Prime Computer, Inc.

PRIME

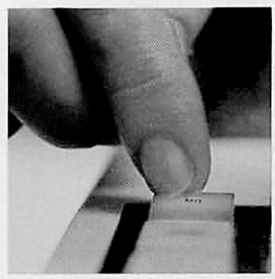
DOC4029-4LA
FORTRAN 77 Reference Guide
Revision 19.4












FORTRAN 77
Reference Guide

Fourth Edition

by

Evelyn Burns

Updated for Rev. 21.0

by

J. Ornstein and D. Laukaitis

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 21.0 (Rev. 21.0).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

PRINTING HISTORY

First Edition (IDR4029) January 1980 for Release 17.0
Second Edition (DOC4029-183) January 1983 for Release 18.3
Third Edition (DOC4029-192) June 1983 for Release 19.2
Fourth Edition (DOC4029-4LA) April 1985 for Release 19.4
Update 1 (UPD4029-41A) August 1986 for Release 20.2
Update 2 (UPD4029-42A) July 1987 for Release 21.0

CREDITS

Editorial: Margaret Hill, Bill Modlin
Project Support: Margaret Taft, Len Bruns
Illustration: Mingling Chang
Production: Judy Gordon

PRINTING HISTORY — FORTRAN 77 Reference Guide

Edition	Date	Number	Software Release
First Edition	January, 1980	IDR4029	17.0
Second Edition	January, 1982	DOC4029-183	18.3
Third Edition	June, 1983	DOC4029-192	19.2
Fourth Edition	April 1985	DOC4029-4LA	19.4

In document numbers, L indicates loose-leaf. This book is also available in perfect-bound form, as DOC4029-4PA.

CUSTOMER SUPPORT CENTER

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)
1-800-343-2320 (within other states)
1-800-541-8888 (within Alaska)
1-800-651-1313 (within Hawaii)

HOW TO ORDER TECHNICAL DOCUMENTS

Obtain an order form, a catalog, and a price list from one of the following:

Inside U.S.

Software Distribution
Prime Computer, Inc.
74 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053

Outside U.S.

Contact your local Prime subsidiary or distributor.

Contents

About This Book

This document is a programmer's guide to the FORTRAN 77 language as implemented on the Prime system. You are expected to be familiar with some version of FORTRAN, and with programming in general, but not necessarily with Prime computers.

If you are familiar with programming but not with FORTRAN, you should consult an appropriate FORTRAN 77 textbook. Here are some examples of textbooks you may find helpful:

Davis, Gordon B., Hoffman, Thomas R., FORTRAN 77: A Structured Disciplined Style, McGraw-Hill, Inc., New York, 1983

Katzan, Harry, FORTRAN 77, Van Nostrand Reinhold Company, New York, 1979

Wagener, Jerrold L., Principles of FORTRAN 77 Programming, John Wiley and Sons, New York, 1980

HOW TO USE THIS BOOK

This book is divided into three parts and a set of appendixes:

PART I -- OVERVIEW OF FORTRAN 77 AND PRIME F77

- Chapter 1 describes the different parts of an F77 statement such as symbols, constants, variables, arrays, etc.
- Chapter 2 gives you general information concerning FORTRAN and introduces basic facts needed before writing FORTRAN programs on Prime equipment.

PART II -- PRIME F77 LANGUAGE REFERENCE

- Chapter 3 describes specification statements, which define characteristics of symbols used in the program, such as data types, array dimensions, etc.
- Chapter 4 describes assignment statements, which define values used in the program.
- Chapter 5 describes control statements, which transfer control from one point in the program to another.
- Chapter 6 describes F77 I/O statements, data storage, and file types.
- Chapter 7 describes the FORMAT statements used in conjunction with formatted I/O statements.
- Chapter 8 discusses subprograms, both user-written and all the intrinsic functions that are supplied by the F77 compiler.

PART III -- WORKING WITH PRIME F77

- Chapter 9 describes how to invoke and use the F77 compiler.
- Chapter 10 describes how to use the PRIMOS commands BIND and RESUME to link and execute F77 programs.
- Chapter 11 describes how to locate errors that occur during compile, load, or execution time using Prime's Source Level Debugger, DBG. (This is a separately priced product.)
- Chapter 12 presents programming considerations and procedures for improving the performance of F77 programs.

Appendixes

- Appendix A lists the Prime Extended Character Set, which F77 uses.
- Appendix B contains two F77 programming examples.
- Appendix C describes the techniques required for converting Prime FTN programs to F77.
- Appendix D illustrates how Prime F77 data types are represented in memory.
- Appendix E contains V-mode and I-mode programs illustrating the use of the SHORTCALL statement.
- Appendix F contains the violations of the ANSI Standard that are flagged by the -STANDARD compiler option.
- Appendix G describes the use of Search Rules Facility in conjunction with INCLUDE and $INSERT statements.
- Appendix H contains an alphabetic summary of the F77 Intrinsic Functions.

RELATED DOCUMENTS

In addition to the FORTRAN 77 Reference Guide, there are several books describing other Prime utilities that will help you with your programming on Prime equipment. These documents are described below.

Prime User's Guide

Instructions for creating, loading, and executing programs in Prime FORTRAN 77 or any Prime language, plus extensive additional information on Prime system utilities for programmers, are found in the Prime User's Guide. Little general information about using the Prime computer system is presented here. The user's guide and this reference guide are complementary documents. Both are essential for programming on Prime machines.

FORTRAN 77 Programmer's Companion

The FORTRAN 77 Programmer's Companion is a pocket-size guide that contains a summary of extracts from the Reference Guide. Included are language statement formats, compiler options, data tables, functions, and reference tables.

FORTRAN Reference Guide

The FTN language is described in the FORTRAN Reference Guide. Those involved with converting programs from FTN to F77 should have a copy of that guide, since it contains some information that applies to FTN but not F77. Such information is not reiterated in this guide. See Appendix C for information on the conversion of FTN programs to F77.

New User's Guide to EDITOR and RUNOFF

EDITOR is Prime's line-oriented text editing system. RUNOFF is a text processor for formatting text. The New User's Guide to EDITOR and RUNOFF provides complete information on these utilities.

EMACS Primer and EMACS Reference Guide

EMACS, Prime's screen editor, can also be used to enter and modify source code and text files. The Primer is a user's guide designed for people with little or no experience with screen editors. The Reference Guide describes the full use of the screen editor. EMACS is a separately priced product.

Subroutines Reference Guide Series

This 4-volume series documents the PRIMOS operating system and application-level subroutines for advanced programmers who wish to incorporate them into their programs.

Programmer's Guide to BIND and EPFs

This document introduces BIND, Prime's new linking utility for creating Executable Program Formats (EPFs). The Programmer's Guide to BIND and EPFs contains a complete dictionary of all BIND commands as well as a dictionary of EPF-related PRIMOS commands and new subroutines that apply to EPFs. It also provides a discussion of programming restrictions and limitations with EPFS and how to build an EPF library.

Advanced Programmer's Guide Series

The Advanced Programmer's Guide series provides an in-depth discussion of the new command environment available with EPFs. It also provides a detailed description of the linking utility, BIND, that is used to create these dynamic runfiles.

Assembly Language Programmer's Guide

This document contains information that you will need to write programs in the Prime Macro Assembler (PMA) language.

Guide to Prime User Documents

This document contains a complete listing of books currently available for Prime products. It includes information on what each book contains, what product the book documents, when and if the book was updated, and how to order Prime documents.

PRIMOS Commands Programmer's Companion

This pocket-size companion contains a brief statement of format and usage for all PRIMOS user commands.

Source Level Debugger User's Guide

This book describes the use of Prime's interactive debugging product, DBG. The Source Level Debugger is a separately priced product used for locating errors that occur upon execution of your program. This book explains the concepts, conventions, and use of the Debugger. A complete list and explanation of all Debugger commands is included.

The ANSI Standard

The definitive reference for FORTRAN 77 is ANSI X3.9-1978 Programming Language FORTRAN. Every installation that uses FORTRAN 77 extensively should have a copy of this standard, which may be obtained from American National Standards Institute, 1430 Broadway, New York, NY, 10018.

Other Sources of Information

In addition to the documents listed above, please consider the following sources when looking for information about the F77 compiler:

- The Software Release Document, also called an MRU, released at each software revision. This document contains a summary of new features and changes in Prime's user software.
- Prime's online HELP files. Information on PRIMOS commands is displayed at your terminal, including a cumulative list of manuals, updates, etc.

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Command and statement formats show the syntax of commands, program language statements, and callable routines. Examples illustrate the uses of these commands, statements, and routines in typical applications. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
Abbreviations	If a command or statement has an abbreviation, it is indicated by underlining, or in some cases, the shortest acceptable form of the command is shown.	LOGOUT Abbreviation: LO
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	LOGIN user-id
Underlining in Examples	In examples, user input is underlined but system prompts and output are not.	OK, stat units USR=LAURAJ ONO NO FILE UNITS OPEN OK,
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items.	SPOOL [-LIST / -CANCEL]
Braces { }	Braces enclose a vertical list of items. Choose one and only one item.	CLOSE { filename / ALL }

Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	DIM array (row,col)
Hyphen -	Wherever a hyphen appears in a command line option, it is a required part of that option.	SPOOL -LIST
(CR)	The (CR) symbol indicates a single carriage return, which is generated by hitting the RETURN key on most terminals.	

ADDITIONAL DOCUMENTATION CONVENTIONS

Convention	Explanation	Example
Shading	Shading around a section of text indicates a Prime extension to or restriction on the ANSI standard.	REAL*16

Filename conventions

Convention	Explanation	Example
filename.language or filename	Source file	MYPROG.F77
filename.BIN or B_filename	Binary (object) file	MYPROG.BIN
filename.LIST or L_filename	Listing file	MYPROG.LIST
filename.RUN	Saved executable runfile	MYPROG.RUN

Filenames may be comprised of 1 to 32 characters inclusive, the first character of which must be nonnumeric. Names should not begin with a hyphen (-) or underscore (_). Filenames may be composed only of the following characters: A - Z, 0 - 9, _ # $ & - * . and /.

Note

On some devices, the underscore (_) may print as backarrow (←).

PRIME ADDRESS SPACE MEASUREMENT UNITS

- byte — 8 bits; 1 Prime ECS character.
- halfword — a unit of address space two bytes (16 bits) in size.
- word, fullword — a unit of address space four bytes (32 bits) in size.

PART I

Overview

1
Introduction to F77

DEFINITIONS

There are many versions of FORTRAN. In this guide, you will find the following terms to describe them:

Term	Definition
FORTRAN	A mathematically oriented programming language developed by IBM in the 1950s.
FORTRAN 66	A standardized FORTRAN, defined in the American National Standards Institute (ANSI) publication ANSI X3.9-1966.
FORTRAN IV	Any version of FORTRAN that is based on ANSI X3.9-1966 and contains extensions developed by a particular computer manufacturer.
FTN	Prime FORTRAN IV
FORTRAN 77	A new standardized FORTRAN, defined in the ANSI publication ANSI X3.9-1978.

F77	Prime's extended version of FORTRAN 77. The F77 language conforms fully to ANSI X3.9-1978.

Certain FORTRAN-specific terms used in this introduction are defined at the beginning of Chapter 2.

FORTRAN 77

In 1978, ANSI published ANSI X3.9-1978 Programming Language FORTRAN. This standard defines a new version of FORTRAN, called FORTRAN 77. The new FORTRAN includes and standardizes nearly all the useful extensions to FORTRAN 66 developed by individual manufacturers. The result is a comprehensive, well-defined, and powerful language.

NEW FEATURES IN FORTRAN 77

FORTRAN 77 provides many capabilities additional to those of FORTRAN 66. Some of them have been used in nearly all manufacturers' versions of FORTRAN IV, but have not previously been defined in any standard. Many of them were incorporated into FTN on the basis of preliminary documents released by ANSI, to facilitate the eventual conversion of FTN programs to F77.

The features available in FORTRAN 77 but not in FORTRAN 66 are as follows.

Data Declaration Capabilities

- A statement to name the main program (PROGRAM statement)
- An implicit type-rule for default typing of data items by first letter (IMPLICIT statement)
- Named constants (PARAMETER statement)
- A CHARACTER data type
- Arrays with up to seven dimensions
- Explicit lower bounds for array dimensions
- Array bounds with negative, 0, or positive values
- Integer constant expressions in array-bound specifications

Execution-time Capabilities

- Operations to concatenate and extract substrings from CHARACTER data
- Use of an array name, character substring, or implied-DO list in a DATA statement
- Use of integer expressions (rather than just integers) for array subscripts, selection values for computed GO TOs, and file units referred to in BACKSPACE, ENDFILE, and REWIND statements
- Use of integer, real, or double precision expressions for DO-loop and implied-DO index and control values
- DO and implied DO loops that may execute zero times and have negative incrementation values
- A block-IF statement, with subsidiary ELSE IF, ELSE, and END IF statements, for conditional execution of blocks of statements
- Use of a format statement label in an ASSIGN statement
- Use of decimal digits or a character string in a PAUSE or STOP statement

Subprogram Capabilities

- Multiple entry points to subprograms
- Alternate returns in subroutines
- Differentiation between external (user-supplied) and intrinsic (built-in) functions
- Generic names for intrinsic functions
- Functions with no arguments
- More than one block data subprogram

Input/Output Capabilities

- Direct-access files
- List-directed I/O
- Internal (storage-to-storage) formatted data transfer

- Statements to open and close files, and to inquire about the status of a file
- Additional edit-control descriptors for formatted I/O, such as sign control, blank editing, and tabbing

PRIME EXTENSIONS TO FORTRAN 77

Unextended FORTRAN 77 already includes features to perform nearly every programming task for which the FORTRAN language is appropriate. Prime has avoided extending its FORTRAN 77 unnecessarily, since needless extensions would serve mostly to reduce compatibility between F77 and other versions of FORTRAN 77.

Prime has extended its FORTRAN 77 for the following reasons:

- To provide added power and convenience of use to the language
- To take advantage of particular features of the Prime computer system
- To provide the maximum possible compatibility with FTN, and substantial compatibility with IBM and other manufacturers' versions of FORTRAN IV. See Appendix C for information on the conversion of FTN programs to F77.

The Prime extensions of greatest interest to a new F77 user are listed below. All Prime F77 extensions are described in detail at appropriate places later in this guide. Throughout this book shading has been used to indicate Prime F77 extensions. The extensions are:

- Variable and array names may have up to 32 characters, may contain lowercase letters, and may contain the characters "$" and "_".
- Comments may appear anywhere in a statement.
- Precision specifications for the FORTRAN data types are provided. REAL*16 (quadruple floating point precision), COMPLEX*16, INTEGER*2, LOGICAL*2, and LOGICAL*1 data types have been added.
- Extended intrinsics to deal with the extended data types are provided.
- Octal constants are accepted in F77 source text.
- Data may be initialized in a type-declaration statement.
- CHARACTER and non-CHARACTER data may be equivalenced and may coexist in the same COMMON block.

- All COMMON block data is static. Blank COMMON may be initialized.
- IBM syntax for direct-access READs and WRITEs is accepted.
- Recursion is permitted in subroutines, though not in functions.
- The B field descriptor for formatting business data (similar to PICTURE formatting in COBOL and PL/I) is provided.
- Files can be automatically inserted into the source file by the compiler.

There are various other extensions that allow certain FTN constructs, which are not standard in FORTRAN 77, to be accepted by the F77 compiler. These are described in Appendix C.

PRIME F77 RESTRICTIONS

The segmented nature of the Prime virtual memory architecture imposes a few restrictions on F77 programs. None of these are contrary to the ANSI standard or need interfere with program design.

- The executable code (exclusive of data storage) for a program unit may not occupy more than one segment (128K bytes).
- No program unit may have more than one segment of local static storage. (For additional static storage, move some of the data to a COMMON block.)
- No program unit may have more than one segment of dynamic storage. (Make the excess static.)
- No data item in a COMMON block may be split across the boundary between two segments. Methods for complying with this rule are described under COMMON Statement in Chapter 3.

INTERFACE TO OTHER LANGUAGES

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

- All I/O routines must be written in the same language.
- There must be no conflict of data types for variables being passed as arguments. For example, an INTEGER in FORTRAN 77 should be declared as FIXED BINARY in PL/I. See Appendix D for

a description of F77 data storage formats. For detailed information on passing arguments from one language to another, see the Subroutines Reference Guide for Rev. 19 and higher.

- Modules compiled in 64V, 32I, or 32IX mode cannot reference or be referenced by modules compiled in any R mode. Modules in 64V or 32I may reference each other if they are otherwise compatible.

A few special restrictions apply when F77 and FTN modules reference each other. These are discussed in Appendix C.

F77 program units can also reference PMA (Prime Macro Assembler) routines, and vice versa. For information, see the Assembly Language Programmer's Guide.

F77 AND PRIME UTILITIES

Prime offers four major utility systems for use in your programming. These are:

- Data Base Management System (DBMS)
- Forms Management System (FORMS)
- Multiple Index Data Access System (MIDASPLUS)
- Prime's Recoverable Indexed Sequential Access Method (PRISAM)

For complete information on any of these utilities, see the appropriate reference guide. Following is a brief description of FORMS, MIDASPLUS, and PRISAM.

Forms Management System (FORMS)

The Prime Forms Management System (FORMS) provides a convenient method of defining a form in a language specifically designed for such a purpose. These forms may then be implemented by any applications program that uses Prime's Input/Output Control System (IOCS), including programs written in F77. Applications programs communicate with FORMS through input/output statements native to the host language. Programs that currently run in an interactive mode can easily be converted to use FORMS.

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

The interface of F77 with FORMS is identical to that of FTN. For more information see the FED User's Guide (IDR4940).

Multiple Index Data Access System (MIDASPLUS)

MIDASPLUS is a system of interactive utilities and high-level subroutines enabling the use of index-sequential and direct-access data files at the applications level. Handling of indexes, keys, pointers, and the rest of the file infrastructure is performed automatically for you by MIDASPLUS. Major advantages of MIDASPLUS are:

- Construction of large data files
- 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

The interface of F77 with MIDASPLUS is identical to that of FTN.

See the MIDAS User's Guide (IDR4558) and the MIDASPLUS PRIME TECHNICAL UPDATE (PTU98).

Prime's Recoverable Indexed Sequential Access Method (PRISAM)

PRISAM is a data management software system designed to provide solutions to users who require automatic recovery, simple file structures and strong performance in a transactional multiuser environment. Major features of PRISAM are:

- Manages sequential, indexed, and relative files
- Supports user defined and mixed transactions
- Provides for recovery from system halts
- Provides for media failure recovery

- Provides for software error recovery
- Allows up to 24 keys per file
- Permits concatenated keys

For more information, see the PRISAM User's Guide (DOC7999-2LA).

THE CONDITION-HANDLING MECHANISM

When an error occurs during execution of a program, PRIMOS responds by raising a condition. For each type of error, a corresponding condition exists.

When a condition is raised, PRIMOS activates the condition-handling mechanism. The condition handler notes what condition exists, then calls an error-handling routine known as an "on-unit" to deal with the error that has occurred.

PRIMOS supplies a default on-unit for each condition. You can specify your own response to a condition by supplying an on-unit of your own. When a condition occurs for which an on-unit exists that you have supplied, the actions specified in the on-unit will be taken, rather than those specified in the PRIMOS default on-unit.

Information on the system default on-units and the method for substituting your own on-units is contained in the Prime User's Guide. For complete information on the condition handler, see the Subroutines Reference Guide.

2

FORTRAN 77 Terms and Concepts

DEFINITIONS

Throughout this guide, several terms are frequently used. You should be aware of their exact meanings if discussions using them are to be understood correctly. You should also be familiar with the text conventions explained in ABOUT THIS BOOK. These terms are:

Term	Definition
Actual Argument	A data item passed to a subprogram. Actual arguments appear in the argument list of a subroutine CALL statement or a function reference.
Arithmetic Expression	Any expression which evaluates to type INTEGER, REAL, DOUBLE PRECISION, REAL*16, or COMPLEX.
Character Expression	A single item of type CHARACTER or the concatenation of any number of such items. Substrings and references to CHARACTER functions are permitted. Trailing blanks are of no significance in a character expression.

Dummy Argument	A variable or array name appearing in the header statement or an ENTRY statement of a subprogram. When the subprogram is invoked, each dummy argument is associated with the actual argument whose name appears in the corresponding position in the CALL statement or function reference.
Fixed-Length Character Expression	A character expression in which no operand is a dummy argument with an adjustable(*) length specification.
Integer Expression	Any expression which evaluates to type INTEGER, either directly or after type conversion via the functions INTS, INTL, or INT.
Integer Constant Expression	Any expression consisting only of integer constants and named integer constants with arithmetic operators and parentheses.
Program Unit	A main program, external function, subroutine, or block data unit.
Segment	A 128K-byte block of address space.
Subprogram	Any program unit except a main program.

FORTRAN 77 CHARACTER SET

As of PRIMOS Revision 21.0, FORTRAN 77 character strings may contain any character in the Prime Extended Character Set (Prime ECS), which contains the ASCII-7 character set as a proper subset.

In FORTRAN 77 program source statements, the valid characters are a subset of Prime ECS, as follows:

- The 26 uppercase 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 26 lowercase letters (F77 Extension):
  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

- These 13 special characters:

  = equals
  ' single quote (apostrophe)
  : colon
  + plus
  - minus
  * asterisk
  / slash
  (left parenthesis
) right parenthesis
  , comma
  . decimal point
  $ dollar sign
  _ underscore (F77 extension. Backarrow on some terminals.)

- Blanks or spaces

For the complete list of the Prime Extended Character Set that F77 supports, refer to Appendix A.

Note

ED, the Editor, interprets the backslash (\) as a logical tab. If you wish to make use of the Prime ECS backslash character in a file you are editing with ED, you must define another character as your logical tab.

Blanks in character and Hollerith constants and in $INSERT statements are treated as character positions. Elsewhere in FORTRAN 77 source text, blanks have no meaning and can be used as desired to improve program legibility. Lowercase letters are mapped to uppercase (except within Hollerith and CHARACTER constants) unless the program is compiled with the -LCASE option. Keywords must be in uppercase if -LCASE is specified. For the collating sequence, see Appendix A.

LINE FORMAT

Each program line is a string of 1 to 72 characters. Each character position in the line is called a column. Columns are numbered from left to right starting with 1. There are three types of lines:

- Comments
- FORTRAN 77 statements (and their continuations)
- Insert statements

In all line types, columns 73-80 are available for line order sequence numbers or other identification. (Usage is optional.) These columns, like comments, are ignored by the compiler, but are printed in the

program listing.

Comments

Comment lines are identified by the letter "C" or an asterisk in column 1. The remainder of the line may contain anything. A comment in columns 2 through 5 will cause the compiler to issue an error message. If you place a "C" or an asterisk in column 6, the compiler will interpret this as a continuation character. A comment line is ignored by the compiler, except that it is printed in the source listing. A comment line is not a statement. It provides a mechanism for you or someone else to better understand what your program is attempting to accomplish.

Starting from column 7 onward, use the following format to place a comment in your program:

```
/* comment */
```

The end of the line terminates the comment and makes the */ unnecessary. A comment within a character string will be treated as part of the character string.

Statements

In the first line of a statement, columns 1-5 are reserved for the statement label. Any statement may have a label between 1 and 99999 affixed to it. 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, to make the program easier to read.

In the continuation of a statement, columns 1-5 must be blank, column 6 may be any character except 0 or a blank, and the statement continuation is in columns 7-72. There may be at most 19 continuation lines.

Inserts

F77 allows files to be included automatically into the compilation stream, via the insert statement. An insert statement consists of the keyword $INSERT beginning in column 1, followed by the pathname of the file to be inserted. See $INSERT Statement in Chapter 3.

DATA TYPES

With the exception of the CHARACTER data type, a new feature of FORTRAN 77, the FORTRAN 66 and FORTRAN 77 data types are the same. Since you are expected to be familiar with some version of FORTRAN IV (extended FORTRAN 66), only highlights and Prime extensions of the FORTRAN 77 data types are discussed in the following sections. The CHARACTER and INTEGER data types are discussed in more detail. Each data type is illustrated with several constants of that type.

The seven major data types that exist in Prime F77 are:

- INTEGER
- REAL
- DOUBLE PRECISION
- REAL*16
- COMPLEX
- LOGICAL
- CHARACTER

Each of these may exist in any of four forms:

- Constant
- Parameter
- Variable
- Array

In addition, there are statement labels and Hollerith constants. Some subtypes exist, differing from each other only in storage size.

Table 2-1 lists the seven data types available in F77.

