the argument list as arguments which will be returned, since this will alter their value. Also, returned arguments may not be expressions.

18

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 2.

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 2 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.

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 is available for assignments 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 related 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.

18

Restrictions: There are a number of restrictions on over 64K COMMON blocks and arrays spanning segment boundary. 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 the starting address 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

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 parameter (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.

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 on 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 (3) and the Compiler Section (2).

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 statement, unless both refer to the same statement label.

TRACE is overridden by the compiler global trace option -TRACE, but not by the -NOTRACE option (see Section 2). 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.

18

COMO filename

where filename is the file into which terminal output is to be written. After the program has halted, output to the file is stopped and the file closed by:

COMO -END

The form of the command given here does not turn off output to the terminal. A complete description of this command is given in the Prime User's Guide.

$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 and another with the following restrictions:

1. Logical data should not be combined with any other mode.
2. No operator can combine Double Precision 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 6-1. 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** is a statement label and **i** is an integer variable. 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.

END

Unconditional GO TO statement

GO TO k

Transfers control to statement labelled **k**.

Computed GO TO statement

GO TO (k1, k2, . . .,kn), i

Transfers control to statement labelled **kj** when integer expression **i** = **j**. If the value of i lies outside the range 1 to n, then control passes to the next executable statement after the computed GO TO.

Assigned GO TO statement

GO TO i[,(k1, k2 . . .,kn)]

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.

18

Note

An arithmetic IF may be the statement in a logical IF but this is *not* recommended as a good programming practice. If the statement is an assignment statement, then the = must be on the first line — not on a continuation line.

Table 6-1. Data Modes 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.
Fix: Truncate fractional part and transform integral part to integer.
Truncate: Take 16 low-order bits and store in short integer datum.
Sign-Extend: Pad 16 high-order bits with 0's or 1's if short integer is positive or negative, respectively.

18

PAUSE statement

PAUSE [n]

Where **n** is an optional decimal number 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 6-2 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 of 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.

18

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 6-2).

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 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. The buffer size for the shared libraries is 6K words.

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 6-2. 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
29-139	Funit 17-127
140	Printer unit 0
141	Printer unit 1

18 19

Formatted READ statement

READ (u, f, [, END = a] [, ERR = b]) list

Causes data on FORTRAN unit **u** to be read into the variables/array names specification according to the format of statement **f**. If no list is given, one record is read and ignored.

CAUTION

Hollerith formats should be avoided in FORMAT statements associated with READ statements. The A format should be used for strings.

Direct-access READ statements

READ (u'r,f,ERR=b) list *IBM format*

READ (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 READ and 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 write 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 on 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 statement

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).

18 The END= specifier is not allowed in the direct access writestatement. 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), INDEX2 (3, INDEX1), INDEX1 = 1, 5)
```

or

18
```
((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 parenthesis 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 6-3 and 6-4 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 replaced, **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 6-3 and 6-4 (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 ten 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 6-3. Results of Formats in Output Statements

FORMAT	OUTPUT
snFw.d Floating	Prints Real or Double Precision Numbers as mixed output (no exponent) with as many significant figures as the data type allows. **w** is the total field width and must allow one position for a decimal point and one for a minus sign (if negative numbers are to be printed). **d** is the number of decimal places (right of decimal point). Numbers are right justified. Leading zeroes are inserted for numbers less than 1; trailing zeroes are used to fill the decimal places if necessary. Only minus signs are printed. If total field width is too small, the number is truncated and a $ printed if positive, a = if negative. If the decimal section is too small, the number is rounded.
snEw.d Exponential	Prints Real or Double Precision numbers as a number with a magnitude between 0.1 and 0.9999999 times an exponent. The field width **w** must allow for a minus sign (if one is to be printed), a decimal point, and three or four positions for the exponent representation (see below). The number **d** sets the number of places to the right of the decimal point — the maximum is seven. The representation with magnitude less than 1 may be overridden using scale factors.
snGw.d General	Prints Real or Double Precision numbers in F or E format according to the magnitude of the number and the decimal place specifier - **d**. Truncation is performed as for E and F formats.

Exponent representations for snEw.d:

	Exponent Value	Exponent Representation		Width
wxyz=	-9999 to -1000	=wxy	(fourth digit lost)	4
xyz=	-999 to -100	-xyz	(no E)	4
yz=	-99 to -10	E-yz		4
z=	-9 to 9	E-z or E z		3
yz=	10 to 99	E yz		4
xyz=	100 to 999	+xyz	(no E)	4
wxyz=	1000 to 9999	$wxy	(fourth digit lost)	4

Effective formats for snGw.d:

Magnitude	Effective Format
0.1 to 1.0	F(w-4) .d,4X
1.0 to 10.0	F(w-4).(d-1), 4X
.	.
.	.
.	.
10**(d-2) to 10**(d-1)	F(w-4) .1, 4X
10**(d-1) to 10**d	F(w-4) .0, 4X
Outside Range	Ew.d

Format	Description
snDw.d Double Precision	Prints Double-Precision Numbers only in an exponential format similar to the E format except that the letter D is used instead of E and that **d** has a maximum value of 14.
wX Space	Writes **w** spaces into the output record (negative w backspaces for replacing).*
Tw Tab	Positions output pointer to column **w** in the output record. Back tabbing is permitted. Example: (T1,40A2,T15,F9.3)
wHc1c2 . . .cw Hollerith	Prints the string **c1c2 . . . cw.*** 1. Does not require an item in the output list 2. Need not be followed by a delimiter.
nAw ASCII	Prints Integer, Real, Complex, or Double Precision variables as ASCII characters. **w** is number of characters per variable or array name. Output is right 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 ouput 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 accommodate 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 within. 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 6-4. 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).
nIw Integer	Stores external numbers in integers. If no sign is present, a plus sign is assumed. A sign or blank is counted as one character position. No decimal points are allowed. If there are more numbers than the field width, w, only the left-most w characters are stored.
B'string' Business	NOT USED*

**No repeat count is allowed 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 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 at 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 6-5 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 6-3 and 6-4)

Where n, the scale factor, is an integer constant with an optional minus sign. Once a scale factor has been specified, it applies to all subsequent F,E,G, and D field descriptors, until another scale factor is encountered. If n=0, an existing scale factor is removed. The scale factor has no effect on type I,A,H,X,L, or B descriptors.