Table 2-1
F77 Data Types

Type	Bytes	Range
INTEGER	2 or 4	Same as for INTEGER*2 or INTEGER*4. (See Note.)
INTEGER*2 (short integer)	2	-(2**15) to (2**15-1) Decimal -32768 to 32767 Octal O'100000' to O'77777' Hexadecimal Z'8000' to Z'7FFF'
INTEGER*4 (long integer)	4	-(2**31) to (2**31-1) Decimal -2147483648 to 2147483647 Octal O'20000000000' to O'17777777777' Hexadecimal Z'80000000' to Z'7FFFFFFF'
REAL (REAL*4)	4	± (10**-38 to 10**38)
DOUBLE PRECISION (REAL*8)	8	± (10**-9824 to 10**9824)
REAL*16	16	± (10**-9824 to 10**9824)
COMPLEX (COMPLEX*8)	4+4	Each component has same range as REAL
COMPLEX*16	8+8	Each component has same range as DOUBLE PRECISION
LOGICAL	2 or 4	T or F
LOGICAL*4	4	T or F
LOGICAL*2	2	T or F
LOGICAL*1	1	T or F
CHARACTER	1 to 32767	1 to 32767 characters
Statement Label	2 or 4	1 to 99999
Hollerith	Varies	1 to 256 characters

INTEGER Data

An INTEGER data item represents an integer exactly. Integers are always written without a decimal point. An integer constant may be represented in decimal or octal form. (Octal form is an F77 extension.)

Here are some examples of INTEGER data items:

Decimal	Octal
-204	-:314 (same as :37777777464)
0	:0
8	:10
1911	:3567

F77 supports two integer subtypes: INTEGER*2 (short) and INTEGER*4 (long). If a variable is declared as an INTEGER with no *(size) specified, or takes on the type INTEGER by default, the variable will either be INTEGER*4 if the program is compiled with -INTL (the default), or INTEGER*2 if it is compiled with -INTS.

Integer constants compiled with the -INTL option (the default) will become INTEGER*4. With the use of the -INTS option, they become INTEGER*2 unless:

- Their magnitude lies outside the range -32768 to +32767 or is greater than :177777.
- Their representation, including leading zeroes, contains more than 5 decimal or 6 octal digits. For example:

```
30          short integer constant (under -INTS)
000030      long integer constant (always)
```

The following rules apply to the use of long and short integers within a program:

- They are interchangeable and may be mixed freely in expressions as long as short integers are not assigned values outside their range. A value outside of the range will result in an error message or indeterminate results.
- Integer arguments that are supplied to preexisting library and I/O routines must be of the type they expect. For example, if the library routine expects an INTEGER*2 argument, you must convert any long-integer arguments (the default), to short integers by the use of the INTS intrinsic function. If you want to convert all short integers in your program to long integers, use the -INTL option at compile time.

REAL Data

A REAL data item is an approximation to a real number. REAL data is always written with a decimal point, an exponent, or both. The decimal point is optional if an exponent is given. Blanks may appear between the mantissa and its exponent. Up to seven significant digits are retained. Exponents may range from -38 to +38. Here are some examples:

```
-204.      -20400 E-2      0.      8.8756E4      8.8756E+4
```

Real constants must fall in the type REAL range. They will not become DOUBLE PRECISION on the basis of magnitude or number of digits.

DOUBLE PRECISION Data

DOUBLE PRECISION data is also called REAL*8. It is similar to REAL except that twice as much storage is allocated, and "D" rather than "E" appears in the exponent. The "D" exponent is mandatory. Examples:

```
123456789.D0      2.5 D-2      0.D0      -999D+21
```

Up to 14 significant digits are retained. The exponent may range from -9812 to +9824.

REAL*16 Data

REAL*16 data is similar to DOUBLE PRECISION data except that up to 28 significant digits are retained. The letter "Q" is used as an exponent. Here are some examples:

```
12345Q03      2.72Q0      4.6Q-2      -1234.Q+01
```

The exponent may range from -9818 to +9824.

COMPLEX Data

A COMPLEX (or COMPLEX*8) data item is an ordered pair of real numbers. The first number represents the real part and the second represents the imaginary part. In a complex constant, or when a complex number is

used in list-directed I/O, the number appears in parentheses with its components separated by a comma. Here are some examples:

```
(1.,-1.)      (25E6, 331.)      (.172E19, 304E-2)
```

The comma and parentheses must appear when a complex number is used in list-directed I/O. They must be omitted from a complex number used in formatted I/O.

COMPLEX*16 Data

The COMPLEX*16 data type is identical to COMPLEX except that DOUBLE PRECISION numbers are used rather than REAL numbers.

LOGICAL Data

LOGICAL data items denote only the logical values TRUE and FALSE. In programs, logical constants must be written:

```
.TRUE.  .FALSE.
```

In input files, either the constants or the letters T and F may denote the values. On output, T and F are always written.

Logical constants and logical variables lacking a *(size) specification become either LOGICAL*4 if the program is compiled with -LOGL (the default), or LOGICAL*2 if it is compiled with -LOGS. A LOGICAL*1 type is also provided for compatibility with IBM FORTRAN. This type should not be used in new programs, because it is processed less quickly than LOGICAL*2 or LOGICAL*4.

CHARACTER Data

The CHARACTER data type is a new feature of FORTRAN 77. It makes Hollerith strings and the use of arithmetic variables to hold character data obsolete. F77 continues to support the Hollerith and arithmetic/character techniques as an aid to upward compatibility of existing programs. New programs should use only CHARACTER data.

A CHARACTER data item is a nonempty string of characters. Each item has a length equal to the number of characters it contains. The character positions are numbered from 1 to LENGTH. Each character occupies one byte.

A character constant consists of a string of characters enclosed in single quotes. Any internal single quotes must be represented by two consecutive single quotes. The two count as only one character position. For example, the character string:

```
'THAT''S ALL'
```

occupies ten positions, since the two quotes count as one for the 'S.

Hollerith Constants

Hollerith constants are accepted in F77 to aid upward compatibility of FORTRAN IV programs. This type is obsolete. Use CHARACTER constants and variables when writing new programs.

OPERANDS

Operands are those elements that are manipulated by the program. Four types of operands exist in FORTRAN 77: Constants, Parameters, Variables, and Arrays.

Constants

Constants exist for every data type. In a program, a constant appears as a literal representation of the desired value. The compiler determines the type of the constant from its appearance, its context, and the compiler options in effect.

The correct form for each type of constant appears in the previous subsection under the appropriate data type.

Parameters

Parameters are named constants, and may be of any data type. They are functionally similar to constants, but are referenced by the name assigned to the value in a PARAMETER statement, rather than by a literal occurrence of the value. Parameters may not appear in FORMAT statements. Parameter names follow the same rules as variable names.

Do not confuse parameters with arguments to subroutines. In FORTRAN 77 the term "parameter" denotes only a named constant.

Variables

Variables are data items whose values may be assigned, and subsequently altered, during program execution.

FORTRAN 77 variable names contain from one to six characters. In F77, variable names may have from one to 32 characters. Character 1 must be alphabetic; characters 2-32 (if any) must be alphanumeric, or the characters "$" or "_". You are discouraged from using "$" in your variable names because this character is used extensively in Prime-supplied software names, where it serves to implement a system of naming conventions.

When no type is explicitly declared, a variable whose name begins with the letters I through N becomes type INTEGER, and a variable whose name begins with A-H or O-Z becomes type REAL. See Chapter 3 for instructions on how to override this implicit convention, and how to specify DOUBLE PRECISION, COMPLEX, CHARACTER, and LOGICAL types.

Arrays

Arrays are ordered, multidimensional sets of variables. An array is declared in a DIMENSION, COMMON, or type-statement such as:

```
DIMENSION array_declarator [,array_declarator]...
```

where each "array declarator" has the form;

```
NAME (d1[,d2]...[,d7])
```

in which NAME is the name of an array (same rules as for a variable name), and each dn has the form:

```
[Ln:]Hn
```

Ln is the lower subscript bound, and Hn is the upper subscript bound, for dimension n. There may be at most seven dimensions. If Ln is omitted, it is assumed to be 1.

For example:

```
INTEGER ARR(-3:3,7,0:204,-207:-91,81)
DIMENSION A (2:4,4,-1:1)
COMMON C (-2:6,8)
```

In a main program, Ln and Hn must be integer-constant expressions. For a dummy argument array in a subprogram, they may be integer expressions (for an adjustable array), and the upper bound of the last dimension may be given an asterisk (to denote an assumed-size array). See Chapter 8 for details. Arrays are stored by columns: the leftmost subscript varies most rapidly when the array is accessed in storage order.

Referencing Arrays

Array references have the form:

```
NAME (S1[,S2]...[,S7])
```

where each Sn is a subscript expression.

A subscript expression is any legal FORTRAN 77 integer-valued expression. It may contain constants, variables, function references, intrinsic references, and other array references.

Notes

Non-integer data items are not allowed in subscript expressions. Convert any such items to integers by using the appropriate conversion function (IDINT, IFIX, INT, etc.)

An array longer than one segment (128K bytes) must be stored in a COMMON block. An array shorter than one segment should not be stored in a COMMON block longer than one segment. See Arrays as Arguments in Chapter 8 for more information. See the COMMON Statement in Chapter 3 for a restriction on the placement of data items (including arrays) in a COMMON block.

Evaluation of a function reference in a subscript expression must not alter any other elements of the subscript expression list, either directly or by altering arguments used in other function references.

Caution

When an array that crosses or may cross a segment boundary is passed as an argument to a subprogram, special action is necessary. See Arrays as Arguments in Chapter 8.

EXPRESSIONS

An expression is formed from one or more operands, operators, and parentheses. It evaluates to a single value. There are four kinds of expressions in F77:

- Arithmetic
- Relational
- Logical
- Character

Arithmetic Expressions

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces a numeric value. The expression can consist of constants, numeric variables, array elements, function references, or other expressions separated by parentheses and arithmetic operators. An arithmetic expression can be just one arithmetic term, or it can consist of more than one arithmetic term separated by operators. In FORTRAN 77, there are six arithmetic operators:

Operator	Representing
**	Exponentiation
/	Division
*	Multiplication
+	Addition
-	Subtraction or Negation
=	Assignment

Operator Evaluation: Arithmetic expressions are evaluated according to a particular operator hierarchy:

Operator	Rank
**	1
* and /	2
+ and -	3

When you have two or more operators of the same rank appearing in an expression, they are generally evaluated in a left-to-right order. For example, the expression A*B/C evaluates to (A*B)/C. However, the compiler takes advantage of groupings of elements (in accordance with mathematical rules) to optimize its output. In the case of A*B - A*C, the compiler may evaluate A*(B-C) instead. Consequently, evaluation may sometimes not be strictly left to right.

With exponentiation, the order of evaluation must be from right-to-left. For example, A**B**C is evaluated as A**(B**C).

The compiler always respects the integrity of parentheses. For example, (A*B) - (A*C) would be evaluated exactly as written. Expressions within parentheses are always evaluated before expressions outside them. For example, A*(B/C) will have its quotient evaluated first. Where evaluation order is critical, use parentheses to eliminate any ambiguity.

If you are combining numeric data types in an expression (mixed-mode arithmetic) the use of parentheses is suggested. For example, the expression = I*J*R is best evaluated as = I*(J*R) as opposed to = I*J*R or = R*I*J. An evaluation that proceeds in this manner may prevent an overflow condition during integer multiplication. An overflow condition happens when an integer or real value exceeds the upper limit allowed by the computer.

Where multiple references to functions occur in an expression, the compiler may evaluate them in any order. No function reference may alter any other value in the expression, either directly or by altering arguments used in other function references.

Character Expressions

A character expression contains a character string. It is a character constant, symbolic name of a character constant, character variable reference, character array element, character substring reference, or character function reference.

To join two or more strings to form one longer string, use the double slash as the concatenation operator:

```
character expression // character expression
```

Relational Expressions

A relational expression consists of two arithmetic or character expressions separated by one of six relational operators:

Relational Operator	Representing
.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

When evaluated, the value of a relational expression is either .TRUE. or .FALSE. For example, the expression 2 .LT. 3 evaluates to .TRUE. The arithmetic rules for operator precedence apply to relational expressions:

Operator	Rank
**	1
* and /	2
+ and -	3
//	4
.GT. .GE. .EQ. .NE. .LT. .LE	5

Logical Expressions

A logical expression uses logical operators to connect relational expressions. When tested, the logical exprssion will be either .TRUE. or .FALSE. Table 2-2 lists the logical operators.

The arithmetic rules for operator precedence apply to logical expressions:

Operator	Rank
**	1
* and /	2
+ and -	3
//	4
.GT. .GE. .EQ. .NE. .LT. .LE	5
.NOT.	6
.AND.	7
.OR.	8
.EQV. .NEQV.	9

Table 2-2
FORTRAN 77 Logical Operators

Operator	Meaning	Example (P and Q are of type LOGICAL)			Result
.NOT.	Logical Negation		.NOT.	Q	
		.TRUE.			.FALSE.
		.FALSE.			.TRUE.
.AND.	Logical Conjunction	Q	.AND.	P	
		.FALSE.		.FALSE.	.FALSE.
		.TRUE.		.FALSE.	.FALSE.
		.FALSE.		.TRUE.	.FALSE.
		.TRUE.		.TRUE.	.TRUE.
.OR	Logical Nonexclusive "ORing"	Q	.OR.	P	
		.FALSE.		.FALSE.	.FALSE.
		.TRUE.		.FALSE.	.TRUE.
		.FALSE.		.TRUE.	.TRUE.
		.TRUE.		.TRUE.	.TRUE.
.EQV.	Logical Equivalence	Q	.EQV.	P	
		.FALSE.		.FALSE.	.TRUE
		.TRUE.		.FALSE.	.FALSE.
		.FALSE.		.TRUE.	.FALSE.
		.TRUE.		.TRUE.	.TRUE.
.NEQV.	Logical Nonequivalence	Q	.NEQV.	P	
		.FALSE.		.FALSE.	.FALSE.
		.TRUE.		.FALSE.	.TRUE.
		.FALSE.		.TRUE.	.TRUE.
		.TRUE.		.TRUE.	.FALSE.

TYPE CONVERSION

Logical operators may combine logical operands of differing storage lengths, and arithmetic operators may combine operands of differing numeric types. The type of the result in such cases depends on the types of the operands.

Logical Conversion

The storage length of the result when logical data of differing lengths are combined is the longer of the two lengths. For example:

```
(LOGICAL*2 .AND. LOGICAL*4) is LOGICAL*4
```

Arithmetic Conversion

The type of the result when differing numeric types are combined will be that of the operand having the higher type in the following list which is ordered from highest to lowest:

```
COMPLEX*16
COMPLEX*8
REAL*16
DOUBLE PRECISION
REAL
INTEGER*4
INTEGER*2
```

For example, REAL + SHORT INTEGER is a REAL.

Special Case: To prevent loss of precision, the result-type when COMPLEX*8 and DOUBLE PRECISION data are combined will be COMPLEX*16.

Caution

When long integers are converted to reals, there may be a loss of precision. No error message will be generated, but incorrect results may occur.

PROGRAM ORGANIZATION IN FORTRAN 77

F77 program units consist of F77 statements. These statements must be arranged in the correct order. This section describes F77 statment order; Figure 2-1 summarizes the order.

Program Unit

A FORTRAN 77 program consists of one or more program units. A program unit is a sequence of statements that perform one or more operations. A program unit may be either a main program or a subprogram.

A program unit always has an END statement as its final statement.

Main Program

A main program is the program unit that receives control when an executable program is initiated. There is only one main program unit. The main program unit directs the flow of control to each subprogram (if any). Control returns to the main program unit after each subprogram has performed its computation.

The first statement in a main program unit is usually a PROGRAM statement. A main program unit does not use a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. A main program unit requires an END statement.

The main program unit is required for program execution.

Subprograms

A subprogram is a program unit that is invoked from the main program. The subprogram performs a computation on the behalf of the main program. There may be any number of subprograms associated with a main program.

Subprograms are introduced by FUNCTION, SUBROUTINE, or BLOCK DATA statements. The END statement is the final statement in a subprogram.

A subprogram is called (referenced) by a FORTRAN 77 statement in the main program or another subprogram.

Organization Considerations

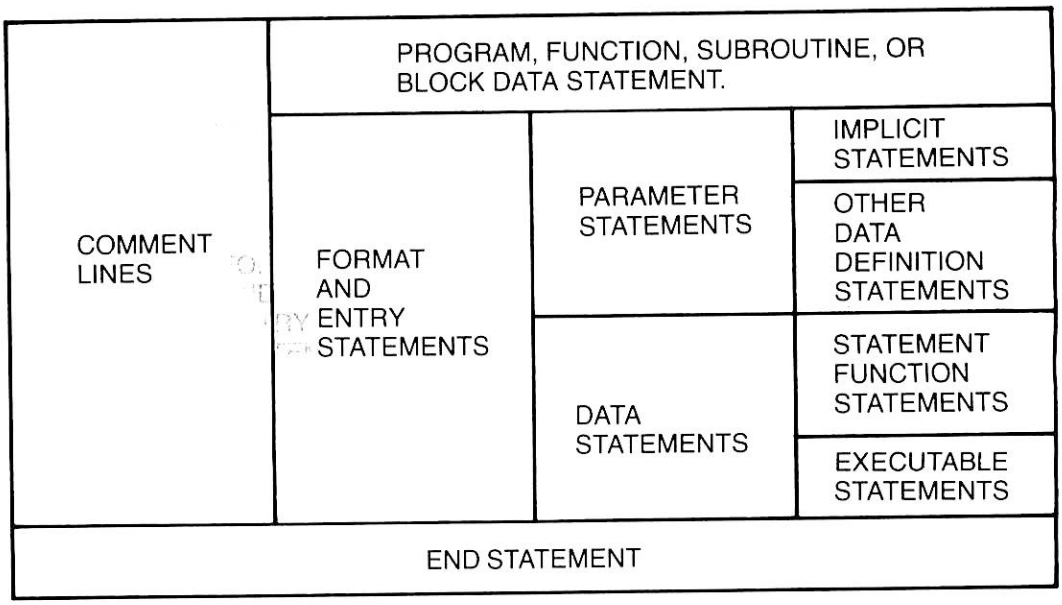
The following considerations apply to F77 program units:

- A main program unit or a subprogram unit may reference other program units that are contained in separate files.
- A file may contain any number of program units.
- Each program unit must be terminated by an END statement.
- Comments are the only statements that can appear between the END statement of one program unit and the header statement in the next program unit.

Size Considerations

In F77, no block of executable code can cross a segment boundary. Therefore, no program unit may produce more than 128K bytes (the size of a segment is 128K bytes) of code. A program unit will rarely be any larger than this. A program unit that is larger than a segment, must be broken up. Program data is kept in separate data segments, and hence does not compete for space with the executable code.

The names of F77 program units may not be more than 32 characters long. Additional characters will be ignored and a warning message printed.



Statement Order in F77
Figure 2-1

PART II

Prime F77 Language Reference

3
Specification Statements

Specification statements are nonexecutable statements that allow you to:

- name your main program (PROGRAM).
- override the language convention for default data (IMPLICIT).
- override implicit typing of symbolic names (TYPE statements).
- define array dimensions (DIMENSION).
- define common blocks (COMMON).
- allocate storage (EQUIVALENCE).
- initialize data (DATA).
- define symbolic names of constants (PARAMETER).
- pass subprograms as arguments to other subprograms (EXTERNAL).
- retain the value of local variables between subprogram invocations (SAVE).
- pass specified function names as arguments to subprograms (INTRINSIC).

- define beginning of block data subprogram (BLOCK DATA).
- perform self-labeling I/O (NAMELIST)

This chapter also discusses the F77 compiler control directives. These nonexecutable statements are extensions to the FORTRAN language. Table 3-1 gives a list of the specification statements and the syntax for each statement.

PROGRAM STATEMENT

The PROGRAM statement gives a name to a main program. It is not required. However, if you use the PROGRAM statement, it must be the first statement of the main program.

The PROGRAM statement has the following format:

```
PROGRAM name
```

where:

name is the symbolic name of the main program in which the PROGRAM statement appears. name must not duplicate the name of any COMMON block or subprogram, or of any data item in the main program.

IMPLICIT STATEMENT

The IMPLICIT statement allows you to override the language convention for default data typing by first letter.

FORTRAN 77 automatically assigns types to all variables, parameters, arrays, and functions that do not appear in type statements. The default types are as follows: if the symbol's first character begins with the letters I through N, the symbol is typed as integer; all other names beginning with letters A - H, or O - Z, are typed as real. The default integers are long integers (INTEGER*4) unless you use the -INTS option at compile time. (See Chapter 9 for information on the -INTS option.)

The IMPLICIT statment has the following format:

```
IMPLICIT type (list) [, type (list)]...
```

where:

type is one of INTEGER*2, INTEGER*4, REAL*4, DOUBLE PRECISION, REAL*16, COMPLEX*8, COMPLEX*16, LOGICAL, or CHARACTER.

list alphabetically lists the letters that will cause default to that type. Letters may be separated by a comma, or an inclusive group of letters may be indicated with a dash.

Symbols not typed in a type statement or by a default specified in an IMPLICIT statement will be typed by the FORTRAN 77 language default.

For example:

```
IMPLICIT DOUBLE PRECISION (A,N,O,P-Z), LOGICAL (B), CHARACTER*3 (M)
```

First letter of symbol	Type
A, or N through Z	Double Precision
B	Logical
C through H	Real
I through L	Integer
M	Character*3

If you use the IMPLICIT statement, it must be the first statement of a main program (the second statement if a PROGRAM statement exists), or the second statement of a subprogram. IMPLICIT affects all symbols not otherwise typed. This includes dummy arguments in the header statement of a subroutine or function, and function names that are not explicitly typed. IMPLICIT typing does not affect the default type of intrinsic functions.

TYPE STATEMENTS

The type statement is used to explicitly type symbolic names, superceding any implicit type assignments of symbol names done either by IMPLICIT or by language default. A data item may be initialized in a type statement. There are two kinds of type declaration statements:

- numeric type declarations
- character type declarations

The following rules apply to type statements:

- Within a program unit, a name must not have its type explicitly specified more than once.
- Type declaration statments must precede all executable statements.
- The name of a main program, subroutine, or block data subprogram must not appear in a type statement.

Numeric Type Declaration Statements

The numeric type statement has the following format:

```
type v [,v]
```

where:

type is replaced by one of the following data type specifications:

```
INTEGER
INTEGER*2
INTEGER*4 (same as INTEGER)

REAL
REAL*4 (same as REAL)
REAL*8 (same as DOUBLE PRECISION)
REAL*16

DOUBLE PRECISION (same as REAL*8)

COMPLEX
COMPLEX*8 (same as COMPLEX)
COMPLEX*16
```

LOGICAL
LOGICAL*1
LOGICAL*2
LOGICAL*4

v is a variable name, array name, array declarator, symbolic name of a constant, function name, or dummy procedure name.

Note

The types INTEGER*2, REAL*16, COMPLEX*16, LOGICAL*1, and LOGICAL*2 are F77 extensions. The names INTEGER*4, REAL*4, REAL*8, COMPLEX*8, and LOGICAL*4 are F77 synonyms for the corresponding FORTRAN 77 data types INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL. These synonyms are provided for upward compatibility of existing FORTRAN IV programs. They should not be used in new programs.

The storage length given in the type will ordinarily apply to all the data items in the statement. In F77, lengths may also be specified for data items singly. When both a single and a general length specification are given, the single specification takes precedence. For example:

Type Statement	Equivalent To
INTEGER A*4, B*2	INTEGER*4 (A), INTEGER*2 (B)
INTEGER*4 C, D*2, E	INTEGER*4 (C), INTEGER*2 (D) INTEGER*4 (E)

Recognition of synonymous data types is provided to ease conversion of existing programs to F77. INTEGER will normally default to INTEGER*4 (long integer) unless the program is compiled with the -INTS option, in which case it will default to INTEGER*2 (short integer). LOGICAL will default to LOGICAL*4 unless the program is compiled with -LOGS, in which case it will default to LOGICAL*2. See Chapter 9 for compiler option information.

To initialize a data item in a type statement, enclose the desired value between slashes and insert it immediately after the data item name. The rules and syntax of the DATA statement apply, except that each initializing value must follow its data item immediately, and not all the items need be initialized.

For example:

```
INTEGER A/5/,B,C,D,E(2)/1,2/,F(5)/5*10/
```

Any initialized data item will be placed in static storage.

Character Type Declaration Statements

The CHARACTER type statement has the following format:

```
CHARACTER [*len [,]] cname [,cname]...
```

where:

len is an integer constant, or an integer constant expression in parentheses. len must be equal to between 1 and 32767 inclusive, giving the length of the CHARACTER variable in bytes. If *len is omitted, the length defaults to 1. In a dummy argument in a subprogram, len may be replaced by an asterisk in parentheses. A character item so declared will take on the length of the corresponding actual argument in the invoking program unit.

cname is a variable name or a list of variable names, parameter names, array names, function names, or array declarators.

CHARACTER entities may be initialized in a type statement. CHARACTER entities having different lengths may be declared in the same type statement. For example:

Type Statement	Equivalent To
CHARACTER*50 F, G*100, H, J*1	CHARACTER*50 (F), CHARACTER*100 (G) CHARACTER*50 (H), CHARACTER*1 (J)

CHARACTER parameters and arrays are declared as with other data types.