E and D output scale factor: Before output conversion, the fractional part of the internal number is multipled 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 6-5. Examples of B-Format Usage

Number	Format	Output Field
123	B'####'	Ø123
12345	B'####'	****
Ø	B'####'	ØØØØ
123	B'ZZZZ'	123
1234	B'ZZZZ'	1234
Ø	B'ZZZZ'	
Ø	B'ZZZ#'	Ø
1.Ø35	B'#.##'	1.Ø4
Ø	B'#.##'	Ø.ØØ
1234.56	B'ZZZ,ZZZ,ZZ#.##'	1,234.56
123456.78	B'ZZZ,ZZZ,ZZ#.##'	123,456.78
Ø	B'ZZZ,ZZZ,ZZ#.##'	Ø.ØØ
2	B'+###'	+ØØ2
-2	B'+###'	-ØØ2
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.ØØ
-123	B+++,++#.##'	-123.ØØ
98	B'$ZZZZZZ#'	$ 98
98	B'$$$$$$#'	$98
156789	B'$***,***,**#.##'	$****156,789.ØØ

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)
-	Three Lines
Other	One line

In all cases the control character is not printed.

DEVICE CONTROL STATEMENTS

For physical positioning of sequential access devices.

BACKSPACE statement

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 also supports disk files.

18

ENDFILE statement

ENDFILE u

Writes an end-of-file mark 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.

7
FORTRAN function and subroutine structure

FUNCTIONS

There are four types of functions; all are called in the same manner (see Section 6).

Prime FORTRAN library functions

These library subprograms (see PRIMOS Subroutine Reference Guide and Section 8) 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 calculations, using the supplied arguments as values.)

.

.

.

name = Final calculation

.

.

.

RETURN

FUNCTION statement: The FUNCTION statement, which must be the first statement of a FUNCTION subprogram, assigns the name of the function and identifies the dummy arguments. In the preceding example, **name** is a symbolic name assigned to identify the function, and each **argument** is a dummy argument. There is no syntactical limit to the number of arguments. The function name must conform to the normal rules for all symbolic names with regard to number of characters, etc. Implicit result mode typing occurs according to the first letter of the name. Implicit mode typing can be overridden by preceding the word FUNCTION with one of the mode specifications. The function name must differ from any variables used in the function subprogram or in any main program which references the function.

Body of subprogram: The body of the function subprogram can consist of any legal FORTRAN statements except SUBROUTINE, BLOCK DATA, or other FUNCTION statements. The statements that evaluate the function use constants, parameters, variables, and expressions in the normal way. The program must produce a single result for a given set of argument values. The subprogram must equate the assigned symbolic function name to the result, by using name on the left side of an assignment statement. It is the function name itself, used as a variable, that returns the result to the main program.

RETURN statement: The RETURN statement consists of a single word RETURN. It terminates the subprogram and returns control to the main program. The RETURN statement must be the last statement in the subprogram (logically, not physically; that is, it must be the last statement to which control passes).

Statement functions

Statement functions are embedded in the coding of the main program and are compiled as part of the main program. Any calculation that can be expressed in a single statement, and produces a single result, may be assigned a function name and referenced in the same way as a library function. A statement function is defined in the form:

name (argument-1, argument-2,. . .argument-n) = expression

where **name** is the symbolic name assigned to the function and each **argument** is a dummy variable that represents one of the arguments.

The following rules apply to all functions:

1. The **name** may consist of one to six alphanumeric characters, the first of which is alphabetic. It must differ from all other function names and variable names used in the main program.
2. The **argument** list follows the name and is enclosed in parentheses. There must be at least one argument. Multiple arguments are separated by commas. Each argument must be a single nonsubscripted variable. These arguments are only dummy variables, so their names may be the same as names appearing elsewhere in the program. The dummy variable names do indicate argument mode, however, by implicit or explicit mode typing. There is no syntactical limit to the number of arguments.
3. During each call of a function, the values 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 definition 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 PRIMOS Subroutines Reference Guide).

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 (PRIMOS Subroutines Reference Guide).

FORTRAN math subroutines

These handle mathematical calculations such as matrix multiply and inversion, permutations, etc. (See PRIMOS Subroutines Reference Guide).

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.

8
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-identify. 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.

Trigonometric functions

The arguments of the trigonometric functions COS, CCOS, DCOS, SIN, CSIN, and DSIN are in radians, not in degrees.

The IMPLICIT statement and FORTRAN intrinsic functions

Changing FORTRAN's typing conventions with the IMPLICIT statement has no effect on the intrinsic functions. However, the random number generators, RND and IRND, are not intrinsic functions. If the IMPLICIT statement changes the default typing of I or R and the random number functions are used in the program, then these functions must be typed explicitly as REAL*4 (for RND) and INTEGER*2 (for IRND).

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 wil also occur. This is not considered to be good programming practice and is discouraged. (See Assignment Statements in Section 6).

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)))
```

The INTS and INTL functions will take as arguments short integers (INTEGER*2), long integers (INTEGER*4), single-precision floating-point numbers (REAL*4), and double-precision floating-point numbers (REAL*8) and return either a short (INTS) or a long (INTL) integer.

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.

19

The INT, IDINT, IFIX, MAX1 and MIN1 functions: The results of these functions will be the default INTEGER type for the module. That is, if compilation uses the -INTS (default) option, then the mode of INT, IDINT, IFIX, MAX1 and MIN1 will be INTEGER*2. If compilation is performed with the -INTL option, then their mode will be INTEGER*4.

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 maximum 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 expontial of a double-precision floating-point number.
DP = DEXP

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 = DLOG 10 (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	$0 \leq I2 \leq 32767$
<0	Initializes the random number generator and returns the first random number	$0 \leq I2 \leq 32767$

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.

	constant
	variable name
I = LOC	**array name**
	array element

Note

In the 64V mode, LOC may be passed as an argument in functions or subroutines, e.g., I = AND(LOC(A),LOC(B)). In this mode, LOC returns a two-word value: the first word represents the segment number; the second is the word number in the segment.

LS

Shifts an integer variable left by a specified number of bits; vacated bits are filled with zeroes.
I2 = LS (I1, IP)
where IP is the number of bits to be shifted to the left. If IP $\leqslant 0$, 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$\leqslant 0$, 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	$0.0 \leq SP \leq 1.0$
<0	Initializes the random number generator and returns the first random number	$0.0 \leq SP \leq 1.0$

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 $IP \leq 0$, 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 $IP \leq 0$, the entire integer is set to zero.

I2 = RT (I1,IP)

SHFT

Performs logical shift operations on integer variables.

1. **IS = SHFT (I)**: In this form, the variable is unchanged and the value is the variable itself; this form has no real use.
2. **IS = SHFT (I,IP1)**: performs a shift operation on the variable. If $IP1 > 0$, the shift is to the right; if $IP1 < 0$, the shift is to the left; if IP1=0, no shift occurs. This form is equivalent to the RS and LS functions.

Operation	Function	Equivalent SHFT function
Right shift	RS (I,IP)	**SHFT (I,IP)**
Left shift	LS (I,IP)	**SHFT (I,-IP)**
Right truncate	RT (I,IP)	**SHFT (I,IP-16,16-IP)**
Left truncate	LT (I,IP)	**SHFT (I,16-IP,IP-16)**

3. **IS = SHFT (I,IP1, IP2)**: Performs two shift operations, first by IP1 (setting zeroes in vacated bits), then by IP2 (setting zeroes in vacated bits). The sign of IP1 and IP2 determine the direction of the shift while their magnitude 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)

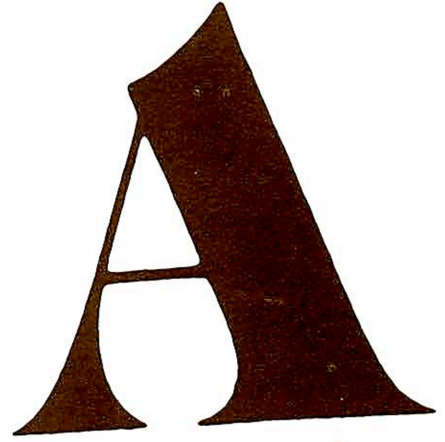
XOR

Performs a logical exclusive OR on a variable list of integers.

I = XOR (I1,I2,. . .,In)

IV

APPENDICES



Error messages

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. Alternatively, COMMON is offset an odd number of words and the compiler is trying to allocate words of a data element in two different segments. Re-arrange order of variables in COMMON so no element overlaps a segment boundary.

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 DEBUG IN R MODE

The -DEBUG (or -PROD) option was included for compilation in a mode other than 64V. Compilation will proceed as if the debugging option had not been included.

NO END STMT

The last statement in the source was not an END statement.

NO PATH TO STMT

18

The current statement does not have a statement number and the previous statement was an unconditional transfer of control. This will also be generated at the end of a program unit for labelled statements, if control cannot reach the statement.

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.

WARNING — DEBUG TURNS OFF OPT

Both the -DEBUG and -OPT (or -UNCOPT) options were selected. Compilation will proceed as if the optimization option had not been included.

WARNING — NO RETURN OR STOP

Occurs if either there is no STOP statement (main program) or RETURN statement (subroutine) at the end of the program unit. This does not mean there is no RETURN statement in a subroutine but that the RETURN statement immediately preceeding the END statement is missing.

WARNING — name — NEVER GIVEN A VALUE

Occurs only if -DEBUG option included. The local variable *name*, used in the program, never had a value assigned to it at any point in the program.

WARNING — name — PARAMETER IS BETTER

Occurs only if -DEBUG option included. The variable *name* was initialized in a DATA statement and remains constant throughout the program. It would be more efficient to assign a value with the PARAMETER statement.

WARNING — name — VARIABLE NOT USED

Occurs only if -DEBUG option included. The variable *name* was declared in a specification statement but not used in the program. Such variables are not accessible when using the source level debugger (DBG).

B
System defaults and constants

TERMINAL

full duplex
X-ON/X-OFF disabled

EDITOR (ED)

INPUT (TTY)
LINESZ 144
MODE NCKPAR
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	