Substrings: A contiguous subset of a CHARACTER data item is known as a substring. A substring of a variable or array element is specified in the following ways:

```
VARNAME(L:H)      ARRAYNAME(subscripts)(L:H)
```

```
VARNAME(L:H)      ARRAYNAME(subscripts)(L:H)
```

where:

L and H are integer expressions giving the lowest and highest character positions of the desired substring. If L is omitted, 1 is assumed. If H is omitted, the length of the variable is assumed. L must be less than or equal to H.

Substrings cannot be extracted from constants and parameters. When a substring of a constant or parameter is needed, assign the constant or parameter to a CHARACTER variable, then extract the substring from the variable.

For example, if CVAR = 'ABCDE'. Then:

```
CVAR (2:5) is equivalent to:  'BCDE'
CVAR (:3) is equivalent to:   'ABC'
CVAR (4:) is equivalent to:   'DE'
```

Use of Octal Constants: Since the ANSI FORTRAN character substring operation and Prime's convention for octal constants both use the colon character, some expressions involving colons are ambiguous. Therefore, if a program unit with a CHARACTER or IMPLICIT CHARACTER statement wishes to use octal constants anywhere in a function argument list, you must specify that function in an INTRINSIC or EXTERNAL statement.

Concatenation: Character entities may be concatenated using the operator '//'. Here are some examples:

```
'Z' // CVAR (2:5) is equivalent to:  'ZBCDE'
'ABC' // 'XYZ' is equivalent to:  'ABCXYZ'
```

Assignment: Where lengths do not match in an assignment of character data, truncation or padding with blanks takes place on the right. Undefined positions on either side of positions assigned by substring remain undefined.

In FORTRAN 77, no position may act as both source and destination in a substring assignment. F77 relaxes this restriction. This extension must be used carefully, because the source string is not copied before execution of a substring assignment. The assignment may therefore encounter its own effects partway through execution.

For example, if K and Q are CHARACTER*5:

```
K = 'A' // 'B' // 'C'          /* K = 'ABCbb'
Q (3:4) = K (2:3)              /* Q = '??BC?'
K = K // K                     /* K = 'ABCbb'
K (1:3) = K (2:4)              /* K = 'BCbbb'
```

Comparison: Character entities may be compared using the relational operators. The collating sequence reflects the Prime Extended Character Set. (See Appendix A.)

```
IF ('ABX' .LT.  (CVAR (2:3)//'ZQ')) GO TO 100
```

Intrinsic Functions: Various intrinsic functions exist to provide services related to CHARACTER data items. They are described in Chapter 8.

Input/Output: I/O of CHARACTER data is similar to I/O for the other data types. Formatted CHARACTER I/O uses the "A" field descriptor. See Chapter 7.

DIMENSION STATEMENT

The DIMENSION statement defines a symbolic name to be an array, and sets the number of dimensions and bounds of each dimension of the array.

The DIMENSION statement has the following format:

```
DIMENSION array declarator [,array declarator]...
```

where:

array declarator consists of the array name followed by parentheses that enclose the maximum values for each dimension of the array. See Chapter 2 for information on array declarators.

The following DIMENSION statement shows how to declare the dimensions of three arrays:

```
DIMENSION JARRAY(10), KARRAY(2,3), LARRAY(5,6,7)
```

The first array has a maximum dimension of 10; the second, a value of 2 X 3; and the third array, a value of 5 X 6 X 7.

Arrays can also be declared in a type statement:

```
INTEGER JARRAY(10), KARRAY(2,3), LARRAY(5,6,7)
```

Table 3-1
Specification Statement and
Compilation Directive Syntax

Statement	Syntax
BLOCK DATA	BLOCK DATA [name]
COMMON	COMMON [/[cb/] nlist [[,]/[cb]/ nlist] ...
DATA	DATA nlist/clist/ [[,] nlist/clist] ...
DIMENSION	DIMENSION array declarator [,array declarator]...
EQUIVALENCE	EQUIVALENCE (nlist) [,(nlist)] ...
EXTERNAL	EXTERNAL name [,name]
FULL LIST	FULL LIST
IMPLICIT	IMPLICIT type (list) [,type (list)]...
$INSERT	$INSERT insert-file (Must start in Col. 1)
INTRINSIC	INTRINSIC name [,name]...
LIST	LIST
NAMELIST	NAMELIST/blockname/v[,v].../
NO LIST	NO LIST
PARAMETER	PARAMETER (p=c [,p=c]...)
PROGRAM	PROGRAM name
SAVE	SAVE [v [,v]...]
Type statement	type v [,v]...

COMMON STATEMENT

The COMMON statement is a means of communicating between program units through a common storage area that can be referenced by two or more program units.

The COMMON statement has the following format:

```
COMMON [/[cb]/] nlist [[,]/[cb]/ nlist] ...
```

where:

cb is a common block name. If you include cb, this is a named common block. If you do not include cb, this is a blank common block. If you do not include the first cb, the first two slashes are optional. cb must not be the same as the name of any subroutine, function, or entrypoint in the program.

nlist is a list of variable names, array names, and array declarators. You cannot use the same name more than once within nlist, nor can you use the names of dummy arguments.

Data items are assigned sequentially within a COMMON block in the order of appearance in the COMMON statement(s) defining the block. BIND assigns all COMMON blocks with the same name to the same storage area, regardless of the program or subprogram in which they are defined.

The length of a COMMON block is the number of bytes used by all the items specified in the COMMON statement(s), plus the number of bytes appended to the block by any EQUIVALENCE statements.

Blank COMMON blocks may be of differing lengths. In FORTRAN 77, all instances of a named COMMON block must have the same length. This restriction is relaxed in F77, as an aid to compatibility with other extended versions of FORTRAN 77.

An F77 extension also allows both character and noncharacter data to be allocated in the same COMMON block.

When a given COMMON block, named or blank, has different lengths in different program units, the program unit containing the longest instance of the block must always be loaded first, because BIND allocates space for a COMMON block on the basis of its first occurrence. Note that a set of program units with COMMON blocks could easily be generated for which no correct load order exists. The preferred method is simply to make all instances of a COMMON block the same length, by padding them as necessary. No inefficiency of time or space utilization can result from following this practice.

These restrictions exist on the layout of data items in a COMMON block.

- In any COMMON block, all data items except CHARACTER and LOGICAL*1 variables and array elements must begin at a 16-bit halfword boundary (0, 2, 4... bytes from storage location 0). Use padding variables as needed to maintain word alignment.

- Every CHARACTER variable or array in a COMMON block more than one segment (128K bytes or characters) in size must have an element length that is a power of two.

- Every variable and array of any kind in a large COMMON block must be offset by a multiple of its element length from the start of the COMMON block. No single element may be larger than one segment. The length of every variable and array element must divide evenly into the length of a segment.

Note

When an array that spans a segment is passed to a subprogram as an actual argument, that subprogram must be compiled with -BIG if that particular array spans the segment boundary in a large COMMON block. (A large COMMON block is anything over one segment in size.) See Arrays as Arguments in Chapter 8.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement specifies that two or more entities within the same program unit share storage locations.

The EQUIVALENCE statement has the following format:

```
EQUIVALENCE (nlist) [,(nlist)]...
```

where:

nlist is a list of two or more variable names, array element names, and character substring names. Each of the entities in nlist are allocated in memory beginning at the same location. When an unsubscripted array name is mentioned, the effect is as if its first element had been mentioned. FORTRAN 77 requires a separate subscript for each dimension of an array, but Prime F77 allows one subscript to be used to denote the whole array.

An EQUIVALENCE statement causes all the items mentioned in each parenthetical list to be stored beginning with the same byte of physical storage. When variables of different lengths are equivalenced, the shorter is stored in the first bytes of the longer. When specific array elements are equivalenced, the arrays as wholes become correspondingly aligned.

When data in a COMMON block is equivalenced to other data, some bytes of the other data may become aligned with storage positions outside of the COMMON block. When this occurs, the block has been extended. Only extensions to the right (towards higher storage addresses) are legal.

Legal example:

```
INTEGER  I, A(3)
COMMON  // I
EQUIVALENCE  (I, A(1))
```

This example extends the COMMON block to the right (towards higher storage addresses).

Illegal example:

```
INTEGER  I, A(3)
COMMON  // I
EQUIVALENCE  (I, A(3))
```

This example attempts to extend the COMMON block to the left. This is illegal and will cause an error message.

Data items already fixed in storage cannot be equivalenced. An equivalence statement cannot make self-contradictory demands. Therefore the following examples are all illegal:

```
INTEGER  A(5)
EQUIVALENCE  (A(1), A(5))

COMMON  // A,B
EQUIVALENCE  (A, B)

INTEGER  A(5), B(5), C(5)
EQUIVALENCE  (A(5), B(1)), (B(5), C(1)), (C(5), A(1))
```

Prime's hardware requires that all COMMON block data items except CHARACTER and LOGICAL*1 variables and array elements must begin at a 16-bit halfword boundary (0, 2, 4...bytes from the start of the COMMON block). No EQUIVALENCE can violate this rule. Hence the following is illegal:

```
CHARACTER*1  CVAR(4)
INTEGER*4  NUM
COMMON // CVAR
EQUIVALENCE  (CVAR(2),NUM)
```

Any data item equivalenced to a static data item will itself be static. In F77, character and non-character data may be equivalenced. FORTRAN 77 does not allow this practice.

DATA STATEMENT

The DATA statement tells the compiler to put initial values into data items before program execution.

The DATA statement has the following format:

```
DATA nlist/clist/ [[,] nlist/clist/] ...
```

where:

nlist is a list of variables, array names, array elements, substring names, and implied DO lists, in which any expressions that appear must be integer constant expressions.

clist is a list of constants and parameters, possibly with repetition factors. A repetition factor is an integer constant followed by an asterisk.

The values in each clist are assigned in order to the corresponding items in nlist. For each item, there must be a value of a type validly assignable to the item. If a scalar numeric object in nlist is being assigned a character value from clist, type conversion and character padding will occur as they would in an assignment statement, but a compiler error will be generated if truncation is necessary. A type declaration of CHARACTER cannot be initialized to a numeric value in a DATA statement. Any implied DO lists and repetition factors present operate as they would in a list-directed READ statement. For example:

```
INTEGER A,K, ARR(1:5, 1:5)
DATA A,K/3,4/((ARR(I,J), I=1,5), J=1,5)/25*5.0/
```

When large arrays of character data must be initialized, effort can be saved by declaring a separate CHARACTER variable equal in length to the entire array, equivalencing it to the array, and initializing it with the concatenation of all the desired initial values. For example:

```
CHARACTER*2 K(3)
CHARACTER*6 INITK
EQUIVALENCE (K,INITK)
DATA INITK /'ABCDE'/
```

In the above example, array K now contains the following values:

```
K(1)='AB' K(2)='CD' K(3)='E '
```

A string can be used to initialize an entire array, provided that the string size is less than or equal to the size of the array elements.

Overflow of string size causes an error message. The following program segment generates an error message due to overflow of string size:

```
                ------
                ------
6               REAL *4 RADIO, RID,DIO
7               DATA RADIO/'CAD'/, RID/'OBEY'/, DIO/'ABSOLUTELY'/
                ------
                ------
```

Here is the error message generated by the above example:

```
ERROR 398 SEVERITY 4 BEGINNING ON LINE 7
A string of incorrect length is
being used to initialize "DIO."

MAX SEVERITY IS 4
```

In the example above, DIO is declared to be 4 bytes. However, the string ABSOLUTELY is larger than 4 bytes. The size of the strings CAD and OBEY are not larger than the declared size of the variables RID and RADIO, and are therefore acceptable.

Any data item initialized in a DATA statement, and any data items equivalenced to it, will be declared static by the compiler. For more information, see the SAVE statement discussed later in this chapter, and the -SAVE option in Chapter 2.

In F77 you can initialize named or blank common items outside of a block data subprogram, using the DATA statement.

F77 also allows variables or symbolic names to be initialized in type statements.

PARAMETER STATEMENT

The PARAMETER statement allows you to reference a constant by using a symbolic name.

The PARAMETER statement has the following format:

```
PARAMETER (p=c [,p=c]...)
```

where:

p is a symbolic name previously typed in any standard way.

c is a constant expression of a type appropriate to the corresponding p. A constant expression consists only of constants, parameters, constant expressions in parentheses, and appropriate operators.

For example:

```
PARAMETER (TEST = 6.8679)
        .
        .
        .
ANS = TEST/B
```

Any parameters that appear must have been defined in a previous PARAMETER statement. Function references and nonintegral exponentiations are prohibited. A parameter may not be used to form a complex constant.

Unless specifically prohibited, parameter names may be used wherever a constant could be used (including DATA and DIMENSION statements) except in FORMAT statements. Since parameters are named constants, they may not be elements of COMMON blocks and cannot be equivalenced. They may be used in declaring bounds of arrays in COMMON.

In F77, the parentheses around the parameter list may be omitted.

EXTERNAL STATEMENT

The EXTERNAL statement allows you to specify subprograms to be passed as arguments to other subprograms, where they may be used directly, or declared EXTERNAL and passed again.

The EXTERNAL statement has the following format:

```
EXTERNAL name [,name]...
```

where:

name is the name of a user-supplied or library subprogram, or is a dummy subprogram name.

Without the EXTERNAL statement, variables would be default-declared and passed instead.

If you specify an intrinsic function name as EXTERNAL in a program unit, the name will refer to the user-supplied subprogram, and the intrinsic function will be unavailable to that program unit. If you want to pass an intrinsic function to a subprogram, you should use the INTRINSIC statement that is discussd in the following sections.

It is recommended that the names of any user-supplied subprograms called from a program unit appear in an EXTERNAL statement in that unit. This method enhances portability to other systems, where some intrinsic function might have the same name as a user-supplied subprogram.

SHORTCALL STATEMENT

The SHORTCALL statement specifies that one or more external subprograms are to be called using the Prime Shortcall Interface. The external subprograms named in the SHORTCALL statement must be written in Prime Macro Assembly Language (PMA), and must conform to the Prime Shortcall Interface. The F77 calling program and the PMA called program must be both I-mode programs or both V-mode programs. Refer to the Assembly Language Programmer's Guide for a complete discussion of this interface and its implications for the PMA programmer. Refer to Appendix E of this manual for an example of V-mode and I-mode SHORTCALL routines.

The advantage of shortcalled subprograms is that the mechanisms of invoking them and passing arguments to them are much more efficient than the Prime procedure call mechanism by which all F77 subprograms are normally called. Small F77 subprograms that are frequently invoked during the execution of an F77 application are excellent candidates for conversion to shortcalled PMA subprograms.

An external shortcalled subprogram is invoked the same way any F77 subprogram is invoked, using either a function reference or a CALL statement.

The SHORTCALL statement has the form:

```
SHORTCALL subprogram[(n)] [,subprogram[(n)]]
```

where:

subprogram is the name of an external subprogram that is called by the Prime Shortcall Interface.

n is an integer constant expression indicating the number of halfwords of scratch space reserved in addition to the default. F77 automatically reserves 20 halfwords in every program unit's stack frame header for use by shortcalled subprograms; if you specify a value for n, F77 reserves 20 + n halfwords. F77 reserves additional space by increasing the size of the calling program's stack frame header by the amount specified by n.

Notes

Subprograms whose names appears in SHORTCALL statements may not be passed as arguments to other subprograms.

The SHORTCALL feature should be used with caution and should be used only by programmers who are thoroughly familiar with PMA and with the Prime Shortcall Interface.

SAVE STATEMENT

The SAVE statement causes the subprogram variables and arrays named in it to retain their values between invocations (static storage) rather than losing their values when the subprogram returns (dynamic storage).

The SAVE statement has the following format:

```
SAVE [v [,v]...]
```

where:

v is a variable or array name or a COMMON block that is not part of or equivalenced to a COMMON block. If COMMON block is used the format is: /COMMON block/. If no vs appear, the save is taken to include all local data items.

In F77, all COMMON blocks, named or blank, are static in all cases. Therefore the appearance of a COMMON block name in a SAVE statement has no effect. If a program is compiled with the -SAVE option, all local data items will be static. A SAVE statement will not have any effect. If a program is compiled with -DYNM, (the default) all local data items will be dynamic unless they are saved.

INTRINSIC STATEMENT

The INTRINSIC statement allows the function names specified to be passed as arguments to other subprograms, that may then reference the particular function passed.

The INTRINSIC statement has the following format:

```
INTRINSIC name [,name]...
```

where:

name is the name of an F77 intrinsic (built-in) function.

Without the INTRINSIC statement, variables would be default-declared and passed instead. No name may appear in both an INTRINSIC and an EXTERNAL statement, or in more than one INTRINSIC statement, in the same program unit.

It is recommended that the names of all intrinsic functions referenced in a program unit be listed in an INTRINSIC statement in that unit. This practice will result in immediate diagnostic messages if the program is run on a different system that does not supply all the needed intrinsics.

BLOCK DATA STATEMENT

The BLOCK DATA statement is the first statement in a BLOCK DATA subprogram. A BLOCK DATA subprogram is a nonexecutable subprogram that initializes variables in named COMMON. You have the option of not providing a name for a BLOCK DATA subprogram; however, if you do provide a name, it must not be the same as any other name used by any other block data subprogram. A program may contain any number of BLOCK DATA subprograms.

The BLOCK DATA statement has the following format:

```
BLOCK DATA [name]
```

where:

name is the name of the block data subprogram in which the BLOCK DATA statement appears.

Initialization of blank COMMON is an F77 extension. The entire block must be specified in a COMMON statement in the subprogram if any part of it is to be initialized. Only COMMON, EQUIVALENCE, DIMENSION, DATA, IMPLICIT, PARAMETER, END, INCLUDE, and type statements may appear in a BLOCK DATA subprogram. The END statement is the last statement in the subprogram.

Notes

The BLOCK DATA statement must appear only as the first statement of a BLOCK DATA subprogram.

All entities in a named COMMON must be specified.

The named COMMON block may be used only once in any subprogram.

INCLUDE STATEMENT

The INCLUDE statement includes into the compilation stream at that point the contents of a file whose pathname is "insert-file". This is useful in avoiding the duplication of code which is required several times in a program or a series of programs. For example, there may be several lines of source code, such as common block specification, that appear in several program units.

The INCLUDE statement is also useful in that it allows multiple users to access common files. The INCLUDE statement must begin in column 7, or thereafter, of the source form.

The INCLUDE statement has the following format:

```
INCLUDE 'insert-file'
```

Here is an example of the INCLUDE Statement:

```
INCLUDE 'CIRCLE'
CALL SIRKLE(RADIUS,AREA)
PRINT*, 'THE AREA OF THE CIRCLE IS:', AREA
STOP
END

SUBROUTINE SIRKLE(RD,ANS)
PI = 3.145929
ANS = PI*RD*2
RETURN
END
```

In the above example, if the INCLUDE file 'CIRCLE' contained the following lines of code:

```
AREA = 0
RADIUS = 0
PRINT*, 'TYPE IN A VALUE FOR THE RADIUS OF THE CIRCLE:'
READ*, RADIUS
```

then, the INCLUDE statement includes those lines of code at the point indicated.

The program compiles as if it were the following:

```
AREA = 0
RADIUS = 0
PRINT*, 'TYPE IN A VALUE FOR THE RADIUS OF THE CIRCLE:'
READ*, RADIUS
CALL SIRKLE(RADIUS,AREA)
PRINT*, 'THE AREA OF THE CIRCLE IS:', AREA
STOP
END

SUBROUTINE SIRKLE(RD,ANS)
PI = 3.145929
ANS = PI*RD*2
RETURN
END
```

The example above demonstrates that the code used in the file 'CIRCLE' need not be repeated when the INCLUDE statement is used.

An INCLUDE statement may appear nested inside a file named in an INCLUDE statement. INCLUDE files may be nested up to 32 levels.

Up to 500 INCLUDE statements may appear in a compilation, including those found inside a file named in an INCLUDE statement.

Note

The coded lines of the inserted file must be comply with the required order for F77 statements as summarized in Figure 2-1.

The pathname of the inserted file must be enclosed by single quotes.

At Revision 21.0, F77 supports using INCLUDE$ Search Rules with the INCLUDE statement. Refer to Appendix G for detailed information about how to use the search rules with the INCLUDE statement.

NAMELIST STATEMENT

Namelist is a convenient method for performing self-labeling input/output through the use of READ and WRITE statements. The basic unit of namelist I/O is the namelist block. A namelist block is a group of variables that namelist treats as unit. Any variable that is to be read or written using namelist must belong to a namelist block.

A namelist block is established using a NAMELIST statement. The NAMELIST statement has the following format:

```
NAMELIST /name/ variable [,variable]...
```

where:

name is the symbolic name of the namelist block.

variables are the data items making up the block. 247 data items are allowed.

Here is an example of the NAMELIST statement:

```
NAMELIST /SHIP/ I,K,SPEED
```

As you can see, the variables are typed in any standard way. A namelist variable may also be part of a COMMON block, and may belong to more than one namelist block. There may be any number of namelist blocks. F77 allows up to 247 entries to be present in a NAMELIST block. No subscripts may appear in a NAMELIST statement.

Namelist names and keywords in lowercase are automatically converted to uppercase at runtime only.

For more information on using the namelist-directed I/O statement, see Chapter 6.

COMPILER CONTROL DIRECTIVES

The following statements are F77 extensions. They provide a means of controlling source-listing generation from within a program, and of directing the compiler to insert files into the source program.

NO LIST Statement

The format of the NO LIST statement is

```
NO LIST
```

If a source listing of any kind has been specified in the compiler options, encounter of a NO LIST statement will suppress generation of the listing for source lines following the statement.

If no source listing has been specified, NO LIST has no effect.

LIST Statement

The format of the LIST statement is

```
LIST
```

The LIST statement reverses the effect of a NO LIST statement. Source-listing generation resumes (or begins) following the LIST statement.

A LIST statement will not of itself cause source listing to be generated. An appropriate compiler option must have been given. If one was not, LIST has no effect.

FULL LIST Statement

The format of the FULL LIST statement is

```
FULL LIST
```

This statement is an obsolete equivalent to LIST. It is supported for compatibility with FTN, and should not be used in new programs. See Appendix C for a discussion of FTN/F77 compatibility.

$INSERT Statement

The $INSERT statement inserts the contents of a file whose pathname is 'insert-file' into the compilation stream at the point where the $INSERT statement is located. $INSERT statements can be nested up to a 32 level depth.

$INSERT is commonly used for:

- Insertion of COMMON specifications into programs
- Frequently used statement functions
- Data initialization statements
- Numeric key definitions, especially for the file management system, applications library, MIDAS, MIDAS PLUS, PRISAM, and so on.

The $INSERT statement has the following format:

```
$INSERT 'insert-file'
```

Unlike other statements, the $INSERT directive must begin in column 1.

F77 allows up to 500 $INSERT files to be included in one source file.

At Revision 21.0, F77 supports using search rules with the $INSERT statement. Refer to Appendix G for detailed information about using the Search Rules with the $INSERT statement.

Note

The coded lines of the inserted file must be comply with the required order for F77 statements as summarized in Figure 2-1.

Here is an example of the $INSERT Statement:

```
$INSERT 'CIRCLE'
      CALL SIRKLE(RADIUS,AREA)
      PRINT*, 'THE AREA OF THE CIRCLE IS:', AREA
      STOP
      END

      SUBROUTINE SIRKLE(RD,ANS)
      PI = 3.145929
      ANS = PI*RD*2
      RETURN
      END
```

In the above example, if the $INSERT file 'CIRCLE' contained the following lines of code:

```
      AREA = 0
      RADIUS = 0
      PRINT*, 'TYPE IN A VALUE FOR THE RADIUS OF THE CIRCLE:'
      READ*, RADIUS
```

then the $INSERT statement includes those lines of code at the point indicated.

The program compiles as if it were the following:

```
      AREA = 0
      RADIUS = 0
      PRINT*, 'TYPE IN A VALUE FOR THE RADIUS OF THE CIRCLE:'
      READ*, RADIUS
      CALL SIRKLE(RADIUS,AREA)
      PRINT*, 'THE AREA OF THE CIRCLE IS:', AREA
      STOP
      END

      SUBROUTINE SIRKLE(RD,ANS)
      PI = 3.145929
      ANS = PI*RD*2
      RETURN
      END
```

The example above demonstrates that the code used in the file 'CIRCLE' need not be repeated when the $INSERT statement is used.

4

Assignment Statements

This chapter discusses the use of assignment statements to perform the following tasks in your program:

- To compute and store calculations.
- To assign a constant to a storage location.
- To copy the contents of one storage location to another.
- To assign statement labels to integer variables.

An assignment statement is an executable statement that specifies an expression whose value is to be computed and assigned to a variable to the left of the equals (=) sign. The direction of evaluation of an assignment statement is always from the right of the equals sign to the left.

There are four kinds of assignment statements:

- Arithmetic
- Character
- Logical
- Statement label (ASSIGN statement)

The target of the assignment statement sign must always be a predefined variable or array element name.