NODEBUG	
NOFRN	
NOTRACE	
NOXREF	
SAVE	
STDOPT	
32R	

18

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)

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	HLT-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.
2. .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceding line.
3. Next byte specifies number of spaces to insert.
4. Next byte specifies number of lines to insert.

Conforms to ANSI X3.4-1968

The parity bit ('200) has been added for Prime-usage. Non-printing characters (^c) can be entered at most terminals by typing the (control) key and the character key simultaneously.

Table C-2. ASCII Character Set (Printing)

<table>
<tr><th>Octal Value</th><th>ASCII Character</th><th>Octal Value</th><th>ASCII Character</th><th>Octal Value</th><th>ASCII Character</th></tr>
<tr><td>240</td><td>.SP. (1)</td><td>300</td><td>@</td><td>340</td><td>‘ (9)</td></tr>
<tr><td>241</td><td>!</td><td>301</td><td>A</td><td>341</td><td>a</td></tr>
<tr><td>242</td><td>" (2)</td><td>302</td><td>B</td><td>342</td><td>b</td></tr>
<tr><td>243</td><td># (3)</td><td>303</td><td>C</td><td>343</td><td>c</td></tr>
<tr><td>244</td><td>$</td><td>304</td><td>D</td><td>344</td><td>d</td></tr>
<tr><td>245</td><td>%</td><td>305</td><td>E</td><td>345</td><td>e</td></tr>
<tr><td>246</td><td>&</td><td>306</td><td>F</td><td>346</td><td>f</td></tr>
<tr><td>247</td><td>' (4)</td><td>307</td><td>G</td><td>347</td><td>g</td></tr>
<tr><td>250</td><td>(</td><td>310</td><td>H</td><td>350</td><td>h</td></tr>
<tr><td>251</td><td>)</td><td>311</td><td>I</td><td>351</td><td>i</td></tr>
<tr><td>252</td><td>*</td><td>312</td><td>J</td><td>352</td><td>j</td></tr>
<tr><td>253</td><td>+</td><td>313</td><td>K</td><td>353</td><td>k</td></tr>
<tr><td>254</td><td>, (5)</td><td>314</td><td>L</td><td>354</td><td>l</td></tr>
<tr><td>255</td><td>-</td><td>315</td><td>M</td><td>355</td><td>m</td></tr>
<tr><td>256</td><td>.</td><td>316</td><td>N</td><td>356</td><td>n</td></tr>
<tr><td>257</td><td>/</td><td>317</td><td>O</td><td>357</td><td>o</td></tr>
<tr><td>260</td><td>0</td><td>320</td><td>P</td><td>360</td><td>p</td></tr>
<tr><td>261</td><td>1</td><td>321</td><td>Q</td><td>361</td><td>q</td></tr>
<tr><td>262</td><td>2</td><td>322</td><td>R</td><td>362</td><td>r</td></tr>
<tr><td>263</td><td>3</td><td>323</td><td>S</td><td>363</td><td>s</td></tr>
<tr><td>264</td><td>4</td><td>324</td><td>T</td><td>364</td><td>t</td></tr>
<tr><td>265</td><td>5</td><td>325</td><td>U</td><td>365</td><td>u</td></tr>
<tr><td>266</td><td>6</td><td>326</td><td>V</td><td>366</td><td>v</td></tr>
<tr><td>267</td><td>7</td><td>327</td><td>W</td><td>367</td><td>w</td></tr>
<tr><td>270</td><td>8</td><td>330</td><td>X</td><td>370</td><td>x</td></tr>
<tr><td>271</td><td>9</td><td>331</td><td>Y</td><td>371</td><td>y</td></tr>
<tr><td>272</td><td>:</td><td>332</td><td>Z</td><td>372</td><td>z</td></tr>
<tr><td>273</td><td>;</td><td>333</td><td>[</td><td>373</td><td>{</td></tr>
<tr><td>274</td><td><</td><td>334</td><td>\</td><td>374</td><td>|</td></tr>
<tr><td>275</td><td>=</td><td>335</td><td>]</td><td>375</td><td>}</td></tr>
<tr><td>276</td><td>></td><td>336</td><td>^ (7)</td><td>376</td><td>~ (10)</td></tr>
<tr><td>277</td><td>? (6)</td><td>337</td><td>__ (8)</td><td>377</td><td>DEL (11)</td></tr>
</table>

1. Space forward one position
2. Terminal usage — erase previous character
3. £ in British use
4. Apostrophe/single quote
5. Comma
6. Terminal usage — kill line
7. 1963 standard ↑; terminal use — logical escape
8. 1963 standard ←
9. Grave
10. 1963 standard ESC
11. Rubout — ignored

Conforms to ANSI X3.4-1968

1963 variances are noted

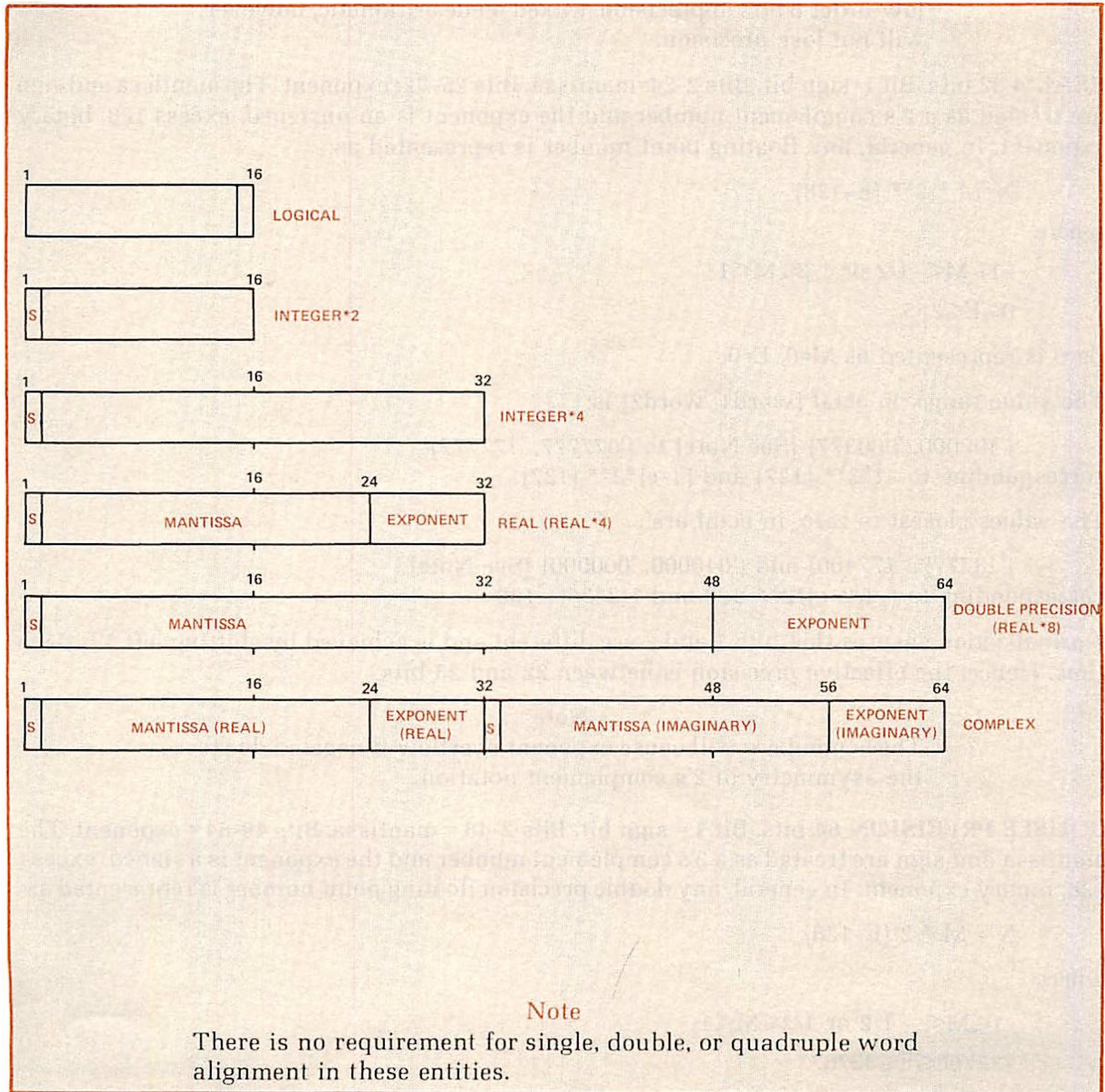
The parity bit ('200) has been added for Prime usage.

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.



Note

There is no requirement for single, double, or quadruple word alignment in these entities.

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.

MISC.

(in B format) 6-24
$ (FORTRAN address constants) 5-6
$ (in B format) 6-24
$INSERT statement 6-10
$INSERT, nesting not allowed 6-10
$X version, subroutines 8-1
, (singles quote in IBM format direct access READ) 6-17
' ' (single quote representation in ASCII string) 5-4
* (in B format) 6-24
+ (in B format) 6-24
, (in B format) 6-24
, (in FORMAT statement) 6-20
- (in B format) 6-24
-32R (compiler option) 2-9
-64R (compiler option) 2-9
-64V (compiler option) 2-9
-BIG (compiler option) 2-2
-BINARY (compiler option) 2-2
-DCLVAR (compiler option) 2-3
-DEBASE (compiler option) 2-3
-DEBUG (compiler option) 2-3
-DYNM (compiler option) 2-3
-DYNM option, compiler, use of 4-6
-ERRLIST (compiler option) 2-4
-ERRTTY (compiler option) 2-4
-EXPLIST (compiler option) 2-4
-FP (compiler option) 2-4
-INPUT (compiler option) 2-4
-INTL (compiler option) 2-4
-INTS (compiler option) 2-5
-LIST (compiler option) 2-5
-LISTING (compiler option) 2-5
-NOBIG (compiler option) 2-5
-NODCLVAR (compiler option) 2-5
-NODEBUG (compiler option) 2-5
-NOERRTTY (compiler option) 2-5
-NOFP (compiler option) 2-5
-NOTRACE (compiler option) 2-6
-NOXREF (compiler option) 2-6
-OPT (compiler option) 2-6
-PBECB (compiler option) 2-6
-PROD (compiler option) 2-6
-SAVE (compiler option) 2-6
-SOURCE (compiler option) 2-6
-STDOPT (compiler option) 2-6
-TRACE (compiler option) 2-6
-UNCOPT (compiler option) 2-6
-XREFL (compiler option) 2-7
-XREFS (compiler option) 2-7
• (in B format) 6-24
•AND• truth table 5-6
•FALSE• 5-4
•NOT• truth table 5-6
•OR• truth table 5-6
•TRUE• 5-4
/ (in FORMAT statement) 6-20
// (blank COMMON) 6-6
32R (compiler option) 2-9
64R (compiler option) 2-9
64V (compiler option) 2-9
64V-mode COMMON, optimization 4-4
: (FORTRAN octal numbers) 5-3

A

A input format 6-23
A output format 6-22
A register 2-9
A register defaults 2-9
Address constants 5-6
Address, call by 6-3
AND truth table 5-6
ANSI standard data storage D-1
Application library subroutines 7-3
Area TRACE statement 6-10
Arguments:
  function 7-2
  subroutine 7-3
Arithmetic:
  IF statement 6-12
  mixed mode 6-10
  operators 5-6
  vs. logical IF 4-6
Arrays: 5-5
  dummy argument, over 64K word COMMON 6-6
  in over 64K word COMMON 6-6
  segment-spanning 2-2
ASCII:
  character set C-1
  character strings 5-4
  characters, non-printing C-2
  characters, printing C-3
  data storage format D-3
  keyboard input C-1
  parity C-1
  Prime usage C-1
Assembly language, interface to 1-8
ASSIGN statement 6-11
Assigned GO TO statement 6-12
Assignment statements 6-10
Assignment statements, data mode rules, table 6-12
ATTDEV subroutine 6-15
Audience 1-1

B

B format, details 6-22
B output format 6-22
B register 2-9
B register defaults 2-9
BACKSPACE statement 6-26
Base areas, conversation 2-3
Batch environment 1-4
BIG (compiler option) 2-2
Binary file, compiler 2-2
Binary file, compiler (unit 3) 2-12
Binary files, concatenating 2-13
Binary READ statement 6-17
Binary WRITE statement 6-19
BINARY:
  (compiler option) 2-2
  (PRIMOS command) 2-13
Bit-device correspondence, compiler 2-12
Bit-mnemonic correspondence, A register 2-10
Bit-mnemonic correspondence, B register 2-10
Blank COMMON 6-5
BLOCK DATA statement 6-3
Block data subprogram 6-3
BLOCKDATA statement 6-3

C

Call by address 6-3
Call by value 6-3
CALL EXIT 1-4
CALL statement 6-27, 6-9
Change I/0 unit physical device correspondence 6-15
Changing record size 6-15
CHARACTER data storage format D-3
Character set, ASCII C-1
Character set, legal 5-1
Character string, input, list directed 6-18
Circular reasoning see proof by assumption
CLOSE (PRIMOS command) 2-13
CLOSE ALL 2-13
Closing files 2-13
COBOL, interface 1-8
Code, relative address 2-9
Code, segmented address 2-9
Codes, concordance 2-8
Coding statements 6-19

Coding strategy 3-1
Column 6 for continuation 5-2
Columns 73-80 5-2
Comment lines 5-2
Comments, use of 3-2
COMMON block F$IOBF 6-15
COMMON block LIST 6-6
COMMON blocks 6-5
COMMON blocks over 64K words 6-6
COMMON statement 6-5
COMO, use with TRACE 6-10
Compatibility, languages 1-1
Compilation statements 6-9
Compilation, end of, message 2-1
Compilation, V-mode vs. R-mode 4-4
Compiler error message 2-2
Compiler option:
- -32R 2-9
- -64R 2-9
- -64V 2-9
- -BIG 2-2
- -BINARY 2-2
- -DCVAR 2-3
- -DEBASE 2-3
- -DEBUG 2-3
- -DYNM 2-3
- -ERRLIST 2-4
- -ERRTTY 2-4
- -EXPLIST 2-4
- -FP 2-4
- -INPUT 2-4
- -INTL 2-4
- -INTL 2-4
- -INTS 2-5
- -LIST 2-5
- -LISTING 2-5
- -NOBIG 2-5
- -NODCLVAR 2-5
- -NODEBUG 2-5
- -NOERRTTY 2-5
- -NOFP 2-5
- -NOTRACE 2-6
- -NOXREF 2-6
- -OPT 2-6
- -PBECB 2-6
- -PROD 2-6
- -SAVE 2-6
- -SOURCE 2-6
- -STDOPT 2-6
- -TRACE 2-6
- -UNCOPT 2-6
- -XREFL 2-7
- -XREFS 2-7

Compiler:
- -DCLVAR usage 3-2
- -DYNM option, use of 4-6
- binary file 2-2
- binary file (unit 3) 2-12
- description 1-6
- devices, default 2-12
- error messages A-1
- error messages, print at terminal 2-4
- error messages, suppress printing 2-5
- file specifications, table 2-3
- file unit usage 2-12
- FORTRAN, defaults B-2
- global trace 3-3
- input file 2-4
- invoking 2-1
- listing file 2-5
- listing file (unit 2) 2-12
- listing, default 2-5
- listing, enable 2-5
- listing, expanded 2-4
- listing, full 2-5
- object file 2-2
- object file (unit 3) 2-12
- parameters 2-2
- source file 2-6
- source file (unit 1) 2-12
- syntax 2-1
- syntax checking 3-2
- warning message 2-2

Compiling 2-1
Compiling from peripheral devices 2-12
Compiling to peripheral devices 2-12
Complete cross reference 2-7
COMPLEX data storage format D-3
COMPLEX mode 6-5
Complex numbers 5-4
Composition, program 5-7
Computed GO TO statement 6-12
Concatenating binary files 2-13
Concatenating listing files 2-13
Concordance see also cross reference
Concordance address, over 64K word COMMON 6-6
Concordance codes 2-8
Concordance, compiler, enable 2-7
CONIOC 6-15
Conserve loader base areas 2-3
Constants: 5-2
- address 5-6
- range 5-2
- system B-1

Continuation lines 5-2
CONTINUE statement 6-11
Control flow, conversion 1-4
Control flow, program, monitoring 3-2
Control lines 5-2
Control statements 6-11
Conversion:
- control flow 1-4
- functions 1-4
- input/output 1-4
- program 1-2
- source language 1-2
- subroutines 1-4

CR (in B format) 6-24
Cross reference:
- see also concordance
- codes 2-8
- compiler, enable 2-7
- complete 2-7
- example 2-7
- explanation 2-7
- full 2-7
- partial 2-7
- short 2-7
- suppression 2-6

D

D input format 6-23
D output format 6-22
DATA statement 6-9
Data:
- definition statement 6-9
- mode convention, FORTRAN, overriding 6-4
- mode of function 6-3
- mode rules for assignment statements, table 6-12
- mode typing, parameter 6-5
- modes 6-5
- storage format, ASCII D-3
- storage format, CHARACTER D-3
- storage format, COMPLEX D-3
- storage format, DOUBLE PRECISION D-2
- storage format, INTEGER*2 D-2
- storage format, INTEGER*4 D-2
- storage format, LOGICAL D-2
- storage format, REAL*4 D-2
- storage, ANSI standard D-1
- types 6-5
- types, FORTRAN, memory formats D-1

Database management system, interface to 1-7
DBG (debugger) 3-1
DCLVAR (compiler option) 2-3