Note

The variable or array element receiving an arithmetic value can be type INTEGER*2, INTEGER*4, REAL, DOUBLE PRECISION, REAL*16, COMPLEX, or COMPLEX*16. The variable or array element receiving a logical value must be type LOGICAL. The variable, element, or substring receiving a character value should be type CHARACTER.

ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement has the following format:

```
target = arith-expr
```

where:

target is the name of a variable or array element of type INTEGER*2, INTEGER*4, REAL, DOUBLE PRECISION, REAL *16, COMPLEX, OR COMPLEX*16.

arith-expr is an arithmetic expression

If the data types in the assignment statement differ, F77 will not assign the value of arith-expr directly. Instead, F77 will convert the value of arith-expr to the type of target, and then assign the value. Table 4-1 describes the conversions and assigments carried out in such cases.

Here are some examples of arithmetic assignment statements:

```
SUM = (X + Y)          /*real variable SUM receives
                         the sum of X + Y.

A(I) = B**4.1          /*element I of array A receives
                         the value of B**4.1
```

Table 4-1

Conversion Rules for Mixed-type Assignments
(The routines are described on the following page)

Value Type \ Target Type	I*2	I*4	REAL	DOUBLE	R*16	C*8	C*16
I*2	ASSIGN	EXTEND ASSIGN	FLOAT ASSIGN	DFLOAT ASSIGN	QFLOAT ASSIGN	FLOAT ASREAL	DFLOAT ASREAL
I*4	TRUNC ASSIGN	ASSIGN	FLOAT ASSIGN	DFLOAT ASSIGN	QFLOAT ASSIGN	FLOAT ASREAL	DFLOAT ASREAL
REAL	SFIX ASSIGN	LFIX ASSIGN	ASSIGN	DFLOAT ASSIGN	QFLOAT ASSIGN	ASREAL	DFLOAT ASREAL
DOUBLE	SFIX ASSIGN	LFIX ASSIGN	FLOAT ASSIGN	ASSIGN	QFLOAT ASSIGN	FLOAT ASREAL	ASREAL
R*16	SFIX ASSIGN	LFIX ASSIGN	FLOAT ASSIGN	DFLOAT ASSIGN	ASSIGN	FLOAT ASREAL	DFLOAT ASREAL
C*8	SFIX* ASSIGN*	LFIX* ASSIGN*	ASSIGN*	DFLOAT* ASSIGN*	QFLOAT* ASSIGN*	ASSIGN	DFLOAT
C*16	SFIX* ASSIGN*	LFIX* ASSIGN*	FLOAT* ASSIGN*	ASSIGN*	QFLOAT* ASSIGN*	FLOAT ASSIGN	ASSIGN

Table 4-1 (continued)
Conversion Rules For Mixed-type Assignments

Operation	Action
ASSIGN	Assign value (after any indicated conversion) to the target.
ASREAL	ASSIGN value as above to the real part of a complex number, and set the imaginary part of the complex number to zero.
SFIX	Truncate fractional part and convert result to a short integer. Overflow may occur.
LFIX	Truncate fractional part and convert result to a long integer. Overflow may occur.
FLOAT	Convert value to REAL form. Loss of precision may occur if the argument was DOUBLE PRECISION, COMPLEX*16, or INTEGER*4. Overflow may occur with DOUBLE PRECISION or COMPLEX*16.
DFLOAT	Convert value to DOUBLE PRECISION form.
QFLOAT	Convert value to REAL*16 form.
EXTEND	Prefix the short integer with 16 binary zeros or ones if the short integer was positive or negative, respectively. This cannot change the value or sign of the integer.
TRUNC	Discard the 16 high-order bits of the long integer. A value outside the short-integer range will be altered, and possibly changed in sign, by this operation.

Note

An asterisk affixed to an operation involving a complex number indicates that the operation is to be performed on the real part only. The imaginary part is not involved. When no asterisk is present, the operation is to be performed on both parts of the number.

LOGICAL ASSIGNMENT STATEMENTS

The logical assigment statement has the following format:

```
target = logical expression
```

where:

target is the name of a logical variable or logical array element.

logical expression is a logical expression that must result in a value of .TRUE. or .FALSE.

Some examples of logical assignment statements:

```
CON = A .LT. 3                  /*CON receives .TRUE. or .FALSE.
                                  depending on A.

X = .FALSE.                     /*variable X receives .FALSE.
```

CHARACTER ASSIGNMENT STATEMENT

The character assignment statement has the following format:

```
target = character expression
```

where:

target is the name of a character variable, character array element, or character substring of type CHARACTER.

character expression is a character expression that must be of character data type whose length need not match target. In such cases, the value of character expression will be truncated or blank-extended so that it matches the length of target, then assigned.

Here are some examples of character assignment statements:

```
WORD = 'BELLS'

TREE(1) = 'MERRY_CHRISTMAS'

NAME = 'JINGLE' // WORD
```

ASSIGN STATEMENT

The ASSIGN statement has the following format:

```
ASSIGN s to i
```

where:

s is a statement label of an executable statement

i is an integer variable name

An ASSIGN statement must be executed prior to an ASSIGNED GO TO. Once i has been assigned, it may be used only in an ASSIGNED GO TO until it has been given an integer value by an arithmetic assignment. See Chapter 5 for a discussion of GO TO statements.

Some examples of the ASSIGN statement:

```
ASSIGN 90 TO NUM          /* assigns statement label 90
                             to variable NUM.

ASSIGN 300 TO POOKA       /* POOKA must have been declared
                             as an integer variable.
```

5

Control Statements

Your program begins executing with the first executable statement that appears in the main program and continues executing the statements in order until a transfer of control interrupts the sequence. A transfer of control can be in the form of a procedure reference or a control statement.

This chapter discusses the following control statements:

- GO TO
- IF
- DO
- CONTINUE
- STOP
- PAUSE
- END

For information on the procedure reference statements, FUNCTION, CALL, and RETURN, see Chapter 8.

GO TO STATEMENTS

GO TO statements transfer control to some other executable statement in your program that may not be the next instruction in the normal sequence. There are three kinds of GO TO statements:

- Assigned GO TO
- Computed GO TO
- Unconditional GO TO

Assigned GO TO Statement

An Assigned GO TO statement transfers control to an integer variable that has been defined as a result of an ASSIGN statement. The integer variable has a value that is the statement label of an executable statement in the same program.

The Assigned GO TO statement has the following format:

```
GO TO i [[,] (s [,s]...)]
```

where:

i is an integer variable name.

s is the label of an executable statement in the program unit containing the assigned GO TO. The list of s's is optional. If it appears, the statement label assigned to i must be one of the labels in the list.

The F77 compiler permits 254 labels in an Assigned GO TO.

Here are some examples of the Assigned GO TO statement:

```
ASSIGN 50 TO NUM                    /*This is the same as an
GO TO NUM                             unconditional GO TO 50.

ASSIGN 300 TO ISTART
GO TO ISTART, (99,100,300,400)      /*This is the same as an
                                      unconditional GO TO 300.
```

The Computed GO TO Statement

A Computed GO TO statement transfers control to a statement corresponding to a value in the Computed GO TO expression.

The Computed GO TO statement has the following format:

```
GO TO (s [,s]...)  [,] i
```

where:

i is an integer expression.

s is the statement label (an integer number) of an executable statement appearing in the same program unit as the Computed GO TO statement. The same statement label may appear more than once in the same Computed GO TO statement. Control is transferred to the statement whose label is in the i'th position in the list of s's. If there is no i'th statement, control passes to the next executable statement.

Some examples of the Computed GO TO statement:

```
I = 2
GO TO (10,20,30,40),I          /*The next executable statement
                                 is statement 20.

M = 4
GO TO (99,100,199,200),M       /*The next executable statement
                                 is statement 200.
```

Unconditional GO TO Statement

An unconditional GO TO statement transfers control to the statement label of an executable statement in the same program unit.

The unconditional GO TO statement has the following format:

```
GO TO s
```

where:

s is the label of an executable statement that is in the same program unit as the unconditional GO TO statement.

Here are some examples of unconditional GO TO statements:

```
GO TO 1000                     /*Control is transferred to
                                 executable statement 1000

GO TO 90                       /*Control is transferred to
                                 statement 90.
```

IF STATEMENTS

IF statements make a comparison and then make a decision based upon that comparison. IF statements conditionally transfer control or conditionally execute a statement or a block of statements.

The three types of IF statements are:

- Arithmetic IF
- Logical IF
- Block-IF (IF...THEN, ELSE IF...THEN, ELSE, END IF).

Arithmetic-IF Statement

The arithmetic-IF statement transfers control to one of three statements based upon the value of an expression.

The arithmetic-IF statement has the following format:

```
IF (exp) label1, label2, label3
```

where:

exp is an arithmetic expression with an integer, real, or double precision value.

label1, label2, label3 are labels of executable statements within the current program unit.

When exp is evaluated, control passes to

label1 if the value of exp is negative

label2 if the value of exp is zero

label3 if the value of exp is positive

Here is an example of the arithmetic-IF statement:

```
      ----
      ----
      IF (X - 90) 10, 20, 30
10    A = B * C
20    A = B / C
30    STOP
      END
      ----
      ----
```

In the above example, if the value (X - 90) is negative, control passes to the statement at label 10. If (X - 90) is zero, control passes to the statement at label 20. If (X - 90) is positive, control passes to the statement at label 30.

Note

The arithmetic-IF statement is considered to be obsolete. Although this statement is still supported, it is recommended that you use the block-IF statement instead.

Logical-IF Statement

The logical-IF statement evaluates a logical expression, and then, based upon one of two possible results, executes a single FORTRAN statement.

The logical-IF statement has the following format:

```
IF (exp) stmt
```

where:

exp is a logical expression.

stmt is any valid executable statement except a DO, block-IF, ELSE-IF, ELSE, END-IF, or another logical-IF statement.

If exp evaluates to .TRUE., then stmt is executed. If exp evaluates to .FALSE., then control passes to the next executable statement after the logical-IF.

Here are examples of the logical-IF statement:

```
IF ((SALES .GE. 50.0) .OR. (PROFIT .EQ. 100)) GO TO 999

IF (MAXVAL) CALL SUBR

IF (A .LE. 500) S = S * C
```

In the above examples, if the value of the logical expression (exp) is true, then the statement (stmt) is executed.

Block-IF Structure

The block-IF structure evaluates a logical expression, and either executes or does not execute a group of statements.

A block-IF structure consists of an IF...THEN statement and an END IF statement. The IF...THEN statement is the first statement. In addition, the block-IF structure may contain ELSE IF...THEN and ELSE statements. A block-IF structure may also contain additional IF...THEN, and END IF statements (see "Block-IF Nesting" below.)

Here are some of the more common constructs that the block-IF structure may take:

```
IF exp THEN
   statement-group-1
   END IF


IF exp THEN
   statement-group-1
ELSE
   statement-group-2
END IF


IF exp THEN
   statement-group-1
ELSE IF exp THEN
   statement-group-2
END IF
```

```
IF exp THEN
   statement-group-1
ELSE IF exp THEN
   statement-group-2
ELSE
   statement-group-3
END IF
```

where:

exp is a logical expression.

statement-group-i is any number of executable statements (including 0) which follow an IF...THEN, ELSE IF...THEN, or an ELSE statement. Statement groups may not follow an END-IF statement.

Statements in the Block-IF Structure: This describes the unnested situation. See "Block-IF Nesting" below for the nested situation.

- An IF...THEN statement is the first statement in the block-IF structure. If the logical expression in the IF...THEN statement is true, then the statement group following the IF...THEN statement is executed. Control then passes to the next executable statement following the END IF statement. If the logical expression in the IF...THEN statement is false, control drops down to the next statement after the statement group for the IF...THEN statement. The next statement may be an ELSE IF...THEN, an ELSE, or an END-IF statement. Each IF...THEN must be on a line by itself.

- An ELSE IF...THEN statement is evaluated when the logical expression in the IF...THEN statement is false. If the expression in the ELSE IF...THEN statement is found to be false, then control is passed on to the next statement after the statement group for the ELSE IF...THEN statement. If the value in the ELSE IF...THEN statement is true, then the statement group associated with the ELSE IF...THEN statement is executed. Control then passes to the next executable statement following the END IF statement. There may be any number of ELSE IF...THEN statements in the block-IF structure (see Block-IF Nesting below). The ELSE IF...THEN statement is optional.

- The ELSE statement and its statement group are executed only when all previous logical expressions in either the IF...THEN or any ELSE IF...THEN statements are false. The ELSE statement is optional. The ELSE statement may appear only once in each block-IF structure. When used, the ELSE statement follows the IF...THEN statement and ELSE IF...THEN statements.

- The END IF statement terminates the entire block-IF structure. There is only one END IF statement in each block-IF structure.

Block-IF Considerations: The END statement may not be used within the block-IF structure.

Transfer of control into a block-IF from outside of that block is prohibited. Entry may occur only when program execution reaches the initial IF statement.

When a DO loop is present in a block-IF, it must be wholly contained in the statement group in which it begins. Similarly, when a block-IF is present in a DO loop, it must be wholly contained in the body of the DO loop.

Here is an example of the block-IF structure:

```
----
----
IF (X .GT. 360) THEN
    S = S + X - 360
    N = N + 1
ELSE IF (X .EQ. 360) THEN
    S = S + X
    N = N + 1
ELSE IF (X .EQ. Z) THEN
    S = S + X + Z
    N = N + 1
ELSE
    S = S + X
    Z = X + N
END IF
----
----
```

Block-IF Execution: When the logical expression in an IF...THEN, ELSE IF...THEN, or an ELSE statement is true, the statement group associated with that expression is executed. Control drops down to the end of the Block-IF structure.

When the logical expression in an IF...THEN, ELSE...IF, or an ELSE statement is false, then the statement group associated with that expression is ignored. Control drops down to the next ELSE IF...THEN, ELSE, or END IF statement in the block-IF structure.

At most, only one statement group in a Block-IF structure is executed.

Block-IF Nesting: A block-IF structure may be included in the statement group of another block-IF structure. The nested block-IF structure must be completely enclosed within the statement group. When Block-IFs are nested, the compiler matches an ELSE statement with the most recent IF...THEN statement.

The ELSE IF...THEN, ELSE, and END IF statements of a nested block-IF are local, and do not affect the flow of control of the containing

block-IF. Nested block-IFs should be indented to indicate this independence. It is a good programming practice to indent nested block-IFs.

Here is an example of nested block-IFs:

```
----
----
IF (X .GT. 360 ) THEN
    S = S + X
    IF (Y - Z .LE. 340) THEN
       Y = Y - X
    ELSE
       Y = Y + X
    END IF
ELSE
    S = S + 1
END IF
----
----
```

The program segment above uses indentation to demonstrate the use of nesting levels. Use the -NESTING compiler option to see how the FORTRAN 77 compiler interprets the indentation.

DO STATEMENT

A DO statement sets up a loop that begins at the DO statement and executes zero or more times. The DO statement executes all the statements between the DO statement and its corresponding label. The DO statement has the following format:

```
DO lab [,] var = intval, maxval [,incr]
```

where:

lab is the label of an executable statement that must follow the DO statement in the same program unit. This is called the terminal statement of the DO loop.

var is an integer or real variable, called the DO variable.

intval, maxval, incr are arithmetic constants or expressions that represent the initial value, the terminal value, and increment parameters, respectively.

The range of the DO statement includes all the statements between it

and the terminal statement. The terminal statement must not be one of the following:

- Unconditional or assigned GO TO statement
- Arithmetic IF statement
- Any block IF statement
- RETURN, STOP, or END statements
- DO statement

Execution of a DO Statement

The following steps are followed when a DO statement is executed:

1. intval, maxval, and incr are evaluated to establish values for the initial, terminal, and increment parameters, respectively, including conversion to the type of the DO variable, var, if necessary. If incr is omitted, the increment parameter defaults to a value of one. incr cannot evaluate to zero.

2. The value of the initial parameter is assigned to the DO variable.

3. The iteration count (number of times to execute the body of the DO loop) is determined by the following expression:

```
INT ((maxval-intval+incr)/incr)
```

   If the iteration count is zero or less, the body of the DO loop will not be executed.

4. If the type of the DO variable is integer, then the expected final value that it will contain upon normal termination of the DO loop will be calculated. This final value is the first value which var would contain, as a result of normal incrementation, that is greater than maxval if incr is greater than zero, or less than maxval if incr is less than zero.

   If the type of the DO variable is real, then its final value will not be calculated.

5. The DO loop variable must not be altered inside the range of the loop. A Severity 2 error message appears if the compiler detects the appearance of the DO-LOOP variable on the lefthand side of an assignment statement. A Severity 1 warning is issued if the variable is being passed as an actual argument, due to its potential for modification.

Example:

```
* The DO variable ID is initialized to equal 3
* The intval is 3
* The maxval is 10
*           - when ID => 10 the program stops
*           executing the loop and continues
* The incr is 2
*           -must be declared if any value other than 1
* The lab, 210, is the statement number (label) of the last
*           -statement that will execute as a part of the
           -loop
      DO 210 ID = 3, 10, 2
      ---------
210   CONTINUE
      ---------
      STOP
      END
```

Execution of the Range of DO Statements

After execution of the DO statement, statements in the range of the DO loop will be executed, up to and including the terminal statement, provided that the calculated number of iterations is greater than zero. The DO variable may not become redefined during execution of the DO loop. Variables in the expressions for the initial, terminal or increment parameters can be modified inside the loop without affecting the number of times the DO loop will execute.

Iteration Control

After completion of the DO loop body, incrementation of the DO variable by the value of the increment parameter occurs. There are different methods employed to determine when to terminate execution of the DO statement:

1. If the type of the DO variable is integer and if the value of the DO variable has not yet reached the expected final value computed during processing of the DO statement, then the body of the DO loop will be executed again.

2. If the type of the DO variable is real, then the iteration count is decremented by one and the loop body will be executed again until the iteration count reaches zero. Note that the number of actual iterations of a loop controlled by a real DO

variable may not be what is expected, due to rounding errors.

When iteration control terminates execution of the DO loop, program control passes to the statement immediately following the last statement in the loop, and the DO variable retains the value it had when execution of the loop terminated. Execution of a DO loop may also be terminated by a statement within the loop that branches out of the loop. In this case, the DO variable retains its current value.

Nested Loops and Transfer of Control

DO loops may be nested within other loops, provided that the range of each loop is completely contained within the range of the next outermost loop. Nested DO loops may share a labeled terminal statement.

Here is an example which correctly demonstrates the use of nested DO loops.

```
      ----
      ----
      ----
      DO 400 M = 1, 3                          /* FIRST LEVEL LOOP
      JO =JO + M
      DO 600 K = 1, 4                          /* SECOND LEVEL LOOP
      JO = JO + K
      IF ((K .GT. M) .XOR. (K .GT. J))  LA =2
      DO 550 L = LA, 5                         /* THIRD LEVEL LOOP
      N = N + 1
      JO = JO + L
      PRINT*, JO, N, L
550   CONTINUE                                 /* END OF THIRD LEVEL LOOP
      PRINT*, JO, K, N
600   CONTINUE                                 /* END OF SECOND LEVEL LOOP
      PRINT*, JO, N, M
400   CONTINUE                                 /* END OF FIRST LEVEL LOOP
      ----
      ----
      STOP
      END
```

Restrictions on Transfer of Control

Program control may not transfer to a statement within a DO loop; therefore, extended-range DO loops are not supported. Control may be transferred from a nested loop to an outer one, but may not be transferred from an outer loop inwards. Two or more nested DO loops can share the same terminal statement. However, any transfer of control to a statement that is not within the innermost DO loop is considered to be a transfer from an outer loop into an inner loop,

because the terminal statement is within the range of the innermost loop.

Here is an example that demonstrates an invalid attempt to pass control from outside a DO loop to a label with the range of a loop.

```
      ----
      ----
      DO 600 K = 1, 4                          /* FIRST LEVEL LOOP
      JO = JO + K
      NUN = 4
      IF ((K .GT. M) .XOR. (K .GT. J))  LA =2
      DO 550 L = LA, 5                         /* SECOND LEVEL LOOP
      N = N + 1
      IF (NUN .EQ. LA + 2) GO TO 600
      JO = JO + L
525   PRINT*, JO, N, L
550   CONTINUE                                 /* END OF SECOND LEVEL LOOP
      PRINT*, JO, K, N
      IF (NUN .EQ. 4) GO TO 525
600   CONTINUE                                 /* END OF FIRST LEVEL LOOP
      PRINT*, JO, N, M
      ----
      ----
      STOP
      END
```

In the example above, the statement

```
IF (NUN .EQ.  4) GO TO 525
```

invalidly attempts to pass control from the second level DO loop to the first level DO loop. However, the statement:

```
IF (NUN .EQ.  LA + 2) GO TO 600
```

has a valid target label (600). Control is conditionally transferred outside the loop.

FTN Compatibility of DO Loops

When programs are compiled with the compiler option -D01, F77 will generate DO loops that execute similarly to those generated by FTN. In addition to this, F77 does not impose the syntactical restrictions on the DO statement that FTN does, so that any valid FORTRAN-77 standard DO statement will be compilable under "-D01".

Differences in the behavior of D01 loops as opposed to FORTRAN-77 loops are outlined here:

- D01 loops always execute at least once.

- Extended-range DO loops are permitted. This occurs when a DO loop contains a statement that transfers control outside of the loop to a sequence of one or more statements, the last of which will transfer control back into the DO loop.

- Evaluation of the terminal and increment parameters of the DO statement occurs differently in different cases:

  If these parameters are scalar variables, then they are reevaluated during each iteration of the loop. Therefore, modification of them in the range of the DO loop will affect the number of times the loop executes.

  If these parameters are expressions, then they are evaluated only once during execution of the DO statement, and not during each iteration. Modification of any variables that appear in such expressions will not, therefore, affect the number of times the DO loop executes. Furthermore, such loops cannot define an extended-range, and F77 will issue a warning to that effect when it encounters such a situation.

DO WHILE Statement

A DO WHILE statement permits loop iteration to execute an indefinite number of times, based on the value of the logical expression being true. The DO WHILE statement can have one of the following formats:

Format 1

```
      DO s[,] WHILE(e)
         -
         -
   s  statement
```

where:

s is the label of a statement that must physically follow in the same program unit.

e is a logical expression.

statement is any valid FORTRAN DO loop terminal statement or END DO.

For example:

```
          N=10
          M=1
          DO 100, WHILE (M .LT. N)
             ARRAY(M) = 1.1
          M = M + 1
100       CONTINUE
```

Format 2

```
DO WHILE(e)
  -
  -
END DO
```

where:

e is a logical expression.

For example:

```
N=10
M=1
DO WHILE (M .LT. N)
   ARRAY(M) = 1.1
   M = M + 1
END DO
```

Execution of a DO WHILE Statement

The DO WHILE statement is different from a DO statement because it executes for as long as a logical expression contained in the statement continues to be true.

In the examples above, an array is initialized until the condition in the DO statement is false. In both cases the condition (M .LT. N) becomes false when M=10.

Nested DO WHILE Loops

DO WHILE loops may be nested according to the following conditions:

- Each labeled DO WHILE must be closed with a matching labeled statement.
- Each unlabeled DO WHILE must be closed with an unlabeled END DO.

An unlabeled END DO may have only one DO WHILE loop.

END DO Statement

The END DO statement can be used to end the range for both DO and DO WHILE loops. If the END DO statement is used with a DO statement, the END DO statement must be labeled.

The END DO statement has the following format:

```
END DO
```

CONTINUE Statement

In your program, a CONTINUE statement serves as a point of reference, and merely transfers control to the next executable statement.

The CONTINUE statement has the following format:

```
CONTINUE
```

A CONTINUE statement is usually used to indicate the end of the range of a DO loop; however, you can use it anywhere in your program where an executable statement is allowed.

For example:

```
      SUM = 6.5
      DO 10  K = 1,5
         SUM = SUM + A(K)
10    CONTINUE
```

STOP Statement

The STOP statement terminates the execution of your program.

The STOP statement has the following format:

```
STOP [n]
```

where:

n is an optional decimal number of up to five digits or a character constant.

A STOP statement halts program execution, closes all file units referenced by the program, prints ****STOP n at your termninal, and returns control to PRIMOS. A STOP statement may appear anywhere in a program unit. In a main program, an END without a STOP causes a STOP to occur automatically.

PAUSE Statement

A PAUSE statement temporarily suspends execution of your program until you or an operator intervenes.

The PAUSE statement has the following format:

```
PAUSE [n]
```

where:

n is an optional decimal number of up to five digits, or is a character constant.

Using this statement will halt your program and display a ****PAUSE n message on your terminal. Program execution will remain suspended until you type the PRIMOS command START. Execution begins at the next executable statement following the PAUSE.

For example:

```
OK, RESUME CIRCLE
 TYPE IN A VALUE FOR THE RADIUS OF THE CIRCLE:
23
**** PAUSE
OK, START
 THE AREA OF THE CIRCLE IS:    144.713
**** STOP

OK,
```

END Statement

The END statement is the final statement of a program, subroutine (including a BLOCK DATA subprogram), or external function. It tells the compiler that it has reached the physical end of the program unit.