DEBASE (compiler option) 2-3
DEBUG (compiler option) 2-3
Debugger code generation 2-3
Debugger code generation, suppress 2-5
Debugger, source level 3-1
Debugging 3-1
DECODE, formatted, statement 6-20
DECODE, list directed, statement 6-20

Default:
compiler devices 2-12
compiler listing 2-5
object code 2-9
record size 6-15
Defaults:
A register 2-9
B register 2-9
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-1
segmented loader B-1
system B-1
Delimiters, format 6-20
Delimiters, list directed 6-17
Descriptor repetition 6-20
Development, program 1-3
Development control statements 6-26
Device-bit correspondence, compiler 2-12
Devices, compiler, default 2-12
DIMENSION statement 6-8
Dimensioning, not allowed in SAVE statement 6-8
Direct access 6-14
Direct access and ATTDEV subroutine 6-14
Direct access and the Editor 6-14
Direct access READ statements 6-17
Direct access WRITE statements 6-19
Direct access, IBM compatibility 6-14
Direct access, use of 6-14
DO:
loop index 6-12
loop optimization 2-6, 4 -1
loop optimization, suppress 2-6
loop, one-trip 6-12
loops, implied 6-19
loops, nesting 6-11
statement 6-11
Documents, related 1-2
DOUBLE PRECISION see also REAL*8
DOUBLE PRECISION data storage format D-2
DOUBLE PRECISION mode 6-5
Double precision numbers 5-3
Dummy argument arrays, over 64K word COMMON 6-6
Dynamic allocation of local storage 2-3
DYNM (compiler option) 2-3
DYNM option, compiler, use of 4-6

E

E input format 6-23
E output format 6-21
ECBs, load into procedure frame 2-6
ED, defaults B-1
Editor defaults B-1
EDitor, description 1-8
Editor, use of on direct access files 6-14
Enable compiler concordances 2-7
Enable compiler cross references 2-7
Enable compiler listings 2-5
Enable flagging of undeclared variables 2-3
Enable global trace 2-6
ENCODE statement 6-20
End of compilation message 2-1
END statement 6-12
END= 6-16
ENDFILE statement 6-26
Ending main program 1-4
Environment:
batch 1-4
interactive 1-4
phantom user 1-4
program, list 1-4
EQUIVALENCE statement 6-8
ERR= 6-16
ERRLIST (compiler option) 2-4
Error:
message, compiler 2-2
messages A-1
messages, compiler A-1
messages, compiler, print only 2-4
messages, compiler, print at terminal 2-4
ERRTTY (compiler option) 2-4
Execution defaults B-2
Exit, normal 6-11
Expanded compiler listing 2-4
EXPLIST (compiler option) 2-4
Extended intrinsic functions 7-1
Extended range, optimization 2-6
Extensions 1-2
External procedure statements 6-9
EXTERNAL statement 6-9

F

F input format 6-23
F output format 6-21
F$IOBF COMMON block 6-15
FALSE 5-4
Field descriptor, format 6-20
File specifications, compiler, table 2-3
File system summary 1-5
File unit usage, compiler 2-12
Floating point skip operations generate 2-4
Floating point skip operations, suppress 2-5
Format delimiters 6-20
Format field descriptor 6-20
Format lines, rescanning 6-21
FORMAT statement 6-20
Format, line 5-1
FORMAT, use of parameters not allowed 6-5
Formats as variables 6-22
Formats in input statements, table 6-23
Formats in output statements, table 6-21
Formats, memory, FORTRAN data types D-1
Formatted DECODE statement 6-20
Formatted printer control 6-26
Formatted READ statement 6-16
Formatted WRITE statement 6-19
Forms management system, interface to 1-7
FORTRAN compiler defaults B-2
FORTRAN data mode convention, overriding 6-4
FORTRAN data types, memory formats D-1
FORTRAN extensions, Prime 1-2
FORTRAN function library 8-1
FORTRAN function reference 8-1
FORTRAN functions 7-1
FORTRAN functions, list 8-2
FORTRAN language elements 5-1
FORTRAN language tutorial books 1-1
FORTRAN library functions 7-1
FORTRAN library, V-mode 8-1
FORTRAN math subroutines 7-3

FORTRAN mathematical functions table 1-5
FORTRAN statements 6-1
FORTRAN under PRIMOS 1-2
FORTRAN unit number, physical devices, table 6-16
FORTRAN, Prime's, overview 1-1
FP (compiler option) 2-4
FTN (PRIMOS command) 2-1
FTN, defaults B-2
FTN, FORTRAN compiler 2-1
FTNLIB 8-1
Full compiler listing 2-5
Full cross reference 2-7
FULL LIST statement 6-9
Function, structure of 7-1
Function:
FUNCTION:
  arguments 7-2
  calls 6-26
  calls, optimization 4-4
  mode 6-3
  mode typing 7-1
  reference, FORTRAN 8-1
  rules 7-2
  statement 6-3, 7-1
  subprograms, user-defined 7-1
Functions:
  conversion 1-4
  extended intrinsic 7-1
  FORTRAN 7-1
  FORTRAN library 7-1
  FORTRAN, list 8-2
  statement 7-1
  trigonometric 8-1

G

G input format 6-23
G output format 6-21
Generalized subscripts 5-5
Generate debugger code 2-3
Generate floating point skip operations 2-4
Generate productions code 2-6
Global mode specification 6-5
Global SAVE 6-9
Global trace, enable 2-6
Global trace, suppress 2-6
Global/IMPLICIT conflict 6-5
GO TO, assigned, statement 6-12
GO TO, computer, statement 6-12
GO TO, unconditional, statement 6-12

H

H output format 6-22
Header statements for subprograms 6-3
Hollerith constants 5-4

I

I input format 6-23
I output format 6-22
I/O unit physical device correspondence, change 6-15
IBM compatibility, direct access files 6-14
IF statements, optimization 4-4
IF, arithmetic, statement 6-12
IF, logical vs. arithmetic 4-6
IF, logical, statement 6-12
IFTNLB 8-1
Implementation, over 64K word COMMON 6-7
Implemented statements, list 6-1
IMPLICIT and intrinsic functions 8-1
IMPLICIT statement 6-4
IMPLICIT/global conflict 6-5
Implied DO loops 6-19
In-line comments, use of 3-2
Indention, use of 3-2
Index, DO loop 6-12
INPUT (compiler option) 2-4
Input/output optimization 4-4
Input/output, for conversion 1-4
Input:
  file, compiler 2-4
  scale factors 6-25