The END statement has the following format:

```
END
```

In a main program, END implies STOP if no STOP statement precedes it. In a subprogram, END implies RETURN if no RETURN statement precedes it.

6

Input/Output Statements, Data Storage, and File Types

Input is the transfer of data values from a file to internal storage. Output is the transfer of data values from storage to a file. Input and output statements control these transfers, and may also specify the representation of the data values on the file. In addition to the F77 input/output statements used in the transfer of data, this chapter also discusses:

- F77 data storage
- Records
- Files and programs
- List and namelist directed I/O

The following discussion is intended as a review, to establish the context in which FORTRAN I/O commands operate. If you are not familiar with the features mentioned, please consult one of the suggested textbooks in Chapter 1.

Input/Output in FORTRAN 77 is based on logical records stored in files. The physical aspects of record and file storage are not dealt with by the language. Therefore, the following descriptions are concerned only with the logical structures involved.

F77 DATA STORAGE

A file is a collection of related records. A file may be empty, or may contain one or more records. Each record is a part of the file because it contains data items similar to all the other records in the file. A record has up to 2048 bytes and is the basic unit of data transfer.

Every open file has a pointer. When a file is first opened, its pointer is positioned before the first record. For data transfer, the pointer first moves to the beginning of the selected record (direct access) or the next record in the file (sequential access), then sweeps across the record as the record is read or written. After data transfer, the pointer remains at the end of the record just read or written, or after the endfile record if one was written or encountered.

Types of Records

There are three types of records:

- Formatted
- Unformatted
- Endfile

No file may contain both formatted and unformatted records.

Formatted Record: A formatted record consists entirely of Prime ECS characters. Such a record may be read or written only by formatted input/output statements containing explicit format specifiers.

Unformatted Record: An unformatted record contains data in the same form in which it is actually used by the computer. No format list is used when it is accessed. The data is transcribed directly to or from the storage medium.

Endfile Record: An endfile record is written by an ENDFILE statement. It may occur only as the last record of a sequential file. If an endfile record is encountered during a READ, the system will be informed that the file has been exhausted. See the discussion of the ENDFILE statement later in this chapter.

Record Lengths

The record length of formatted records is measured in characters (bytes) while the length of unformatted records is measured in 16-bit halfwords. Formatted and unformatted records may be stored in either fixed or variable length form. No file may contain both fixed and variable length records.

Fixed Length: A file of fixed-length records is produced when the RECL (record length) option is given in the OPEN statement creating the file. All records written to the file will be of the length specified.

Use of the RECL option for a sequential-access file is an F77 extension. (For a complete description of the RECL option refer to the OPEN statement presented later in this chapter.)

Varying Length: A file of variable-length records is produced when the RECL option is omitted from the OPEN statement creating the file. Each record will have the length needed to hold its data, up to the current maximum record length. See INCREASING MAXIMUM RECORD LENGTH later in this chapter. Files of variable-length records cannot be used under direct access.

Implementation: The following information is significant when space conservation on disk files is a major concern.

Files of variable-length records are kept in compressed ASCII format. Sequences of blanks are replaced by one instance of the blank character, a compression indicator, and a repeat count. All such files are processed through Physical Device 7. Compressed format saves disk space, but requires some additional processing time to compress and uncompress the records.

Files of fixed-length records are kept in uncompressed ASCII format. Records are stored just as they are created by the program, with no compression. All such files are processed through Physical Device 8.

Uncompressed format can be quite wasteful of space, but I/O on uncompressed files is faster than on compressed files because no time is spent compressing and decompressing the records.

Types of File Access

There are two types of file access:

- Sequential
- Direct

Sequential Access: With sequential access, the file pointer can move only one record at a time, either forward or backward, or it may be positioned at the beginning of the file.

Direct Access: With direct access, the file pointer may be positioned to the beginning of any record in the file.

Types of Files

The terms SAM and DAM refer to the basic file organization PRIMOS uses to implement files. These organizations are not specific to any one language. The terms sequential access method and direct access method refer to the two types of file access offered by the FORTRAN 77 language. Either type of FORTRAN file could be implemented using either PRIMOS file organization. The implementation used is transparent at the programming level, and is subject to change. See the Subroutines Reference Guide.

SAM Files: In a SAM file, the records are stored in the order they were written, and are usually read in that order. New records can be added only to the end of the file, and records cannot be deleted. SAM files can be read or written only under sequential access. SAM records may be of fixed or variable length.

DAM Files: In a DAM file, the records are stored in a manner that enables direct access. New records can be added anywhere in any order. Existing records can be deleted by overwriting them. DAM files must be written only under direct access, but can be read by either direct or sequential access. DAM records must always be fixed-length. Every record in a DAM file is identified by a key (a positive integer). This key is specified when the record is written. Under direct access, a record is retrieved by giving its key in a direct access READ statement. Under sequential access, a DAM file acts like a SAM file to which the records were written in order by key. A record is retrieved by reading through the file until it is reached.

> **Caution**
>
> A direct access file must not be modified by the EDITOR, EMACS, or any sequential data transfer statement, or its usefulness for direct access will be partly or wholly lost.

Internal Files

Internal files provide a way to convert data from one form to another within main memory. An internal file is an area of memory where a type CHARACTER variable, array, array element, or substring is stored. Such an area acts as an internal file when the name of the data item stored there is given in place of the file unit number in a formatted, sequential READ or WRITE statement.

The READ or WRITE proceeds as usual, but the "file" used is the designated internal storage area, rather than an external file on secondary storage. Data is transferred to or from the file area, after conversion as directed by the associated format list. List directed formatting is not permitted with internal READ or WRITE statements.

After each read or write file the pointer returns automatically to the beginning of the record.

The characteristics of an internal file are as follows:

- An internal file consists of a character variable, character array, or character substring.
- A record of an internal file is a character variable, an element of a character array or character substring.
- If the file is a character variable, character array element, or a character substring then it consists of a single record whose length is the same as the length of either the variable, array element, or substring. When the file is a character array, it is treated as a sequence of character array elements, and each array element is a record of the file. The sequence of the file records is the same as the sequence of the array elements. Each record of the file is the same length as the length of an array element.

EDITING F77 FILES

The PRIMOS EDITOR and EMACS text editing utilities produce and expect SAM files of formatted, variable-length records. A file created with these attributes by an F77 program may be edited freely. A file created by EDITOR or EMACS may be opened with formatted, variable-length records in an F77 program and modified as desired.

EDITOR and EMACS should not be used to modify a SAM file of fixed-length, formatted records because they will automatically compress the file, effectively transforming it to a file of variable-length records. Neither should they be used on a DAM file, since they will not maintain the fixed-length records a DAM file requires. Text editors can be used to examine a fixed-length file provided it is not modified. EDITOR and EMACS cannot be used to process unformatted files.

INCREASING MAXIMUM RECORD LENGTH

When the shared libraries are used in linking an F77 program (unqualified LI command to BIND during linking) records of all types have a maximum length of 32K bytes. This limit cannot be increased.

When the unshared libraries are used (LI NPFTNLB and LI IFTNLB to BIND during loading), the maximum record size is initially 256 bytes, but it may be increased to a maximum of 32K bytes. When records longer than 256 bytes are needed, the PRIMOS I/O Control System (IOCS) must be notified. Two aspects of IOCS are involved:

- The size specified by the variable in the I/O size-control block F$IOSZ.
- The size of the I/O buffer F$IOBF. This buffer is discussed further under DATA TRANSFER STATEMENTS later in this chapter.

Specifications to IOCS must be given in halfwords. To increase the maximum record length, proceed as follows:

1. Increase the value specified in F$IOSZ to the desired record length by inserting the following statements into the main program:

```
COMMON/F$IOSZ/MAXSIZE
INTEGER*2 MAXSIZE/halfwords/
```

   where halfwords is an integer constant giving the desired record length in two-byte halfwords (half the length in bytes).

2. Increase the size of F$IOBF to the desired record length by inserting the following statements into the main program:

```
COMMON/F$IOBF/BUFSIZE
INTEGER*2 BUFSIZE(halfwords)
```

   where halfwords is as above.

Any variable names could be used in place of MAXSIZE and BUFSIZE.

No special action is needed to obtain the maximum record size when using the shared libraries, since they automatically provide a MAXSIZE and BUFSIZE of 16K halfwords (32K bytes).

Note

The value in F$IOSZ and the size of F$IOBF set an upper size limit on all records, but do not determine the actual record size for any particular file. The actual record size for a fixed-length file is determined by the RECL option in the OPEN statement for the file. Arguments to RECL must be given in bytes. For a formatted file, the arguments given must be in bytes, but for an unformatted (binary) file, the number is in halfwords. For variable-length files, including the terminal, it depends on the individual record.

FILES AND PROGRAMS

Before a program can read or write a file, the programmer must establish a connection between the file and the program. This is accomplished by assigning a device if necessary, and by opening the file on a FORTRAN unit.

Assigning a Device

When a file is on the card punch or reader, the paper tape punch or reader, a magnetic tape drive, or is being written directly to the line printer without the use of SPOOL, the device must be assigned, using the PRIMOS ASSIGN command, before program execution begins. See the Prime User's Guide.

Opening a File on a File Unit

A file unit is a numbered channel through which data passes between a program and a file. Every file except the user's terminal, which is always open on FORTRAN unit 1, must be connected to a file unit prior to data transfer. There are three ways of doing this:

- With the FORTRAN 77 OPEN statement. This is the recommended way.
- With a call to one of the PRIMOS file-opening subroutines. These provide more power and flexibility than the FORTRAN 77 OPEN, but these advantages are usually not needed. See the Subroutines Reference Guide for details on the PRIMOS file-opening subroutines.
- With a PRIMOS OPEN command executed before the program is run. This is known as preconnection.

A preconnected file may be opened again within the program, and additional attributes added to the connection. In case of conflicting attributes, those specified within the program take precedence.

> Caution
>
> PRIMOS and F77 use different numbering systems to describe their set of file units. When a file unit is referenced in F77, its FORTRAN unit number must be used. When it is referenced in a PRIMOS subroutine call, the corresponding PRIMOS Funit number must be given instead. Beware of confusing the two descriptive systems. See Table 6-1.

Integer arguments to most PRIMOS subroutines must be INTEGER*2.

Table 6-1
Devices and Their Default FORTRAN Unit Numbers

FORTRAN Unit Number	PRIMOS 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	Funit 17
.	.
.	.
.	.
.	.
.	.
139	Funit 127
140	Printer unit 0
141	Printer unit 1

The mapping of FORTRAN unit numbers to PRIMOS devices shown here may be altered for the duration of a program through a call to the PRIMOS Subroutine ATTDEV. See the Subroutines Reference Guide.

FILE OPERATIONS

The possible operations on a file and the statements that accomplish the operation are:

- Create a new file (OPEN).
- Access an old file (OPEN).
- Change file-connection attributes (OPEN).
- Determine current status and attributes of a file (INQUIRE).
- Transfer data to/from a file (READ, WRITE, PRINT).
- Indicate the end of a file (ENDFILE).
- Reposition the file pointer (BACKSPACE, REWIND).
- Disconnect from a file (CLOSE).
- Delete a file (Options in OPEN and CLOSE).

The statements that perform these operations are divided into three categories:

- File control statements:

  OPEN
  CLOSE
  INQUIRE

- Device control statements:

  ENDFILE
  BACKSPACE
  REWIND

- Data transfer statements:

  READ
  WRITE
  PRINT

FILE CONTROL STATEMENTS

File control statements establish, alter, or read out the current attributes and status of a file. In file control statements, all integer arguments must be INTEGER*4 and all logical arguments must be LOGICAL*4. An argument that is not an expression may be either a variable or an array element.

OPEN Statement

An OPEN statement may be used to create a new file and establish its basic properties, or to connect a file to a file unit and establish the properties of the connection. For a new file, one OPEN statement will perform both these functions. The same file may be connected with different properties at different times, but must always be closed before it is reopened. The BLANK='s specifier is the only specifier that you can change from its current value to a new value without terminating the connection of a file to a unit.

The OPEN statement has the following format:

```
OPEN ([UNIT= ]unit# [,FILE= filename] [,STATUS= stat] [,ACCESS= acc]
     [,FORM= fm] [,RECL= reclngth] [,BLANK= blnk] [,ERR= label]
     [,IOSTAT= ios] [,ACTION= act])
```

where:

UNIT= unit# specifies the logical unit you want to open.

FILE= filename specifies the name of the file to be connected to the unit.

STATUS= stat specifies the status of the file to be opened.

ACCESS= acc specifies whether the type of file access is sequential or direct.

FORM= fm specifies whether the file is being connected for formatted or unformatted I/O.

RECL= reclngth specifies the logical record length of a file.

BLANK= blnk specifies the interpretation of blanks in numeric input fields.

ERR= label specifies the transfer of control if an error occurs.

IOSTAT= ios specifies an error status indicator.

ACTION= act specifies whether a file is to be opened for reading only, writing only, or both.

The options used may be given in any order, except that if UNIT= is omitted, unit# must appear first. The options, their defaults, and the data types required for the arguments, are described in Table 6-2.

The following is an example of the OPEN statement:

```
INTEGER*4  STATVAL
CHARACTER*20  ACCTYPE
ACCTYPE = 'SEQUENTIAL'
OPEN (10, FILE= 'YORD', STATUS= 'OLD', ACCESS= ACCTYPE,
+     FORM= 'FORMATTED', RECL= 25, ERR= 999, IOSTAT= STATVAL)
```

An existing file named YORD is opened for formatted sequential access on FORTRAN unit 10. The record length is 25. Should a numeric field containing blanks be read from the file, the blanks will be deleted. Should an error occur -- for instance if the file does not in fact exist, or unit 10 is already in use -- control will transfer to Statement 999, and STATVAL will be given a positive value.

PRIMOS File-opening Subroutines: These permit files to be created interactively at runtime, allow files to be opened with various protection attributes, and provide other services additional to those of the FORTRAN 77 OPEN statement. See the Subroutines Reference Guide. See also the Caution under FILES AND PROGRAMS earlier.

Table 6-2
OPEN Statement Options

Option	Argument Data-type	Results of Arguments Specified
UNIT=	Integer*4 Expression	File is opened on the FORTRAN unit specified.
FILE=	Character Expression	The file has the name specified. A pathname may be used. If no FILE= is specified for a non-scratch file, the file will be named F#nnn where nnn is the number of the file unit on which the file was opened.
STATUS=	Character Expression	'OLD': Specified if the file already exists.
		'NEW': Specified if the file is being created. A filename must be specified using the FILE= option.
		'SCRATCH': File is temporary: it will be automatically deleted at program end. No filename may be specified.
		'UNKNOWN': (Default) Specified if the status is not known to the programmer. The processor will determine the appropriate status.
ACCESS=	Character Expression	'SEQUENTIAL': (Default) File is connected for sequential access.
		'DIRECT': File is connected for direct access.
FORM=	Character Expression	'FORMATTED': (Default under sequential access) File is connected for formatted data transfer.
		'UNFORMATTED': (Default under direct access)File is connected for unformatted data transfer.

Table 6-2 (continued)
OPEN Statement Options

Option	Argument Data-type	Results of Arguments Specified
RECL=	Integer*4 Expression	Sets record length for a file of fixed-length records in words. Must be omitted for a file of variable-length records. Use in sequential access files is an F77 extension. Required in direct access files. See page 6-3 for a discussion of record lengths.
BLANK=	Character Expression	This item specifies treatment of blanks in numeric input fields when data is read into the file. 'NULL': (Default) All blanks are deleted, and digits compressed to the right side of the input field. An all-blank field will be interpreted as a 0 value. 'ZERO': All but leading blanks are converted to 0's, as in FORTRAN 66.
ERR=	Statement Label	Control transfers to statement specified if an error occurs during execution of the OPEN statement.
IOSTAT=	Integer*4 Variable	Set to 0 if the OPEN statement executes successfully. Set positive on error in OPEN-statement execution.
ACTION=	Character Expression	Allows files to be opened for reading only, writing only, or both. READ — Allows reading and file positioning operations to be performed. WRITE — Allows writing and file positioning operations to be performed. READ/WRITE — (Default) Allows all types of I/O operations to be performed.

CLOSE Statement

The CLOSE statement disconnects a file from a unit, regardless of the number of times you've reopened this file and unit. The CLOSE statement has the following format:

```
CLOSE ([UNIT= ]unit# [,STATUS= stat] [,ERR= label] [,IOSTAT= ios])
```

where:

ERR= and IOSTAT= have the same meanings as in the OPEN statement.

STATUS= determines the final disposition of the file.

The argument stat is a character expression which may have the values:

'KEEP'	The file will be retained after it is closed. This is the default for non-SCRATCH files, and must not be given for SCRATCH files.
'DELETE'	The file will be deleted after it is closed. Default for SCRATCH files.

The options used may be given in any order, except that if UNIT= is omitted, unit# must appear first.

When execution terminates normally, all files opened or referenced in the program (except COMO files) are automatically closed. However, when execution terminates due to an error, all open files remain open.

INQUIRE Statement

An INQUIRE statement is used to ascertain the properties of a file, or of its connection to a file unit. The INQUIRE statement has the following format:

```
INQUIRE   (FILE= filename or [UNIT= ]unit# [,IOSTAT= ios]
          [,ERR= s] [,EXIST= ex] [,OPENED= od] [,NUMBER= num]
          [,NAMED= nmd] [,NAME= fn] [,ACCESS= acc]
          [,SEQUENTIAL= seq] [,DIRECT= dir] [,FORM= fm]
          [,FORMATTED= fmt] [,UNFORMATTED= unf] [,RECL= rcl]
          [,NEXTREC= nr] [,BLANK= blnk])
```

where:

FILE= filename specifies the name of the file being inquired about.

UNIT= unit# specifies the number of the logical unit being inquired about.

IOSTAT= ios specifies an error status indicator.

ERR= s specifies the transfer of control if an error occurs.

EXIST= ex specifies whether a file or unit exists.

OPENED= od specifies whether a file or file unit is open.

NUMBER= num specifies whether a logical unit is connected to a file.

NAMED= nmd specifies whether a file or unit has a name.

NAME= fn specifies the name of the file being inquired about.

ACCESS= acc specifies whether the type of file access is sequential or direct.

SEQUENTIAL= seq specifies whether sequential access is allowed for the file.

DIRECT= dir specifies whether direct access is allowed for the file.

FORM= fm specifies whether the file is being connected for formatted or unformatted I/O.

FORMATTED= fmt specifies the record type of the file.

UNFORMATTED= unf specifies the record type of the file.

RECL= rcl specifies the record length of the file for direct access.

NEXTRC= nr specifies the number of the next record in the file.

BLANK= blnk specifies the interpretation of blanks in numeric input fields.

Each option acts as a question. When the INQUIRE statement executes, the variable you supply for each option is set to a value that answers the question the option asks. The correct data types for the variables, and the meanings of the various responses, are described in Table 6-3.

The file must be specified by name (INQUIRE by name) or unit (INQUIRE by unit) but not both. Options may appear in any order, but no option may appear more than once. If FILE= (or UNIT=) is omitted, the filename (or unit#) must appear first.

A variable or array element that may become defined or undefined as a result of its use in an INQUIRE statement, or any associated data item, must not be referenced by any other option in the same INQUIRE statement.

Table 6-3
INQUIRE Statement Options

Specifier	Argument Data Type	Significance of Possible Values
FILE=	Character Expression	Specifies file by name.
UNIT=	Integer*4 Expression	Specifies file by unit number.
IOSTAT=	Integer*4	Zero: no error condition exists. Positive: error condition exists.
ERR=	Statement number	Control transfers to statement indicated if error occurs during INQUIRE statement execution.
EXIST=	Logical*4	.TRUE.: the file exists (for INQUIRE by name) or the unit exists (for INQUIRE by unit). .FALSE.: the file or the unit does not exist.
OPENED=	Logical*4	.TRUE.: the file is open (INQUIRE by name) or the file unit is open (INQUIRE by unit). .FALSE.: the file or the unit is not open.
NUMBER=	Integer*4	Variable supplied is set to the file's unit-number. If there is none, variable becomes undefined.
NAMED=	Logical*4	.TRUE.: the file has a name. .FALSE.: the unit has no name.
NAME=	Character	Variable is set to the filename. If none or file not connected, variable becomes undefined.

Table 6-3 (continued)
INQUIRE Statement Options

Specifier	Argument Data Type	Significance of Possible Values
ACCESS=	Character	'SEQUENTIAL': file open for sequential access. 'DIRECT': file open for direct access. Becomes undefined if file is closed.
SEQUENTIAL=	Character	'YES': file can be connected for sequential access. 'NO': cannot connect file for sequential access. 'UNKNOWN': suitability of the file for sequential access cannot be determined.
DIRECT=	Character	'YES': file can be connected for direct access. 'NO': file cannot be connected for direct access. 'UNKNOWN': suitability of file for direct access cannot be determined.
FORM=	Character	'FORMATTED': open for formatted data transfer. 'UNFORMATTED': open for unformatted data transfer. Becomes undefined if file is not open.
FORMATTED=	Character	'YES': file consists of formatted records. 'NO': file consists of unformatted records. 'UNKNOWN': record type cannot be determined.

Table 6-3 (continued)
INQUIRE Statement Options

Specifier	Argument Data Type	Significance of Possible Values
UNFORMATTED=	Character	'YES': file consists of unformatted records. 'NO': file consists of formatted records. 'UNKNOWN': record type cannot be determined.
RECL=	Integer*4	Variable is set to the record-length for which the file is open. Becomes undefined if file consists of varying-length records or is closed.
NEXTREC=	Integer*4	Variable is assigned the value n+1 where n is the record number of the last record read or written on a file connected for direct access. If no records have been read or written, the variable is set to 1. If the file is not connected for direct access, or if the position of the file pointer is indeterminate due to a previous error, the variable becomes undefined.
BLANK=	Character	'ZERO': non-leading blanks in numeric fields will be converted to 0's. 'NULL': non-leading blanks in numeric fields will be deleted. If the file is not open for formatted data transfer, the variable becomes undefined.

DEVICE CONTROL STATEMENTS

These statements apply only to sequential (SAM) files. They reposition the file pointer, either physically (file on tape) or logically (file on disk), or write the endfile record that prevents a device from reading past the end of a file.

BACKSPACE Statement

The BACKSPACE statement moves the pointer of a file open for sequential access back to the beginning of the previous record. The BACKSPACE statement has the following format:

```
BACKSPACE unit#

BACKSPACE ([UNIT= ] unit# [,IOSTAT= ios] [,ERR= label])
```

where:

unit# is an external unit identifier

UNIT= unit# is an expression that evaluates to the integer number of the unit you want to backspace. UNIT= can be omitted if unit# is the first argument.

IOSTAT= ios causes ios to become set to 0 if the BACKSPACE statement executes successfully, or, set positive on error in BACKSPACE statement execution.

ERR= label is a statement label, label where control will transfer to on an error.

BACKSPACE may be performed:

- On any formatted file, except that records written using list-directed I/O may not be backspaced over.
- On any unformatted file having a fixed record-length (RECL size specified in the OPEN statement).

Note

To use the BACKSPACE statement on a file opened with an ACTION= WRITE specifier, you must have system read access rights to the file.

When a file pointer is positioned after the endfile record, as is the case after the ENDFILE condition has been raised, BACKSPACE will reposition the file pointer before the endfile record. When a file pointer is at the initial point of the file, BACKSPACE has no effect.

The BACKSPACE statement cannot be used on an unformatted file with a variable record length.

REWIND Statement

The REWIND statement repositions the file pointer to the initial point of a file, either by physically rewinding a tape, or by resetting a disk file's logical pointer. The REWIND statement has the following format:

```
REWIND unit#

REWIND ([UNIT= ]unit# [,IOSTAT= ios] [,ERR= label])
```

where:

unit# is an external unit identifier

UNIT= unit# is an expression that evaluates to the integer number of the unit you want to rewind. UNIT= can be omitted if unit# is the first argument.

IOSTAT= ios causes ios to be set to 0 if the REWIND statement executes successfully. Set positive on error in REWIND statement execution.

ERR= label is a statement label, label where control will transfer to on an error.

When a file pointer is at the initial point of the file, REWIND has no effect.

ENDFILE Statement

The ENDFILE statement writes a device-specific endfile record on the file connected to the FORTRAN unit unit#. The pointer is left positioned after the endfile record. This statement can also be used to truncate disk files. The ENDFILE statement has the following format:

```
ENDFILE unit#

ENDFILE ([UNIT= ]unit# [,IOSTAT= ios] [,ERR= label])
```

where:

unit# is an external unit identifier

UNIT= unit# is an expression that evaluates to the integer number of the unit you want to write an endfile record. UNIT= can be omitted if unit# is the first argument.

IOSTAT= ios causes ios to be set to 0 if the ENDFILE statement executes successfully. Set positive on error in ENDFILE statement execution.

ERR= label is a statement label, label where control will transfer to on an error.

The following rules apply to the ENDFILE statement:

- On a sequential tape file, an endfile record must be explicitly written following the last data record.
- On a sequential disk file, the computer will supply an endfile record automatically whenever one is appropriate. However, use of an explicit ENDFILE statement for such files is strongly recommended, for compatibility with other systems.
- On a DAM file, no endfile record should ever be written. If one is, unpredictable and undesirable results will occur.

DATA TRANSFER STATEMENTS

These statements control the actual transfer of data between files and program variables. READ transfers data from files. WRITE and PRINT transfer data to files.

How a Data Transfer Statement Works

Data is not transferred directly between files and program variables. In a READ, the current record is first transferred from the file to the FORTRAN I/O buffer F$IOBF, which resides in main memory. The FORTRAN I/O system then scans F$IOBF (using a pointer similar to a file pointer), reads out the separate data items, edits them if the READ is formatted, and assigns them to the appropriate variables. In a WRITE, the order is reversed: the data items are edited or transferred into F$IOBF, then the contents of F$IOBF are written as a whole to the file.

Usually F$IOBF is scanned sequentially. However, the T edit-control descriptor can be used in a formatted data transfer to scan it in any desired order. See Edit-Control Descriptors in Chapter 7.

For simplicity, the following descriptions will not mention F$IOBF, since you do not need to be concerned with it except when its size must be increased (See INCREASING MAXIMUM RECORD LENGTH above) or the T descriptor is used.

Data transfer statements may be used to convert data from one type to another. See Internal Files above.

Note

A function must not be referenced anywhere in a data transfer statement if the function itself causes execution of a data transfer statement.

READ Statement

A READ statement transfers data from an internal file. The READ statement has the following formats:

```
Sequential:    READ format [,input list]

               READ ([UNIT= ]unit# [,[FMT= ]format] [,END= label]
               [,ERR= label] [,IOSTAT= ios]) [input list]

ANSI direct:   READ ([UNIT= ]unit# [,[FMT= ]format] ,REC= record#
               [,END=label] [,ERR=label] [,IOSTAT=ios]) [input list]

IBM direct:    READ (unit#'record# [,[FMT= ]format] [,END= label]
               [,ERR= label] [,IOSTAT= ios]) [input list]

NAMELIST:      READ ([UNIT=]unit#,blockname)
```

The unit# is an integer expression specifying the FORTRAN unit to be read. It must be present. All other items are optional. An asterisk may be given for unit#. This is equivalent to specifying FORTRAN unit 1, the terminal.

If a format is present, the read is formatted. Otherwise it is unformatted. A format may be any of the following:

- The statement number of a FORMAT statement (See Chapter 7 for a discussion of the FORMAT statement.)
- An INTEGER variable that has been ASSIGNed such a number
- A CHARACTER array name, array element, variable, or constant
- A fixed-length CHARACTER expression
- An asterisk, denoting list-directed I/O

When a format consists of any character entity, the entity must contain the same format list, including outer parentheses, that would appear following the keyword FORMAT in an ordinary FORMAT statement. Only those positions that will actually be referenced during data transfer need be defined. Any data at other positions will be ignored. If an unsubscripted array is used, the format list will be obtained from the concatenation of all its elements. Blanks are of no significance in any type of format list. Widths greater than 255 for the transfer of character data may be specified in this style of format.

For example:

```
READ (UNIT, '(A300)') CHAR_STR
```

A record# is an integer expression. If a record# is present, the READ statement is a direct-access READ; otherwise it is a sequential-access READ. Any file may be read sequentially, but only a file created for direct access (DAM file) can be read by direct access.

If END= label appears, control will transfer to the statement label specified by label (an integer constant) if endfile should occur during the READ. Do not specify END= for a direct-access read.

If ERR= label appears, control will transfer to label if an error should occur during the READ.

If IOSTAT= ios appears, ios (an integer variable) will be set to:

- A positive value if an error occurred
- Zero if the READ executed successfully
- A negative value if endfile was encountered and no error occurred

Note

In an IBM direct READ, unit#'record# must be the first item in the list.

If UNIT= is omitted from a sequential or ANSI direct READ, unit# must be the first item in the list.

If FMT= is omitted from any formatted READ, format must be the second item in the list, and UNIT= must not appear.

In all other cases, the items may appear in any order.

Namelist for READ: Namelist is a method for performing self-labeling input/output. Its distinguishing feature is that a data value never appears alone, but is always labeled with the name of the variable that has the value. On input, namelist allows runtime selection of the variables to the values that will be assigned, and provides free-format assignment similar to that of list-directed I/O. Refer to the end of this section for a complete description of namelist.

Input Lists: An input list is a list of variables, arrays, array elements, and character strings. These data items provide the destination of the data transferred in a READ statement. An input list may be empty, in which case the record is read but skipped. Redundant parentheses may not appear in an input list.

Input lists may contain implied DO loops, to simplify assignment of data to arrays. An implied DO follows the same rules as an ordinary DO. The DO loop control values may have been read in at an earlier stage of the READ statement. Implied DO loops may be nested; for each implied DO, a set of parentheses must exist surrounding it, the array names it references, and any DO loops nested within it. An implied DO must be preceded by a comma.

Array elements not specifically referenced in a READ remain unchanged. If an array name appears without indexes, the computer will generate implied DO loops to scan it in storage order. Assumed-size dummy arrays may not appear in input lists.

Input list examples:

```
DIMENSION ARR(-1:10,-1:10), VEC(5)

READ(1,200) ARR
```

is equivalent to:

```
READ(1,300) ((ARR(I,J), I=-1,10), J=-1,10)
```

WRITE Statement

A WRITE transfers data to a file. The WRITE statement has the following formats:

```
Sequential:    WRITE ([UNIT= ]unit# [,[FMT= ]format] [,ERR= label]
               [,IOSTAT= ios]) [output list]

ANSI direct:   WRITE ([UNIT= ]unit# [,[FMT= ]format] ,REC= record#
               [,ERR= label] [,IOSTAT= ios]) [output list]

IBM direct:    WRITE (unit#'record# [,[FMT= ]format] [,ERR= label]
               [,IOSTAT= ios]) [output list]

NAMELIST:      WRITE ([UNIT=] unit#,blockname)
```

WRITE statements differ from READ statements primarily in the direction of data transfer. The unit#, format, record#, ERR=, and IOSTAT= specifiers have the same significance as in a READ statement. END= is not an option, and ios will never become negative, because endfile cannot occur when a file is written.

The rules governing omission of UNIT= and FMT= are the same as for a READ statement.

Namelist for WRITE: A namelist block may be output with a WRITE statement but not with a PRINT statement. Refer to the end of this section for a complete description of Namelist.

Output lists: An output list has the same form as an input list. The data items in an output list provide the source of the data transferred in a WRITE statement. They must all be defined when the WRITE occurs. An output list may be empty, in which case a null record is written.

Output lists may contain implied DO loops and array names without indexes, which act as they do in input lists. They may also contain expressions. Any CHARACTER expression in an output list must be fixed-length. When the WRITE statement executes, each expression is evaluated and the result written to the file. An expression might consist only of a constant, in which case the constant is written. A format descriptor for an expression must be appropriate to the data type of its final value. (See Chapter 7 for a discussion of format descriptors.) If an output list expression contains function references, invocation of the functions must not change any other value in the expression, either directly or indirectly.

Length Mismatch: When a fixed-length record is written, the output list need not always have the same byte-length as the record. When an attempt is made to write a record too short to hold all the output list

items, an error will occur. When a record longer than necessary to hold the output list is written, the extra positions will be padded with blanks if the WRITE is formatted, or with binary zeroes if the WRITE is unformatted. Padding of extra positions in unformatted DAM file records is an F77 extension; FORTRAN 77 leaves such positions undefined.

Carriage Control: The first character of each record in a file to be printed controls vertical spacing, and is not printed. The remaining characters in a record are printed starting at the left-hand margin. The significance of the permissible carriage-control characters is:

Character	Vertical Spacing Before Printing
Blank	One line
0 (zero)	Two lines
1	To first line of next page
+	No advance (overprint of last line)

Records that contain no characters, generated by slash editing in a FORMAT statement or by an empty output list, cause a blank line to be printed. See Chapter 7 for a discussion on the FORMAT statement.

Unrepresentable Values: If a numeric item cannot be printed in the form required by a format code, the output field will be filled with asterisks.

PRINT Statement

```
PRINT format [,output list]
```

PRINT is a simplified WRITE. It prints the output list at the user terminal according to the format given in format. A PRINT statement will not output a namelist block. The format is as described for READ and WRITE. It is equivalent to:

```
WRITE (1,format) [output-list]
```

For error handling a PRINT acts as a WRITE in which no options were given.

LIST-DIRECTED I/O

Also known as free-format I/O, list-directed I/O occurs when an asterisk appears as the format in a READ, WRITE, or PRINT statement.

When list-directed output occurs, the values in the output list are converted to printable form as directed by FORTRAN-supplied format list defaults. The values are then written to the designated file.

List-directed input is usually employed when data is being read by a program from a free-format device such as the user terminal. A data item for list-directed input must have the same form as a constant of its data type. FORTRAN 77 supplies default format descriptors appropriate to the types of the data items in the input list, and uses those descriptors to convert the data as it is read in. List-directed I/O cannot be used in accessing internal files or DAM files.

This feature also provides a method to indicate in the input data that an item in the input list is to remain unchanged by a READ statement. This is accomplished by using appropriate delimiters.

Delimiters

Adjacent values in a data line for list-directed input must be separated by one or more blanks, a comma, or a slash. Consecutive blanks are equivalent to single blanks. Blanks adjacent to a comma or slash are of no significance. An end-of-record is treated as a blank.

Two adjacent commas with no intervening characters except blanks will leave the corresponding item in the input list unchanged. A slash terminates a read, leaving any remaining items in the input list unchanged. A list-directed read continues until a slash is encountered or all the items in the input list have been satisfied. If there are not enough values to complete the read, an error will occur unless the data is being read from the terminal, in which case the program will wait for the remaining values to be typed in.

Repeat Counts

Repeat counts may modify data items under list-directed input. This format:

```
r*C
```

represents r consecutive occurrences of the input value c. If c is omitted, r null values are read in, leaving the next r elements of the input list unchanged. No blanks may appear between r, *, and c.

Examples:

1. Source line: READ(1,*) A,B,C,D
   Input data: 151,,2*2E2
   Result: A = 151.
   B is unchanged
   C = 2.E2
   D = 2.E2

2. Source line: READ(1,*) I,J,K
   Input data: 5 -3
   Result: I = 5
   J = -3
   K is unchanged

INPUT/OUTPUT ERRORS

If an error occurs during execution of a READ or WRITE (including PRINT statement), execution of the statement terminates and the position of the file pointer becomes indeterminate.

If an error or endfile condition occurs during a READ statement, the data items in the input list and any implied DO index variables become undefined. Data items used solely in subscripts, substring expressions, and implied DO control values do not become undefined.

If an error occurs during a WRITE statement, any implied DO index variables become undefined. The contents of the file remain as they were before execution of the Write began.

If an error occurs during a Read or Write that contains no IOSTAT= or ERR= option, or if endfile occurs during a READ that contains no IOSTAT= or END= option, execution of the program terminates.

NAMELIST-DIRECTED I/O

Namelist Input

To read values into a namelist block, give the name of the block in a READ statement where a format would ordinarily appear.

For example:

```
READ (1,SHIP)
```

When control reaches a namelist READ statement, the program reads values from the designated FORTRAN unit. If values are to be read from the terminal, the program waits for them to be typed in.

Namelist Output

To output the values in a namelist block, give the name of the block in a WRITE statement where the format would ordinarily appear. Namelist is not designed to be used with a PRINT statement.

For example:

```
WRITE (1,SHIP)
```

An output group will then be printed, giving the values of all the variables in the namelist block. Each value will be labeled with the appropriate variable name. Values of an array are printed consecutively, separated by commas.

An output group has the format:

```
$namelist_block_name
name = value, [value,]...
.
.
.
$END
```

For example, the lines:

```
NAMELIST /SHIP/ I,K,SPEED
DIMENSION K(3)
DATA I/32/, K/1,2,3/, SPEED/40.0/
WRITE (1,SHIP)
```

would produce the output:

```
$SHIP
I = 32,
K = 1, 2, 3,
SPEED = 4.000000E+01,
$END
```

When an uninitialized namelist variable is output, no error will occur, but the value printed will be meaningless.

Input Groups

Values input under namelist must be presented in an input group. An input group has the same syntax as an output group. However, not all the variables in a namelist block need be referenced in an input group for that block, and the references may appear in any order. When the input group is complete, as indicated by a $END, program execution resumes.

A data value input under namelist has the same form as a constant of its data type. When the type of a value does not match that of the variable to which it is assigned, type conversion occurs as in an ordinary assignment statement.

For example, the lines:

```
NAMELIST /SHIP/ I,K,SPEED
DIMENSION K(3)
WRITE (1,*) 'ENTER DATA:'
READ (1,SHIP)
WRITE (1,*) 'THE DATA IS:'
WRITE (1,SHIP)
```

could result in the following sequence:

```
ENTER DATA:
$SHIP
K(2) = 1,
SPEED = 30,
K(1) = 2,
$END
THE DATA IS:
$SHIP
I = 0,
K = 2, 1, 0,
SPEED = 3.000000E+01,
$END
```

Input Group Format: An input group may be given as shown above, given in a single line, or subdivided into multiple lines as desired. A token (keyword, name, number, or quoted string) must not be broken across an end-of-line boundary. (A complex number may be broken at the comma between its parts.)

Thus the input groups:

```
$SHIP K(2) = 1, SPEED = 30, K(1) = 2, $END
```

and

```
$SHIP
K(2) =
1, SPEED = 30, K
(1) = 2
,$END
```

are both equivalent to the input group above, while the group below is illegal.

```
$SHIP
K(2) = 1, SP
EED = 30,
K(1) = 2,
$END
```

Inputting Arrays With Namelist

Several methods that make it easier to assign values to an array using namelist are illustrated below with examples. The examples are outputs of the following program:

```
      PROGRAM NLIST
      NAMELIST /IOTA/ ARR
      INTEGER ARR(10)
C
C INITIALIZE THE ARRAY
C
100   DO 200  I = 1,10
      ARR(I) = 0
200   CONTINUE
C
C READ A SET OF VALUES
C
      WRITE (1,*) 'ENTER DATA:'
      READ (1,IOTA)
C
C WRITE THE VALUES
C
      WRITE (1,*) 'THE DATA IS:'
      WRITE (1,IOTA)
```

```
C
C DO IT AGAIN
C
      GO TO 100
300   STOP
      END
```

Omitted Values: Consecutive commas in an assignment to an array cause the corresponding array elements to be skipped over.

```
ENTER DATA:
$IOTA
ARR = 1,,,,3,,,,5,
$END
THE DATA IS:
$IOTA
ARR = 1,0,0,0,3,0,0,0,5,0,
$END
```

Ellipses of Indexes: When a contiguous subset of an array is to be assigned values, the subset may be indicated by giving its bounds separated by an ellipsis.

```
ENTER DATA:
$IOTA
ARR(2...6) = 1,3,5,7,9,
$END
THE DATA IS:
$IOTA
ARR = 0,1,3,5,7,9,0,0,0,0,
$END
```

Repetition of Values: When consecutive elements of an array are to have the same value, the value may be given once with a repeat count:

```
ENTER DATA:
$IOTA
ARR = 5*1, 4*2,
$END
THE DATA IS:
$IOTA
ARR = 1,1,1,1,1,2,2,2,2,0,
$END
```

Scalar Promotion to an Array: When all the elements of an array are to be assigned the same value, the value may be given as a scalar.

```
ENTER DATA:
$IOTA
ARR = 5,
$END
THE DATA IS:
$IOTA
ARR = 5,5,5,5,5,5,5,5,5,5,
$END
```

Scalar promotion to a contiguous subset of an array is also possible:

```
ENTER DATA:
$IOTA
ARR(2...6) = 5,
$END
THE DATA IS:
$IOTA
ARR = 0,5,5,5,5,5,0,0,0,0,
$END
```

Errors When Using Namelist

If an error occurs during namelist I/O, and no ERR= has been supplied in the READ or WRITE statement, the program will:

1. Print the offending line
2. Print a caret under the offending token
3. Print an error message
4. Exit to PRIMOS

Restriction on Namelist

Namelist cannot be used to access an array whose bounds are dynamically declared. That is,

```
SUBROUTINE BARB (A,N)
DIMENSION A(N)
NAMELIST /FOB/ A
```

will cause an error at compile time.

SUMMARY OF STATEMENT SYNTAX

In Table 6-4, all input/output statements are listed in alphabetical order with their syntax requirements. The table is intended only as a reminder for those already familiar with the statements.

Table 6-4
Input/Output Statement Syntax

Statement	Syntax
BACKSPACE	BACKSPACE ([UNIT=]unit# [,IOSTAT=ios] [,ERR=label])
CLOSE	CLOSE ([UNIT=]unit# [,STATUS= stat] [,ERR= label] [,IOSTAT= ios])
ENDFILE	ENDFILE ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])
INQUIRE	INQUIRE ([FILE=]filename or [UNIT=]unit# [,ERR= s] [,EXIST= ex] [,OPENED= od] [,NUMBER= num] [,NAMED= nmd] [,NAME= fn] [,ACCESS= acc] [,SEQUENTIAL= seq] [,DIRECT= dir] [,FORM= fm] [,FORMATTED= fmt] [,UNFORMATTED= unf] [,RECL= rcl] [,NEXTREC= nr] [,BLANK= blnk] [,IOSTAT= ios])
NAMELIST:	Read ([Unit=]unit#,blockname)
NAMELIST:	WRITE ([UNIT=] unit#,blockname)
OPEN	OPEN ([UNIT=]unit# [,FILE= filename] [,STATUS=stat] [,ACCESS= acc] [,FORM= fm] [,RECL= reclength] [,BLANK= blnk] [,ERR= label] [,IOSTAT= ios] [,ACTION= act]
PRINT	PRINT format [,output list]
READ direct ANSI:	READ ([UNIT=]unit# [,[FMT=]format],REC= record# [,END=label][,ERR=label] [,IOSTAT=ios])[input list])
READ direct IBM:	READ (unit#'record# [,[FMT=]format] [,END= label] [,ERR= label] [,IOSTAT= ios]) [input list])
READ Sequential:	READ format [,input list] READ ([UNIT=]unit# [,[FMT=]format] [,END= label] [,ERR= label] [,IOSTAT= ios]) [input list])
REWIND	REWIND ([UNIT=]unit# [,IOSTAT= ios] [,ERR= label])
WRITE direct ANSI:	WRITE ([UNIT=]unit# [,[FMT=]format] ,REC= record# [,ERR= label] [,IOSTAT= ios]) [output list])
WRITE direct IBM:	WRITE (unit#'record# [,[FMT=]format] [,ERR= label] [,IOSTAT= ios]) [output list])
WRITE Sequential:	WRITE ([UNIT=]unit# [,[FMT=]format] [,ERR= label] [,IOSTAT= ios]) [output list])

7
Format Statements

The FORMAT statement is used to direct the input and output of data being read or written in your program via READ, WRITE, or PRINT statements. This chapter discusses using the FORMAT statement to describe how data is to be organized coming from or going to a file. Included in this discussion are:

- Format and I/O list interaction
- Format list rescanning
- Field descriptors
- Edit-control descriptors

For information on Input/Output, see Chapter 6, I/O STATEMENTS.

FORMAT STATEMENT

Formatted data transfer occurs when a format is given in a READ, WRITE or PRINT data transfer statement. Most often, the format is the statement number of a FORMAT statement. The other possibilities are discussed under the READ Statement in Chapter 6.

In the following discussion, the term "I/O list" means either an input list or an output list.

A FORMAT statement has the following format:

```
label FORMAT (d [,d]...)
```

where:

label is the mandatory statement label.

d is a field descriptor or an edit-control descriptor.

The parenthetical list of descriptors is known as a format list. Blanks are of no significance in a format list. Parentheses may appear inside a format list to delineate group repeat counts.

Field Descriptors control the data conversion process during data transfer. For each item in the I/O list, an appropriate field descriptor must be given. Data moving to or from the data item is converted as specified by the corresponding descriptor. A field descriptor in a FORMAT statement cannot specify a width that is greater than 255.

Edit-control Descriptors control more general aspects of the formatting process, such as scale factors, tab control, and the optional printing of literal character items to label the output.

Repeat Counts are integer constants prefixed to a field descriptor, or to a parenthetical portion or the entirety of a format list. Individual edit-control descriptors cannot have repeat counts. As data transfer proceeds, the format list items modified by the repeat count will be reused the number of times specified before format control proceeds to subsequent format list items. Repeat counts have a maximum nesting of 10 levels.

FORMAT AND I/O LIST INTERACTION

During data transfer, the format list is scanned from left to right, except as modified by repeat counts. The I/O list is also scanned from left to right.

When an edit-control descriptor is encountered in a format list, the action or alteration required by it is performed. When a field descriptor is encountered, the next I/O list item is edited appropriately and transmitted. If no I/O list items remain when an edit-control descriptor is encountered, data transfer terminates.

When the colon edit-control descriptor is encountered, data transfer terminates if no I/O list items remain to be transmitted; otherwise data transfer continues.

An empty format list may be given to correspond to an empty I/O list.

FORMAT LIST RESCANNING

If the format list is exhausted before the I/O list, the file pointer is positioned at the beginning of the next record; format control then reverts to the beginning of the portion of the format list that was terminated by the last preceding right parenthesis. If there is no such parenthesis, format control reverts to the beginning of the format list. Any repeat count preceding the rescanned format is reused. On output, the current record is padded with blanks and a new record started. On input, the remainder of the current record is skipped, and the file pointer advanced to the beginning of the next record. Reversion of format control, of itself, has no effect on the scale factor, the sign control (S, SP, SS), or the blank control (BN, BZ) in effect at the time of reversion.

FIELD DESCRIPTORS

A field descriptor mediates the conversion of a data item between internal and external form. Usually, the data is supplied by the I/O list. In a character constant field descriptor, it is contained in the descriptor itself.

Numeric Descriptors

The numeric descriptors are I, F, E, D, Q, and G. Unless specified otherwise or modified by edit-control descriptors, the following rules apply to all numeric descriptors:

- Leading blanks are not significant for input. For output, leading 0's are suppressed. A minus sign is printed for a negative number, but a positive number is left unsigned.
- For input with F, E, D, Q, and G descriptors, a decimal point in the input field overrides the d specification in the descriptor.
- If the field width is insufficient to represent the number, then the field width specified is filled with asterisks.
- Excess digits of precision may be specified on input to non-INTEGER numeric data types. The excess will be ignored.
- See the BLANK= option of the OPEN statement for the rules concerning blanks in input fields.

The numeric descriptors are described in the following sections.

Integer Editing (I): Used to edit a short or long integer. The I numeric descriptor has the following format:

Iw[.m]

where:

w is the size of the external field, including blanks and a sign.

m is the minimum number of places to be displayed on output. Leading 0's will be printed if necessary to fill the field. For input, m has no effect.

Real Editing (Nonexponential (F)): Writes a real number without an exponent. Reads any real, double precision, or REAL*16 number. The F edit descriptor has the following format:

Fw.d

where:

w is the size of the field, including blanks, the sign, and the decimal point.

d is the number of places to the right of the decimal point.

The following rules apply to the F edit descriptor:

On input: The decimal point may be omitted from the field. The rightmost d digits will be interpreted as decimal digits. If a decimal point is present, its position overrides d. Input fields appropriate for E and D editing will also work for F editing.

On output: d decimal positions are always written.

Real Editing (Exponential (E)): Edits a real or double precision number with an exponent. The E descriptor has the following format:

Ew.d[Ee]

where:

w is the size of the external field, including an exponent and its sign.

d is the number of decimal places. On input, an explicit decimal point overrides d.

e is the number of exponent digits to be displayed on output. It is ignored for input. When Ee is omitted from an E field descriptor used for output, the defaults listed below under output will apply.

The following rules apply to the E descriptor:

On input: The exponent may be omitted. E+00 will be assumed.

On output: If Ee is present, e digits of the exponent will be printed. If Ee is omitted, the appearance of the exponent will be as follows:

Value of Exponent	Appearance of Exponent
$-99 \leq exp \leq 99$	E ± zz
$-999 \leq exp < -99$	-zzz (no "E")
$99 < exp \leq 999$	+zzz (no "E")
$-9999 \leq exp < -999$	****
$999 < exp \leq 9999$	****

Note that the number is always normalized. For output that is not normalized scalars should be used.

Double Precision Editing (D): Edits a double precision number. The D descriptor has the following format:

Dw.d

The following rules apply to the D descriptor:

On input: Operates exactly like an E descriptor.

On output: Operates exactly like an E descriptor with no Ee present, except that a D is substituted wherever an E would appear in the output field. For explicit control of double precision exponent format, output the number with an Ew.dEe descriptor.

REAL*16 Editing (Q): Used to edit a REAL*16 number. The Q descriptor has the following format:

Qw.d

REAL*16 editing is the same as double precision editing except that a Q is used in place of a D.