  specifications, compiler 2-4
  Specifications, compiler 2-10
  statements 6-14
  statements, formats in, table 6-23
INSERT see $INSERT
INTEGER see also INTEGER*2, INTEGER*4
INTEGER mode 6-5
INTEGER*2 see also INTEGER, INTEGER*4
INTEGER*2 data storage format D-2
INTEGER*2 default 2-5
INTEGER*2 mode 6-5
INTEGER*4 see also INTEGER, INTEGER*4
INTEGER*4 data storage format D-2
INTEGER*4 default 2-4
INTEGER*4 mode 6-5
Integer:
  division optimization 4-6
  random number generator 8-5
  sign extension 8-1
  truncation 8-2
Integers: 5-3
  in subroutine calls 2-4
  long 5-3
  short 5-3
Interactive environment 1-4
Interface to assembly language 1-8
Interface to COBOL 1-8
Interface to database management system 1-7
Interface to Forms management system 1-7
Interface to PL/I subset G 1-8
Interface to PMA 1-8
INTL (compiler option) 2-4
Intrinsic functions and IMPLICIT 8-1
Intrinsic functions, extended 7-1
INTS (compiler option) 2-5
Item TRACE statement 6-10

K

Keyboard input, ASCII characters C-1

L

L input format 6-23
L output format 6-22
Language compatibility 1-1
Language elements, FORTRAN 5-1
Language interface 1-8
Language, source, conversion 1-2
Legal character set 5-1
Libraries, descriptions 1-6
Library:
  calls optimization 4-5
  FORTRAN function 8-1
  functions, FORTRAN 7-1
Line format 5-1
LIST:
  (COMMON block) 6-6
  (compiler option) 2-5
List:
  directed character string input 6-18
  directed DECODE statement 6-20
  directed delimiters 6-17
  directed numerical input 6-18
  directed READ statements 6-17
  FORTRAN functions 8-2
  statement 6-9
LISTING:
  (compiler option) 2-5
  (PRIMOS command) 2-12

Listing:
compiler, default 2-5
compiler, enable 2-7
compiler, expanded 2-4
file, compiler 2-5
file, compiler (unit 2) 2-12
file, spooling 2-5
files, concatenating 2-13
full, compiler 2-5
LOAD, defaults B-1
Load:
ECBs into procedure frames 2-6
sequence, optimization 4-3
Loader:
conservation of base areas 2-3
description 1-6
SEG, defaults B-1
segmented, defaults B-1
Local storage, dynamic allocation 2-3
Local storage, static allocation 2-6
LOGICAL mode 6-5
LOGICAL, data storage format D-2
Logical:
constants 5-4
functions, mixed integers in 8-2
IF statement 6-12
operators 5-6
shift operator 8-7
vs. Arithmetic IF 4-6
Long and short integers, mixing 8-1
Long integers 5-3

M

Main program, ending 1-4
Math subroutines, FORTRAN 7-3
Mathematical functions, FORTRAN, table 1-5
Matrix subroutines, table 1-6
Memory allocation, optimization 4-3
Memory formats, FORTRAN data types D-1
Message:
end of compilation 2-1
error A-1
error, compiler 2-2, A-2
warning, compiler 2-2
MIDAS see also Multiple Index Direct Access System
MIDAS, descriptor 1-8
Mixed integers in logical functions 8-2
Mixed mode arithmetic 6-10
Mixing long and short integers 8-1
Mnemonic-bit correspondence, A register 2-11
Mnemonic-bit correspondence, B register 2-11
Mode:
date 6-5
data see data type
mixing rules 6-11
of function 6-11
specification statement 6-4
specifications, global 6-5
typing, function 7-1
Modular program structure 3-1
Monitoring program control flow 3-2
Multi-dimensioned arrays, optimization 4-3
Multiple Index Direct Access System see also MIDAS

N

Nesting DO loops 6-11
Nesting, not allowed in $INSERT files 6-10
NO LIST statement 6-9
NOBIG (compiler option) 2-5
NODCLVAR (compiler option) 2-5
NODEBUG (compiler option) 2-5
NOERRTTY (compiler option) 2-5
NOFP (compiler option) 2-5
Non-printing ASCII characters C-2
Normal exit 6-11
NOT truth table 5-6
NOTRACE (compiler option) 2-6
NOXREF (compiler option) 2-6
Numerical input, list directed 6-18

O

Object:
code generation 2-9
code, default 2-9
file, compiler 2-2
file, compiler (unit 3) 2-12
One-trip DO loop 6-12
Operands 5-2
Operator priority 5-7
Operators 5-6
Operators, arithmetic 5-6
Operators, logical 5-6
Operators, relational 5-7
OPT (compiler option) 2-6
Optimization 4-1
Optimization of DO loops, suppress 2-6
Optimization:
64V-mode COMMON 4-4
DO loops 2-6, 4-1
functions calls 4-4
IF statements 4-4, 4-6
input/output 4-4
integer division 4-6
library calls 4-5
load sequence 4-3
memory allocation 4-3
multi-dimensioned arrays 4-3
parameter statements 4-5
statement functions and subroutines 4-5
statement number 4-2
statement sequence 4-5
unconditional 2-6
Options, compiler see also parameters, compiler
OR truth table 5-6
Order of statements in a program 5-8
Organization 1-1
Output/input optimization 4-4
Output:
scale factors 6-25
specifications, compiler 2-2, 2-10
statements 6-14
statements, formats in, table 6-21
Over 64K word COMMON blocks 6-6
Over 64K word COMMON, arrays 6-6
Over 64K word COMMON, concordance address 6-6
Over 64K word COMMON, dummy argument array 6-6
Over 64K word COMMON, implementation 6-7
Over 64K word COMMON, programming considerations 6-7
Over 64K word COMMON, restrictions 6-7
Overriding FORTRAN data mode convention 6-4
Overview of Prime's FORTRAN 1-1

P

PARAMETER statement 6-5
Parameter: , 5-4
compiler see also options, compiler
compiler 2-2
data mode typing 6-5
not allowed in FORMAT statement 6-5
statements optimization 4-5
usage 6-5
Parity, ASCII C-1

Partial cross reference 2-7
PAUSE statement 6-13
PAUSE, recovering from 6-13
PBECB (compiler option) 2-6
Peripheral devices with compiler 2-12
Petitio principii see circular reasoning
PFTNLB 8-1
Phantom user environment 1-4
Physical device FORTRAN unit numbers, table 6-16
Physical device I/O unit correspondence, change 6-15
PL/I subset G, interface to 1-8
PMA see also Prime Macro Assembly Language