Complex Editing: A complex number consists of a pair of real or double precision numbers. It is edited with an appropriate pair of real or double precision field descriptors. The fact that the two numbers form one entity mathematically is irrelevant to input/output. Edit-control descriptors may appear between the two field descriptors.

General Editing (G): Edits real data where the magnitude of the data is not known beforehand. The G descriptor has the following format:

Gw.d[Ee]

where:

w, d, and e are as defined for the F descriptor.

General editing produces the more readable F format when possible, but converts to E format when the magnitude of the number exceeds F format representational limits.

The following rules apply to the G descriptor:

On input: The G descriptor is equivalent to the F descriptor.

On output: The G descriptor acts as follows:

Magnitude (M) of Real Data Item	G Descriptor Acts As
0.1 <= M < 1	F(w-n).d, n('b')
1 <= M < 10	F(w-n).(d-1), n('b')
10 <= M < 100	F(w-n).(d-2), n('b')
.	.
.	.
.	.
10**(d-2) <= M < 10**(d-1)	F(w-n).1, n('b')
10**(d-1) <= M < 10**(d)	F(w-n).0, n('b')
Otherwise	Ew.d[Ee]

where b is a blank and n is 4 for Gw.d and e+2 for Gw.dEe.

If M < .01 or M >= 10**d, then Gw.d is equivalent to kPEw.d, where k is the current scale factor.

For input, the Gw.dEe field descriptor is treated identically to the Gw.d descriptor. For output, the Gw.dEe acts as Fw.dEe if 0.1 <= M < 10**d, and acts as Ew.dEe otherwise.

Nonnumeric Descriptors

L, A, X, B, and format-list character constants are nonnumeric descriptors, that is, they are not number oriented edit descriptors.

Logical Editing (L): Edits logical data, transmitting a logical value to or from an I/O list entity. The L descriptor has the following format:

```
Lw
```

where:

w is the width of the field.

The following rules apply to the L descriptor:

On input: A valid input field consists of optional blanks, optionally followed by a decimal point, followed by a T or an F. The T or F may be followed by additional characters in the field, but they will be ignored.

On output: The output field consists of w-1 blanks followed by a T or F, as the value of the internal datum is true or false, respectively.

Character Editing (A): Used for transferring character or Hollerith values. The A descriptor has the following format:

```
A[w]
```

where:

w is the width of the field, and must not be greater than 255 characters when appearing in a FORMAT statement. It is required when inputting or outputting Hollerith data, and optional when inputting or outputting data from a variable of type CHARACTER.

In the following general rules that apply to the A descriptor, L is the length of the character item being edited.

On input: If w >= L, the rightmost L characters are taken from the external input field. If w < L, the w characters are left justified in the data item and padded with blanks.

On output: If w > L, the characters are printed right justified in the field, preceded by blanks as needed. If w <= L, the leftmost w characters are printed. If w is not specified it is assumed to be equal to L.

Character Constant Editing: Used for transmitting character and Hollerith constant data. The apostrophe edit descriptor and the Hollerith edit descriptor have the following format:

'ccc...c' or nHccc...c

where:

Each c is any character in Prime ECS (not necessarily a member of the F77 character set).

n is the number of characters in the character constant.

Character strings in either of these formats may appear as constants in an output format list. Such a string contains its own data, eliminating the need for a corresponding item in the output data list. When the string is encountered during the scan of the format list, the characters it contains are written to the current record. A character constant may not appear in a format list used for input, and may not be modified by an individual repeat count.

Note

FORTRAN 66 permitted data to be read into an H format field, altering the value it would print when the format list involved was later used for output. FORTRAN 77 will not accept this practice.

Space Skipping (X): Used for skipping one or more character positions. The X descriptor has the following format:

nX

where:

n is an integer. On output, equivalent to a character constant of n blanks. On input, equivalent to the positional edit descriptor, TRn, which is explained later in this chapter. No repeat count may appear.

Business Editing (B): The B descriptor is used in printing business reports where it is desirable to fill number fields to prevent unauthorized modifications, as on checks, suppress leading 0's and plus signs, print trailing minus signs (accounting convention), and convert minus signs to CR for indicating credit entries on bills. Business editing is an F77 extension.

B 'string'

where:

The length of string determines the field width. If the width is too small for the number, then the output will be a string of asterisks filling the field. Valid characters for the string are:

+ - $, * Z # . CR

The use of the valid characters is explained below.

- Plus (+):

  If only the first character is +, then the sign of the number (+ or -) is printed in the leftmost portion of the field (Fixed sign). If the string begins with more than one + sign, then these will be replaced by asterisks 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 (-):

  The minus sign behaves the same as a plus sign except that a space (blank) is printed instead of a + if the number is positive (Plus sign suppression).

- Dollar sign ($):

  A dollar sign $ may at most be preceded in the string by an optional fixed sign. A single dollar sign will cause a $ to be printed in the corresponding position in the output field (Fixed dollar).

  Multiple dollar signs will be replaced by printing characters in the number, and a single $ will be printed in the position immediately to the left of the leftmost printing character of the number (Floating dollar).

- Asterisk (*):

  Asterisks may be preceded only by an optional fixed sign and/or a fixed dollar. Asterisks in positions used by digits of the number will be replaced by those digits. The remainder will be printed as asterisks (Field filling).

- Zed (Z):

  If the digit corresponding to a Z in the output number is a leading 0, a space (blank) will be printed in that position; otherwise the digit in the number will be printed (Leading-0 suppression).

- Number sign (#):

  #'s indicate digit positions not subject to leading-0 suppression; the digit in the number will be printed in its corresponding portion whether 0 or not (zero nonsuppression).

- Decimal point (.):

  A 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 comma 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. In that case 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, two spaces will be printed following it. If negative, the letters CR will be printed.

See Table 7-1 for examples of B-Format usage.

Table 7-1
Examples of B-Format Usage

Number	Format	Output Field
123	B'####'	0123
12345	B'####'	****
0	B'####'	0000
123	B'ZZZZ'	123
1234	B'ZZZZ'	1234
0	B'ZZZZ'	
0	B'ZZZ#'	0
1.035	B'#.##'	1.04
0	B'#.##'	0.00
1234.56	B'ZZZ,ZZZ,ZZ#.##'	1,234.56
123456.78	B'ZZZ,ZZZ,ZZ#.##'	123,456.78
0	B'ZZZ,ZZZ,ZZ#.##'	0.00
2	B'+###'	+002
-2	B'+###'	-002
2	B'-ZZ#'	2
-2	B'-ZZ#'	- 2
234	B'ZZZZZ+'	234+
-234	B'ZZZZZ+'	234-
234	B'ZZZZZ-'	234
-234	B'ZZZZZ-'	234-
12345	B'ZZZ,ZZ#CR'	12,345
-12345	B'ZZZ.ZZ#CR'	12,345CR
123	B'+++,++#.##'	123.00
-123	B+++,++#.##'	-123.00
98	B'$ZZZZZZ#'	$ 98
98	B'$$$$$$$#'	$98
156789	B'$***,***,**#.##'	$****156,789.00

EDIT-CONTROL DESCRIPTORS

Edit-control descriptors control general aspects of the formatting process. They differ from field descriptors in that they do not correspond to or supply individual data items, but modify the environment in which the data transfer process occurs.

Scale Factors (P)

The use of a scale factor allows you to move the location of the decimal point in real numbers. The P descriptor has the following format:

kP

where:

scale factor k is an unsigned or negative integer constant. The comma following a scale factor is often omitted, so that it becomes a prefix of a subsequent field descriptor. The scale factor has various effects, depending on the descriptor type and the direction of data transfer.

The following rules apply to the P descriptor:

On F, E, D, Q, and G input: If there is an exponent in the field, the scale factor has no effect. Otherwise, it converts the data so that:

External Value = Internal Value*(10**k)

On F output: The scale factor converts the value as for F input.

On E, D, and Q output: The mantissa is multiplied by 10**k and the exponent is reduced by k to maintain the same overall value. This permits output of E, D, and Q numbers in non-normalized form.

G Output: If the G is acting as an F, the scale factor is ignored. If it is acting as an E, the scale factor behaves as described for E output.

Note

Once a scale factor has been used, it remains in effect for all subsequent descriptors of appropriate type, until it is reset to another value or to 0. When a format list is rescanned, the scale factor is not reset to 0 automatically. If a scale factor is to affect only one field, "0P" must appear before the next scalable descriptor that occurs.

Sign Control Editing (SP,SS,S)

The SP, SS, and S descriptors control the placement of plus signs in numeric output. Once a sign control descriptor is encountered, it remains in effect until it is explicitly altered or revoked. These descriptors have the following format:

SP SS S

Each effect of each descriptor is explained below.

SP: The processor will insert a plus sign wherever one may optionally appear.

SS: The processor will not insert any plus sign whose appearance is optional.

S: The processor will return to the locally defined system default for sign editing.

Blank Control Editing (BN,BZ)

The BN and BZ edit descriptors can be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. The method of handling blanks in numeric input fields that is established for a file by the BLANK= option of the OPEN statement may be temporarily overridden by BN or BZ. The method may be altered as often as desired, and will revert to the BLANK= value when the READ statement is complete. Blank control descriptors have no effect on output. The BN and BZ descriptors have the following format:

BN BZ

The effects of the blank control descriptors are explained below.

BN: All blanks will be deleted, and digits will be compressed to the right side of the input field. An all-blank field is interpreted as a 0 value.

BZ: All but leading blanks will be converted to zeros, as in FORTRAN 66.

Positional Editing (T)

The T edit-control descriptors are used to set tab positions in the current file record. They have the following format:

Tn TLn TRn

where:

n is an integer constant less than or equal to 255.

The following description presupposes that you have read about the I/O buffer F$IOBF at the beginning of the subsection on DATA TRANSFER STATEMENTS in Chapter 6. The pointer that scans F$IOBF during data transfer ordinarily behaves as follows:

1. Before data transfer, it points to the first position (byte) of F$IOBF.

2. While an F$IOBF position is being read or written, it points to that position.

3. After a position has been read or written, it moves to the next position to the left and remains there.

4. After the last F$IOBF position has been read or written, it remains at that position.

Note that this behavior is the same as that of the carriage position on an ordinary typewriter.

The T edit-control descriptor is used to alter the sequential progress of the F$IOBF pointer. The pointer can be moved to the left or right of its current position, or to an absolute position, in any desired sequence. Subsequent data transfers will begin at the new position. Thus F$IOBF positions, and hence the corresponding current-record positions, can be accessed as often as desired and in any order.

If an attempt is made to move the F$IOBF pointer beyond the first (or last) F$IOBF position, the pointer will stop and remain at that position. If T descriptors are used during a WRITE in such a way that some F$IOBF positions remain undefined after all data items have been transferred, the undefined positions will be filled with blanks before F$IOBF is written to the current file record.

Moving the F$IOBF pointer has no effect on the file pointer, which never skips positions within a record. Beware of confusing these two pointers.

The effect of using the positional descriptors is explained below.

TLn: Move the F$IOBF pointer n positions left.

TRn: Move the F$IOBF pointer n positions right.

Tn: Move the F$IOBF pointer to the nth character of the record.

Conditional Output

A colon (:) in the format list causes data transfer to end at that point if no items remain in the output list. This feature increases the versatility of a format list that contains character constant descriptors used in labeling the output. A colon is ignored on input.

Record Skipping

The slash (/) in a FORMAT statement indicates the end of data transfer on the current record. Format of the slash edit descriptor:

/[/]...

A slash in a format list causes I/O processing to go to the next record. As many new records will be begun as there are slashes. The effect of slashes at the beginning or end of a format list is additional to the automatic beginning of a new record with each data transfer statement.

The following general rules apply to the slash edit descriptor:

On input: Under sequential access, a slash causes the remaining portion of the current record to be skipped. The file pointer is positioned at the beginning of the next record, making it the current record. Under direct access, the remainder of the record is skipped, the record number increased by one, and the file pointer positioned at the beginning of the record that has that record number.

On output: Use of the slash is similar to input, except that all positions skipped over will be filled with blanks.

Commas adjacent to slashes may be omitted.

8

Subroutines and Functions

This chapter discusses how to create and use your own functions and subroutines, as well as those supplied by the F77 compiler. In addition to one main program, a FORTRAN 77 program may contain any number of functions and subroutines, collectively called subprograms. There are five categories of subprograms in FORTRAN 77:

1. Intrinsic functions
2. Statement functions
3. External functions
4. Subroutines
5. Block data

F77 INTRINSIC FUNCTIONS

FORTRAN 77 supplies you with a library of a wide variety of intrinsic (built-in) functions. You can use these functions for type conversion, character data evaluation, lexical comparison, and the calculation of various mathematical quantities.

The F77 intrinsic function set includes:

- All FORTRAN 77 intrinsics
- Additional functions for bitwise logical operations
- Bitwise shifts
- Truncation of an integer
- Determination of a data item's storage address
- Operations on the `REAL*16` and `COMPLEX*16` data types

Intrinsic Function Tables

Tables 8-1 through 8-7 provide a complete list all F77 intrinsic functions by category:

Table	Intrinsic Function
8 - 1	Logarithmic and Exponential
8 - 2	Trigonometric
8 - 3	Hyperbolic
8 - 4	Mathematical
8 - 5	Conversion and Maximum/Minimum
8 - 6	Character Manipulation
8 - 7	Bit Manipulation

Where a specific F77 function has the same name as an existing FTN function, the functions are the same, except as noted under Reimplemented FTN Constructs in Appendix C.

Before using any function with which you are not completely familiar, be sure to study carefully the table entry and accompanying notes, if any, for that function. Notes for Tables 1-7 immediately follow those tables.

Since all F77 intrinsic functions are built into the language, you can invoke an F77 intrinsic function at any point in any F77 program unit. The F77 compiler and the BIND linking loader will automatically supply the functions you invoke. No additional action is required.

Referencing an Intrinsic Function

To invoke an intrinsic function, you use the function name followed by the arguments on which you want it to act within an expression. After the invoked function completes its calculations, the function name is replaced by the value from the calculation.

For example, in the assignment statement

```
X = SQRT(A + B)
```

SQRT is the instrinsic function name. The purpose of this function is to determine the square root of the value of the expression A + B.

For information on declaring certain intrinsic functions INTRINSIC in a program unit, see Chapter 3.

Generic and Specific Functions

Many FORTRAN 77 intrinsic functions are generic. They exist in several versions, called specific functions, which differ only in the data type of the argument each accepts. Both generic and specific functions are listed in Tables 8-1 through 8-7. When you reference a generic function, the F77 compiler will examine the argument list at the reference and select the specific function appropriate to the data type of the arguments.

All arguments for either generic or specific functions must be of the appropriate data type. If not, the compiler will signal an error.

Not all specific functions are individually named. Those that are may be invoked directly by name, in which case you must be careful to supply the correct data types.

Intrinsic Functions as Arguments

Only named specific functions can be passed as arguments to subprograms. In some cases, a specific function has the same name as its generic function. When this name appears in an argument list, it is the specific function that is passed.

The following intrinsic functions cannot be passed as arguments:

- Type conversion
- Selection of a maximum or minimum value
- Lexical comparison

- Logical operation
- Shifting or truncation of bits

For more information on passing arguments with intrinsic functions, see the Subprograms as Arguments section later in this chapter.

Long and Short Integer Arguments to Intrinsic Functions

All new programs you write in F77 should use long integers exclusively, in conformance with the ANSI standard. The use of short integers in an F77 program unit may become necessary when:

- You convert program units from FORTRAN IV to F77, or,
- You write F77 program units which will return values to an existing FORTRAN IV program unit.

No constraint on the use of short integers is imposed by the F77 intrinsic set. All F77 intrinsic functions have been extended to accept either long or short arguments, or a mixture of the two, and to produce short integer results where appropriate. ANSI FORTRAN 77 does not provide short integers or permit data types to be mixed in an intrinsic function's argument list.

An intrinsic function that produces an integer result (an integer intrinsic) will produce either a long or short integer. For integer intrinsics other than INT whose arguments are integers, the result type depends on the argument list at the particular invocation. For integer intrinsics whose arguments are not integers, and for INT, the result type depends on the compiler option (-INTS or -INTL) in effect when the program unit containing the intrinsic was compiled. The notes for the tables on intrinsic functions tell how the result type for each integer intrinsic is determined.

Table 8-1
Intrinsic Functions: Logarithmic and Exponetial

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Natural Logarithm	log(a)	1	LOG	ALOG	Real	Real	18
				DLOG	Double	Double	
				QLOG	Real*16	Real*16	
				CLOG	Complex	Complex	
				CDLOG	Complex*16	Complex*16	
Common Logarithm	log10(a)	1	LOG10	ALOG10	Real	Real	18
				DLOG10	Double	Double	
				QLOG10	Real*16	Real*16	
Exponential	e**a	1	EXP	EXP	Real	Real	
				DEXP	Double	Double	
				QEXP	Real*16	Real*16	
				CEXP	Complex	Complex	
				CDEXP	Complex*16	Complex*16	

Table 8-2
Intrinsic Functions: Trigonometric

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Sine	sin(a)	1	SIN	SIN	Real	Real	19,21
				DSIN	Double	Double	
				QSIN	Real*16	Real*16	
				CSIN	Complex	Complex	
				CDSIN	Complex*16	Complex*16	
Cosine	cos(a)	1	COS	COS	Real	Real	19,21
				DCOS	Double	Double	
				QCOS	Real*16	Real*16	
				CCOS	Complex	Complex	
				CDCOS	Complex*16	Complex*16	
Tangent	tan(a)	1	TAN	TAN	Real	Real	19,21
				DTAN	Double	Double	
				QTAN	Real*16	Real*16	
Arcsine	arcsin(a)	1	ASIN	ASIN	Real	Real	20,22
				DASIN	Double	Double	
				QASIN	Real*16	Real*16	
Arccosine	arccos(a)	1	ACOS	ACOS	Real	Real	20,23
				DACOS	Double	Double	
				QACOS	Real*16	Real*16	
Arctangent	arctan(a)	1	ATAN	ATAN	Real	Real	20,24
				DATAN	Double	Double	
				QATAN	Real*16	Real*16	
	arctan(a1/a2)	2	ATAN2	ATAN2	Real	Real	20,24
				DATAN2	Double	Double	
				QATAN2	Real*16	Real*16	

Table 8-3
Intrinsic Functions: Hyperbolic

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Hyperbolic Sine	sinh(a)	1	SINH	SINH	Real	Real	19
				DSINH	Double	Double	
				QSINH	Real*16	Real*16	
Hyperbolic Cosine	cosh(a)	1	COSH	COSH	Real	Real	19
				DCOSH	Double	Double	
				QCOSH	Real*16	Real*16	
Hyperbolic Tangent	tanh(a)	1	TANH	TANH	Real	Real	19
				DTANH	Double	Double	
				QTANH	Real*16	Real*16	

Table 8-4
Intrinsic Functions: Mathematical

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Absolute Value	(a**2)**.5	1	ABS	IABS	Integer	Integer	10,16
				ABS	Real	Real	
				DABS	Double	Double	
				QABS	Real*16	Real*16	
				CABS	Complex	Real	
				CDABS	Complex*16	Double	
Square Root	(a)**.5	1	SQRT	SQRT	Real	Real	17
				DSQRT	Double	Double	
				QSQRT	Real*16	Real*16	
				CSQRT	Complex	Complex	
				CDSQRT	Complex*16	Complex*16	
Real Part of Complex Argument	ar	1	-	REAL	Complex	Real	15,16
				DREAL	Complex*16	Double	
Imaginary Part of Complex Argument	ai	1	-	AIMAG	Complex	Real	16
				DIMAG	Complex*16	Double	
Conjugate of Complex Argument	(ar,-ai)	1	CONJG	CONJG	Complex	Complex	16
				DCONJG	Complex*16	Complex*16	
Truncation	REAL(INTL(a)) DBLE(INTL(a))	1	AINT	AINT	Real	Real	
				DINT	Double	Double	
				QINT	Real*16	Real*16	

Table 8-4 (continued)
Intrinsic Functions: Mathematical

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Nearest Whole Number	See Note 8	1	ANINT				
				ANINT	Real	Real	8
				DNINT	Double	Double	
				QNINT	Real*16	Real*16	
Nearest Integer	See Note 9	1	NINT	NINT	Real	Integer	9,32
				IDNINT	Double	Integer	
				IQNINT	Real*16	Real*16	
Modulo Arithmetic	See Note 11	2	MOD	MOD	Integer	Integer	11,33
				AMOD	Real	Real	
				DMOD	Double	Double	
				QMOD	Real*16	Real*16	
Transfer of Sign	|a1| if a2 >= 0 -|a1| if a2 < 0	2	SIGN	ISIGN	Integer	Integer	12,33
				SIGN	Real	Real	
				DSIGN	Double	Double	
				QSIGN	Real*16	Real*16	
Positive Difference	a1-a2 if a1 > a2 0 if a1 <= a2	2	DIM	IDIM	Integer	Integer	33
				DIM	Real	Real	
				DDIM	Double	Double	
				QDIM	Real*16	Real*16	
Double Precision Product	a1*a2	2	-	DPROD	Real	Double	
				QPROD	Double	Real*16	

Table 8-5
Intrinsic Functions: Conversion and Maximum/Minimum

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Type Conversion	Numeric to Integer	1	INT	-	Integer	Integer	1,32,34
				INT	Real	Integer	
				IFIX	Real	Integer	
				IDINT	Double	Integer	
				IQINT	Real*16	Integer	
				-	Complex	Integer	
				-	Complex*16	Integer	
	Numeric to Short Integer	1	INTS	-	Integer	Integer*2	2,34
				-	Real	Integer*2	
				-	Double	Integer*2	
				-	Real*16	Integer*2	
				-	Complex	Integer*2	
				-	Complex*16	Integer*2	
	Numeric to Long Integer	1	INTL	-	Integer	Integer*4	2,34
				-	Real	Integer*4	
				-	Double	Integer*4	
				-	Real*16	Integer*4	
				-	Complex	Integer*4	
				-	Complex*16	Integer*4	
	Numeric to Real	1	REAL	FLOAT	Integer	Real	3,34
				-	Real	Real	
				SNGL	Double	Real	
				-	Real*16	Real	
				-	Complex	Real	
				REAL	Complex*16	Real	
	Numeric to Double Precision	1	DBLE	-	Integer	Double	4,34
				-	Real	Double	
				-	Double	Double	
				DBLEQ	Real*16	Double	
				-	Complex	Double	
				DREAL	Complex*16	Double	

Table 8-5 (continued)
Intrinsic Functions: Conversion and Maximum/Minimum

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
	Numeric to Real*16	1	QUAD	-	Integer	Real*16	4,34
				QEXT	Real	Real*16	
				QEXTD	Double	Real*16	
				-	Real*16	Real*16	
				-	Complex	Real*16	
				-	Complex*16	Real*16	
	Numeric to Complex	1 or 2	CMPLX	-	Integer	Complex	5,34
				-	Real	Complex	
				-	Double	Complex	
				-	Real*16	Complex	
				-	Complex	Complex	
				-	Complex*16	Complex	
	Numeric to Complex*16	1 or 2	DCMPLX	-	Integer	Complex*16	6,34
				-	Real	Complex*16	
				-	Double	Complex*16	
				-	Real*16	Complex*16	
				-	Complex	Complex*16	
				-	Complex*16	Complex*16	
Choosing Largest Value	max(a1,a2,...)	>= 2	MAX	MAX0	Integer	Integer	33,34
				AMAX1	Real	Real	
				DMAX1	Double	Double	
				QMAX1	Real*16	Real*16	
				AMAX0	Integer	Real	32,34
			-	MAX1	Real	Integer	
Choosing Smallest Value	min(a1,a2,...)	>= 2	MIN	MIN0	Integer	Integer	33,34
				AMIN1	Real	Real	
				DMIN1	Double	Double	
				QMIN1	Real*16	Real*16	
				AMIN0	Integer	Real	32,34
			-	MIN1	Real	Integer	

Table 8-6
Intrinsic Functions: Character Manipulation

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Convert Character	Character to Integer	1	-	ICHAR	Character	Integer	7,32
Convert Integer	Integer to Character	1	-	CHAR	Integer	Character	7
Length	Length of Character Entity	1	-	LEN	Character	Integer	13,32
Index of Substring	Location of Substring a2 in String a1	2	-	INDEX	Character	Integer	14,32
Lexically > =	a1 >= a2	2	-	LGE	Character	Logical	25,34
Lexically >	a1 > a2	2	-	LGT	Character	Logical	25,34
Lexically < =	a1 <= a2	2	-	LLE	Character	Logical	25,34
Lexically <	a1 < a2	2	-	LLT	Character	Logical	25,34

Table 8-7
Intrinsic Functions: Bit Manipulation

Class of Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function	Notes
Logical Operations	Bitwise AND	Any	-	AND	Integer	Integer	26,34
	Bitwise OR	Any	-	OR	Integer	Integer	26,34
	Bitwise XOR	Any	-	XOR	Integer	Integer	26,34
	Bitwise NOT	1	-	NOT	Integer	Integer	27,34
Shifts	Shift Left	2	-	LS	Integer	Integer	28,34
	Shift Right	2	-	RS	Integer	Integer	28,34
	Shift	2 or 3	-	SHFT	Integer	Integer	29,34
Truncation	Truncate Left	2	-	LT	Integer	Integer	30,34
	Truncate Right	2	-	RT	Integer	Integer	30,34
Storage Address	Actual Storage Address of Data Item	1	-	LOC	See Note 31	Integer*4	31,34
Random Number	Generate Random Number From 0 to 1.0	1	-	RND	Real	Real	34,35
	Generate Random Number From 0-32767	1	-	IRND	Integer	Integer	35

Notes for Tables 8-1 through 8-7

In the following notes the names of data types are given in lowercase. Uppercase is reserved for intrinsic function names.