PMA, interface to 1-8
Prime extension to FORTRAN 1-2
Prime Macro Assembly Language see also PMA
PRIMOS defaults B-2
PRIMOS command:
  BINARY 2-13
  CLOSE 2-13
  FTN 2-1
  LISTING 2-12
PRIMOS, FORTRAN under 1-2
Print compiler error messages at terminal 2-4
Print only error messages 2-4
PRINT statement 6-15
Printer control, formatted 6-26
Printing ASCII characters C-3
Priority of operators 5-7
Procedure frames, load ECBs into 2-6
PROD (compiler option) 2-6
Production code, generate 2-6
Program
  compositions 5-7
  control flow, monitoring 3-2
  conversion 1-2
  development 1-3
  environments, list 1-4
  order of statements in 5-8
  structure, modular 3-1
Programming considerations, over 64K word COMMON 6-7
Proof by assumption see petitio principii

R

R-mode vs. V-mode compilation 4-4
Random number generator, integer 8-5
Random number generator, real 8-6
Range of constants 5-2
READ:
  binary, statement 6-17
  direct access, statements 6-17
  formatted, statement 6-16
  list directed, statement 6-17
  statements 6-16
REAL see also REAL*4
REAL mode 6-5
Real numbers 5-3
Real random number generator 8-6
REAL*4 see also REAL
REAL*4 data storage format D-2
REAL*4 mode 6-5
REAL*8 see also DOUBLE PRECISION
REAL*8 mode 6-5
REC= 6-17, 6-19
Record:
  size over 128 words 6-15
  size, changing 6-15
  size, default 6-15
Recovering from PAUSE 6-13
Recursive subroutines 6-9
Related documents 1-2
Relational operators 5-7
Relative address code 2-9
Repetition, field descriptor 6-20
Representation, ASCII character strings 5-4
Representation, complex numbers 5-4
Representation, double precision numbers 5-3
Rescanning format lines 6-21
Resources, system, list 1-5
Restrictions on over 64K word COMMON 6-7
RETURN statement 6-14
REWIND statement 6-26
Rules for functions 7-2
Rules for subroutines 7-3
Rules for variables 5-4
Rules, mode mixing 6-11
Run-time statements 6-9

S

SAVE (compiler option) 2-6
SAVE statement 6-8
SAVE statement, dimensioning not allowed in 6-8
SAVE, global 6-9
Scale factors 6-25
SEG loader defaults B-1
SEG utility, description 1-6
Segment-spanning arrays 2-2
Segmented address code 2-9
Segmented loader defaults B-1
Sequence numbers 5-2
Setting A register 2-9
Setting B register 2-9
Short and long integers, mixing 8-1
Short call subroutines 8-1
Short cross reference 2-7
Short integers 5-3
Sign extension, integer 8-1
Skip operations, floating point, generate 2-4
Skip operations, floating point, suppress 2-5
SOURCE (compiler option) 2-6
Source:
  file, compiler 2-6
  file, compiler (unit 1) 2-12
  language conversion 1-2
  level dubugger 3-1
Spacing, using of 3-2
Specification statements 6-4
Spooling the listing file 2-5
Statement:
  data definition 6-9
  functions 7-2
  functions and subroutine optimization 4-5
  lines 5-1
  number, optimization 4-2
  sequence optimization 4-5
Statements: 6-1
  assignment 6-10
  coding 6-19
  compilation 6-9
  control 6-11
  device control 6-26
  External procedure 6-9
  grouped, list 6-2
  header, for subprograms 6-3
  implemented, list 6-1
  input 6-14
  order of in programs 5-8
  output 6-14
  READ 6-16
  run-time 6-9
  specification 6-4
  storage 6-5
  WRITE 6-18
Static allocation of local storage 2-6
STDOPT (compiler option) 2-6
STOP statement 6-14
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*2 D-2
  data, LOGICAL D-2
  data, REAL*4 D-2
Storage statements 6-5

Storage, ANSI standard D-1
Storage, local, dynamic, allocation 2-3
Storage, local, static allocation 2-6
Strategy, coding 3-1
Structure of function subprogram 7-1
Structure of subroutine subprograms 7-3
Structure, program, modular 3-1
Subprogram, block data 6-3
Subprograms, functions, user-defined 7-1
Subprograms, header statements for 6-3
SUBROUTINE statement 6-3, 7-3
Subroutine:
  arguments 7-3
  ATTDEV 6-15
  calls 6-27
  calls, integers in 2-4
  rules 7-3
  subprogram, structure of 7-3
Subroutines:
  $X versions 8-1, 7-2
  application library 7-3
  conversion 1-4
  FORTRAN math 7-3
  matrix, table 7-3
  PRIMOS system 7-3
  recursive 6-9
  short call 8-1
  user-defined 7-3
Subscripts:
  generalized 5-5
  maximum number of 5-5
Suppress:
  cross reference 2-6
  debugger code generation 2-5
  DO loop optimization 2-6
  flagging of undeclared variables 2-5
  floaring point skip operations 2-5
  globals trace 2-6
  printing of compiler error messages 2-5
Syntax:
  checking, compiler 3-2
  compiler 2-1
System:
  constants B-1
  defaults B-1
  resources 1-5

T

T input format 6-23
T output format 6-22
Terminal defaults B-1
Trace:
  global, compiler 3-3
  global, enable 2-6
  global, suppress 2-6
TRACE:
  (compiler option) 2-6
  area, statement 6-10
  item, statements 6-10
  statements, use of 3-2
  use with COMO 6-10
Trigonometric functions 8-1
TRUE 5-4
Truncation, integer 8-2
Truth tables 5-6
Tutorial books, FORTRAN language 1-1
Type, data see also data mode
Types, data 6-5

U

Unconditional GO TO statement 6-12
Unconditional optimization 2-6
UNCOPT (compiler option) 2-6
Undeclared variables, enable flagging 2-3
Undeclared variables, suppress flagging 2-5
User-defined function subprograms 7-1
User-defined subroutines 7-3

V

V-mode FORTRAN library 8-1
V-mode vs. R-mode compilation 4-4
Value, call by 6-3
Variable rules 5-4
Variables 5-4
Variables, formats as 6-22

W

WARNINGS 2-2
WRITE:
  binary, statement 6-19
  direct access, statements 6-19
  formatted, statement 6-19
  statements 6-18

X, Y, Z

X input format 6-23
X output format 6-22
XREFL (compiler option) 2-7
XREFS (compiler option) 2-7
Z (in B format) 6-24