1 The generic INT discards the fractional part of its argument, producing a truncated (unrounded) integral value. The result will be INTEGER*2 in a program unit compiled with -INTS, and INTEGER*4 in a program unit compiled with -INTL (the default).

2 INTS and INTL are similar to INT, differing only in that the result type is determined by the function selected rather than the compiler option in effect.

3 For a of type real, REAL(a) is a. For a of type integer or double precision, REAL(a) is as much precision of a as a real datum can contain. For a of type COMPLEX, REAL(a) is the real part of a.

4 For a of type double precision, DBLE(a) is a. For a of type integer or real, DBLE(a) is the value of a in double precision form. For a of type COMPLEX, DBLE(a) is the real part of a in double precision form.

For a of type REAL*16, QUAD(a) is a. For a of type integer, real, or double precision, QUAD(a) is the value of a in REAL*16 form. For a of type COMPLEX, QUAD(a) is the real part of a in REAL*16 form.

5 CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or COMPLEX. If there are two arguments, they must both be of the same type and may be of type integer, real, or double precision.

For a of type COMPLEX, CMPLX(a) is a. For a of type integer, real, or double precision, CMPLX(a) is the COMPLEX value whose real part is REAL(a) and whose imaginary part is zero.

CMPLX(a1,a2) is the COMPLEX value whose real part is REAL(a1) and whose imaginary part is REAL (a2).

6 DCMPLX is similar to CMPLX, except that a COMPLEX*16 number is produced.

7 Every ASCII-7 character is represented in the computer as a sequence of eight bits ranging from 10000000 to 11111111 (octal :200 to :377, Decimal 128 to 255). The Prime Extended Character Set (Prime ECS) is represented as a sequence of eight bits ranging from 00000000 to 11111111 (octal :000 to :0377, Decimal 0 to 255). The ASCII-7 characters are a proper subset of Prime ECS. Any such sequence can be interpreted either as a character or as an integer. CHAR and ICHAR provide a means for converting between the two

interpretations. On a Prime computer, a value that is out of the range of the character set is automatically mapped into the range of the character set as noted under the following discussion of CHAR. (For Prime ECS characters to be accepted by the compiler, use the -ECS compiler option. Refer to Chapter 9.)

ICHAR operates on a single character. For an ASCII-7 character, it returns an integer between 128 and 255; for a Prime ECS character, it returns an integer between 0 and 255. These represent the decimal equivalent of the bit pattern (ASCII-7 or Prime ECS) for that character.

CHAR operates on any integer, and under one of two possible conditions:

1. When the -ECS compiler option is invoked:

   If the integer is between 0 and 255, then all but the eight rightmost bits are truncated. The integer is used directly.

2. When the -ECS compiler option is not invoked:

   If the integer is between 128 and 255, then all but the eight rightmost bits are truncated. The integer is used directly.

   If the integer is not between 128 and 255, it is converted as follows:

   - Truncate all but the eight rightmost bits (the lowest-order byte).
   - Set the leftmost remaining bit to 1.

If conversion is required, CHAR returns the character whose bit pattern corresponds to the binary equivalent of its argument.

The effect of the conversion is that for every integer I

```
CHAR(I) = CHAR(MOD(I,128) + 128)
```

The compiler option -ECS is discussed in Chapter 9. This option deals with the ICHAR and with the CHAR functions. The complete Prime Extended Character Set is described in Appendix A.

8 ANINT(a) is defined as:

```
REAL(INTL(a+.5)) if a >= 0
REAL(INTL(a-.5)) if a < 0
```

DNINT(a) is defined as:

```
DBLE(INTL(a+.5)) if a >= 0
DBLE(INTL(a-.5)) if a < 0
```

QNINT(a) is defined as:

```
REAL*16(INTL(a+.5)) if a >=0
REAL*16(INTL(a-.5)) if a <=0
```

9 NINT(a) and IDNINT(a) are defined as:

```
INT(a+.5) if a >= 0
INT(a-.5) if a < 0
```

10 The argument to IABS may be INTEGER*2 or INTEGER*4. The result will be of the same type as the argument.

11 MOD yields the remainder when its first argument is divided by its second argument. Both arguments must be of the same type. The result will also be of that type.

The four specific functions under MOD are defined:

```
MOD(a1,a2) = a1 - (INTL(a1/a2) * a2)
AMOD(a1,a2) = REAL(a1 - (INTL(a1/a2) * a2))
DMOD(a1,a2) = DBLE(a1 - (INTL(a1/a2) * a2))
QMOD(a1,a2) = REAL*16(a1 - (INTL(a1/a2) * a2))
```

The result for MOD, AMOD, DMOD, and QMOD is a "Division by Zero" error when the value of the second argument is zero.

12 SIGN combines the magnitude of its first argument with the sign of the second. If the value of the first argument is zero, the result is zero, which is neither positive nor negative.

13 The value of the argument of the LEN function need not be defined at the time the function reference is executed.

14 INDEX(a1,a2) returns an integer value indicating the starting position within the character string a1 of a substring identical to string a2. If a2 occurs more than once in a1,

the starting position of the first occurence is returned.

If a2 does not occur in a1, the value zero is returned. Note that zero is returned if LEN(a1) < LEN(a2).

15 The REAL function for real part extraction is the same specific function that is selected when the generic function REAL is given a COMPLEX*8 argument.

The DREAL function for real part extraction is the same specific function that is selected when the generic function DBLE is given a COMPLEX*16 argument.

REAL and DREAL for real part extraction could not be passed as arguments in FORTRAN 77 because they are specific type conversion functions. To provide symmetry with AIMAG and DIMAG imaginary part extraction, which can be passed, F77 allows REAL and DREAL to be passed as arguments.

16 A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

17 The value of the argument of SQRT, DSQRT, and QSQRT must be greater than or equal to zero. The result of CSQRT and CDSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

18 The value of the argument of ALOG, DLOG, QLOG, ALOG10, DLOG10, and QLOG10 must be greater than zero. The value of the argument of CLOG, DLOG, and QLOG must not be (0.,0.). The result of CLOG, DLOG, and QLOG is the principal value, i.e. the range of the imaginary part of the result is -pi < imaginary part <= pi. The imaginary part of the result is pi only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

19 All angles are expressed in radians.

20 The result will be expressed in radians.

21 The absolute value of the argument of SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, QTAN is not restricted to a value less than 2*pi.

22 The absolute value of the argument of ASIN, DASIN, and QASIN must be less than or equal to one. The range of the result is -pi/2 <= result <= pi/2.

23 The absolute value of the argument of ACOS, DACOS, and QACOS must be less than or equal to one. The range of the result is: 0 <= result <= pi.

24 The range of the result for ATAN, DATAN, and QATAN is -pi/2 <= result <= pi/2. If the value of the first argument of ATAN2, DATAN2, or QATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and pi if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is pi/2. The arguments must not both have the value zero. The range of the result for ATAN2, DATAN2, and QATAN2 is: -pi < result <= pi.

25 All comparisons are based on one of the following collating sequences:

- ASCII-7 (American National Standard Code for Information Exchange ANSI X3.4-1977).
- Prime ECS, as described in Appendix A of this manual.

If both of the characters being compared are ASCII-7 characters, the ASCII-7 collating sequence is used; otherwise, the Prime ECS collating sequence is used.

LGE(a1,a2) returns the value .TRUE. if a1=a2, or if a1 follows a2 in the appropriate collating sequence. Otherwise, LGE(a1,a2) returns the value .FALSE.

LGT(a1,a2) returns the value .TRUE. if a1 follows a2 in the appropriate collating sequence. Otherwise, LGT(a1,a2) returns the value .FALSE.

LLE(a1,a2) returns the value .TRUE. if a1 = a2, or if a1 precedes a2 in the appropriate collating sequence. Otherwise, LLE(a1,a2) returns the value .FALSE.

LLT(a1,a2) returns the value .TRUE. if a1 precedes a2 in the appropriate collating sequence. Otherwise, LLT(a1,a2) returns the value .FALSE.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

The result-type for LGE, LGT, LLE, and LLT will be LOGICAL*4 in a program unit compiled with -LOGL, and LOGICAL*2 in a program unit compiled with -LOGS.

26 AND, OR, and XOR perform the bitwise logical function named on a list of long and short integers. The result will be a long integer if any argument is long; otherwise it will be a short integer.

When short and long integers are mixed, the short integers will be sign-extended, not zero-extended.

27 Performs a bitwise logical NOT function (ones complement) on a long or short integer. The result has the type of the argument.

28 LS and RS take two arguments; each argument may be either a long or a short integer. These arguments are called ARG1 and ARG2 in the following.

LS shifts ARG1 to the left by the number of bits specified in ARG2. The result has the type of ARG1 — that is, no type-change occurs. Vacated places are filled with zeros. If ARG2 is not positive, no shift occurs.

RS is identical to LS, except that the shift is to the right.

29 SHFT is similar to LS and RS, except that it can shift in either direction, and can perform two shifts rather than one. The additional shift occurs if a third integer argument, ARG3, is given.

If ARG2 is negative, the shift is to the left. If it is positive, the shift is to the right. If it is zero, no shift occurs.

If ARG3 appears, the shift specified by it will be carried out after the shift specified by ARG2 is complete. The rules are the same as for the ARG2 shift.

30 LT takes two arguments. Each argument may be either a long or a short integer. These arguments are called ARG1 and ARG2 in the following.

LT preserves the left ARG2 bits of ARG1, and sets the rest to zero (left truncation). The result has the type of ARG1 that is, no type change occurs. If ARG2 is <= 0, no bits are preserved.

RT is identical to LT, except that the right ARG2 bits are preserved.

31 LOC operates on an item of any data type except CHARACTER and LOGICAL*1. The result is an INTEGER*4 value representing the memory address where the first byte of the data item is located.

The LOC function returns two halfwords (four bytes) containing the address of its argument. The format for the returned address is as follows:

Bit	Meaning
1	Set to zero
2-3	Ring number
4	Data format code. Set to zero, indicating only two words in this data format
5-16	Segment number of argument
17-32	Halfword number of argument

32 An integer result produced by this function will be INTEGER*2 in a program unit compiled with -INTS, and INTEGER*4 in a program unit compiled with -INTL.

33 When this function operates on integers, the arguments may be a mixture of INTEGER*2 and INTEGER*4. The result will have the type of the longest argument.

A special case arises when IABS, MOD for integers, ISIGN, or IDIM is passed as an actual argument to a subprogram. In this case, the invoking program unit has no opportunity to examine the argument list on which the function will operate. Therefore it cannot select the version of the function that will implement the above rule. For compatibility with the FORTRAN 77 standard, the following rule is used instead.

When IABS, MOD for integers, ISIGN, or IDIM is passed as an actual argument to a subprogram, the function passed will accept and produce INTEGER*4 values if the invoking program unit was compiled with -INTL, and INTEGER*2 values if it was compiled with -INTS. This is the only case in which integer types cannot be mixed in the argument list of an integer intrinsic function.

34 This function cannot be passed as an argument to a subprogram.

35 The argument for RND and IRND is interpreted as follows:

Arg>0, Arg is used to initialize the random number generator. Arg is returned as the value of the call.

Arg=0, The function returns a random number: from 0 to 1.0 for RND, 0 to 32767 for IRND.

Arg<0, Initializes the random number generator and then returns a random number as in the Arg=0 case.

36 The COS function will raise a SIZE error whenever it calculates COS(x) for a REAL number x that does not fit into a 32-bit long integer.

STATEMENT FUNCTIONS

Statement functions are supplied by the user. Statement functions are useful or convenient when a particular function would otherwise be repeated at several different points in your program.

A statement function is a single statement procedure that you specify in much the same way as an assignment statement. After you define the statement function, the operation you specify executes whenever the name of the function appears in an expression within the same program unit. The value of the expression is assigned to the function name you specify in the statement function.

The definition of a statement function has the following format:

```
name ( [argument [,argument]...] ) = expression
```

where:

name is the symbolic name of the statement function. The data type of the statement function can be any of the data types. The same data type will be returned by the function.

argument is a dummy argument name that holds the value of an actual argument during execution of the statement function. You can have a list of dummy arguments specifying the order, number, and type of actual arguments substituted when the function is referenced. If the dummy argument name is duplicated in another statement function, the two names have no connection since the name is defined only within that function. You must uses parentheses even if you don't specify a dummy argument.

expression is any arithmetic, logical, or character expression, except one whose name duplicates that of a dummy argument and does not appear in the actual argument list at the function reference.

A statement function reference has the following format:

```
name ( [argument [,argument] ...] )
```

where:

name is the function name

argument is an actual argument that corresponds to the dummy argument in the statement function definition.

The following examples illustrate valid statement function definitions and references.

Definitions	References
TOTAL(X,Y,Z) = X+Y+Z	SUM = NET-TOTAL(TAX,COST,SALES)
AVG(A,B,) = A*B/2.0	PRINT*, AVG(DEPOS,DEDUCT)
FUNC(A) = A**B+10	Y = FUNC(SALARY)

At compilation time, F77 encounters the function reference and uses the reference's actual arguments with the dummy argument of the corresponding statement function. In the illustration above, you can see that the reference in the first example associates the actual argument TAX with dummy argument X, COST with Y, and SALES with Z. The actual arguments agree in order, number, and type with the corresponding dummy arguments.

EXTERNAL FUNCTIONS

External functions are supplied by you or from Prime's libraries. (See the Subroutines Reference Guide for more information.) An external function, also called a function subprogram, is another type of subprogram that looks very much like a subroutine or an intrinsic function. Unlike an intrinsic function or a subroutine, however, a function subprogram is an external program unit that is compiled separately from its calling program.

To define a function subprogram, you use the FUNCTION statement:

```
[type] FUNCTION name ( [argument [,argument]...])
```

where:

type is any F77 data type

name is the symbolic name of the function in which the FUNCTION statement appears. name is an external function name and must have the same data type as the function name in the calling program.

argument is a dummy argument.

To execute a function subprogram, you reference the function in an expression. The function reference has the following format:

```
name ( [argument [,argument] ...] )
```

where:

name is the name of the function.

argument is an actual argument or argument list of data items whose values are transferred to the function subprogram from the calling program and are treated the same as in subroutine calls.

You call a function subprogram by using its name and actual arguments in an expression. For example, the assignment statement:

```
VALUE = COST(X)
```

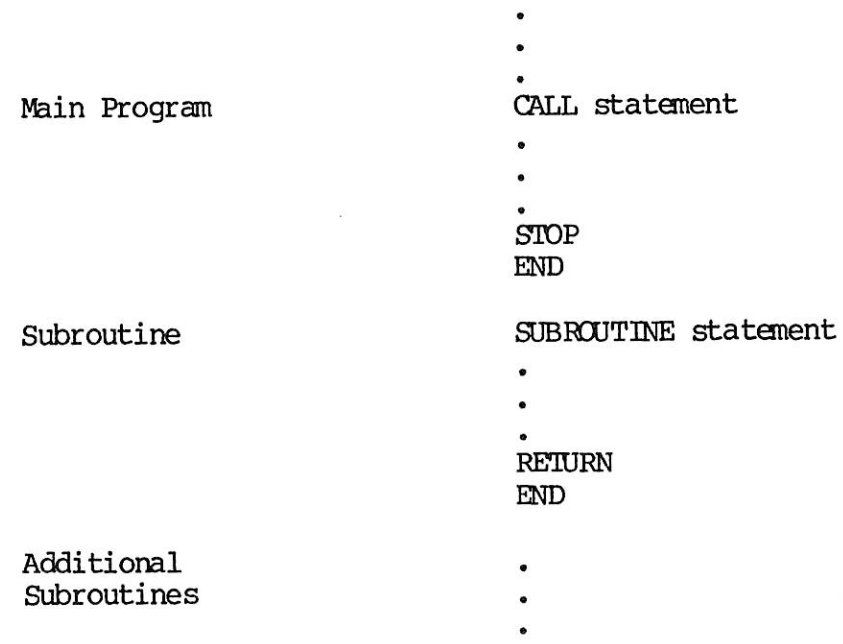
contains a function reference to a function subprogram called COST. X is an actual argument that will be associated with a dummy argument in the FUNCTION statement. The reference COST(X) passes the value in X to the function subprogram. After statements in the subprogram are executed, control returns to the main program. A single value for X is now available to the expression in the main program.

Caution

When a function reference appears in an expression, evaluation of the function must not alter the value of any other part of the expression, either directly or by altering arguments to other functions.

SUBROUTINES

A subroutine subprogram, also called a subroutine, is a separate program unit that accomplishes some particular computing task. It is placed immediately after the main program. A subroutine operates on the arguments it is passed and acts as a statement. Figure 8-1 shows the relationship of a subroutine to a main program.



Relationship of Subroutines to Main Program
Figure 8-1

You invoke a subroutine from the main program by using a CALL statement:

```
CALL name [( [argument [,argument]...] )]
```

where:

name is the symbolic name of the subroutine

arguments are a list of actual arguments, separated by commas, agreeing in number, order, and type with the dummy argument list in the subroutine's header statement. If the argument list is empty, you can omit the parentheses. Constants and expressions are permissible as arguments.

When program control reaches a call to a subroutine, control passes to the subroutine. A subroutine starts with a SUBROUTINE statement; its statements execute; and a RETURN statement is issued to return control to the main program. (Alternate returns are discussed later in this chapter.) Data is returned via the values of arguments and of data in COMMON. Data must not be returned to an actual argument that was an expression. If this were to happen no error message would be printed but invalid results might occur.

Caution

In FORTRAN 77, arguments are passed by reference (address). Therefore it is extremely important not to alter the value of a dummy argument whose actual argument is a constant or a parameter (a constant item). Such an alteration will alter the value kept in storage for the constant item, just as it would for a variable. If the compiler has utilized the same storage copy of the constant item in coding other references to the item, the altered value will be used when the code is executed.

Using the SUBROUTINE Statement

A subroutine always starts with a subroutine statement that has the form:

```
SUBROUTINE name [( argument [,argument]...])
```

where:

name is any legal F77 name having fewer than or exactly 32 characters.

arguments are a list of dummy arguments corresponding to actual arguments passed by the calling program unit. A dummy argument may be:

- A variable name or an array name that is typed and dimensioned.
- A dummy subprogram name. You must declare the corresponding actual argument INTRINSIC or EXTERNAL in the main program.
- An asterisk corresponding to an alternate return specifier.

In the following example there is a CALL statement from the main program to a subroutine, SUB1, to perform calculations on an argument having a value of 5:

```
      I = 5
      PRINT 10,I                  /* Value printed is 5
      CALL SUB1(I)
      PRINT 10,I                  /* value printed is 25.
   10 FORMAT (I2)
      STOP
      END

      SUBROUTINE SUB1 (J)
      J = J**2
      RETURN
      END
```

As you can see, SUB1 performs calculations on the value of the argument (I) passed to the dummy argument (J), then transfers control back to the main program.

Subroutine Libraries

Prime supplies several libraries of subroutines. These allow PRIMOS subroutines to be called by an F77 program, and also provide access to various commonly used utilities. To load a subroutine library, use the BIND subcommand, LIBRARY:

```
LIBRARY library name
```

This command must be given at load time before the unqualified LI command is given. However, many PRIMOS subroutines and functions are loaded simply with the unqualified LI.

For more information, see the Subroutines Reference Guide.

Note

Many PRIMOS subroutines require and return short integer arguments. When long integers are used to supply data to such a subroutine, convert them directly in the argument list with the INTS intrinsic function. Arguments to which data is returned must themselves be short integers, since data cannot be returned to an expression.

Recursion

In FORTRAN 77, recursion is not permitted. F77 has been extended to permit recursion in subroutines, though not in functions. The rules and syntax are identical in recursive and non-recursive subroutine calls.

Number of Arguments

The following is a description of the upper limit to the number of arguments that F77 allows:

- 254 arguments can be passed to and from subroutines
- 247 entries may be present in a Namelist block

There are exceptions to these limits:

1. If all the arguments to a subroutine are of type CHARACTER, F77 only allows 127 arguments.

2. If the arguments are of mixed data types (CHARACTER and other types), the maximum number of arguments is between 127 and 254, and depends upon the positions occupied by the CHARACTER data in the argument list.

Similiar restrictions exist for namelist blocks. For information on using namelist blocks, see Chapter 6.

BLOCK DATA SUBPROGRAM

A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. A block data subprogram is nonexecutable. The BLOCK DATA statement is discussed in Chapter 3.

SECONDARY ENTRY POINTS

The ENTRY statement allows you to call a function subprogram or subroutine subprogram in a place other than where a FUNCTION or SUBROUTINE statement begins. The ENTRY statement has the following format:

```
ENTRY name [( [argument [,argument]...] )]
```

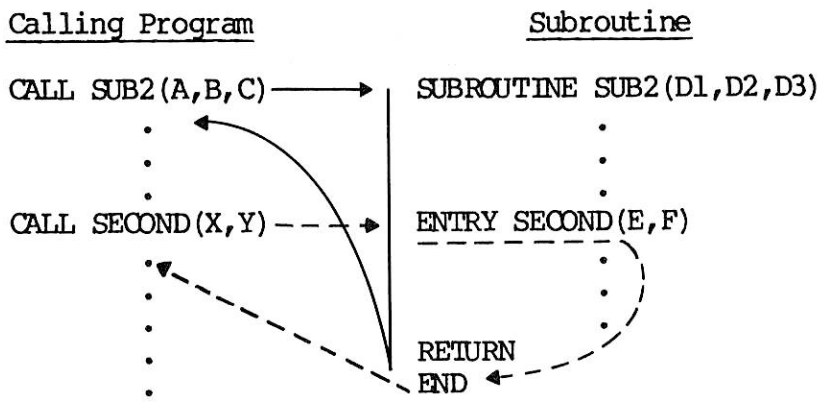
where:

name is the symbolic name of an entry in a function or subroutine subprogram and is called an entry name. If ENTRY appears in a function subprogram, name is a function name.

argument is a dummy argument corresponding to an actual argument in a CALL statement or a function reference. argument can be a variable name, array name or dummy procedure name or an asterisk. An asterisk is permitted only in an ENTRY statement in a subroutine subprogram.

A secondary entry is referenced (in a function) or CALLed (in a subroutine) exactly as the main entry point would be, and supplied arguments corresponding to its particular argument list. Program execution begins at the entry and proceeds until a RETURN or END statement is encountered. ENTRY statements are non-executable and therefore ignored if encountered.

Figure 8-2 illustrates the use of an ENTRY statement in a subroutine subprogram.



Flow of Control of a Secondary Entry Point in a Subroutine.

Figure 8-2

An entry name to a function may be typed by default or in a type statement. The type may differ from that of the function name and of other entry names, except that all entry names in a CHARACTER function must be of type CHARACTER and have the same *(length) specification. All entry names in a function are automatically equivalenced. Before the function returns, assignment to an entry name of the same type as the entry name used in referencing the function must occur.

Alternate returns are permitted following the CALL of a subroutine at an entry point. Only the statement labels in the entry point's argument list are counted. Alternate returns are discussed below.

Note

In some versions of FORTRAN IV, the association of actual and dummy arguments established when a subprogram is invoked at any entry point persists following return to the invoking program unit. Consequently, a subprogram can be invoked repeatedly at various entry points, and reference made after each invocation to any dummy argument that became associated with an actual argument at any previous invocation. This technique is not accepted by any Prime FORTRAN.

ALTERNATE RETURNS

As the above illustration shows, usually a subroutine returns control to the statement following the point of call. It can also return control to any labeled executable statement that you specify in the calling program unit through the use of a RETURN statement. The RETURN statement has the following format:

```
RETURN [n]
```

where:

n is an integer expression indicating the alternate point in the main program that is to receive control from the subroutine subprogram. If n is not specified, a normal return to the statement following the CALL statement is executed.

The subroutine can select the statement to which it will return. Alternate returns are accomplished as follows:

1. The label of every statement to which the subroutine might return must appear in the argument list of the CALL statement, prefixed by an asterisk (or a dollar sign -- F77 extension).

2. An asterisk appears in the dummy argument list of the subroutine at each position corresponding to a statement label in the CALL statement.

3. RETURN statements in the subroutine may optionally be followed by an integer expression n. When control encounters a RETURN n in the subroutine, the subroutine will return to the statement of the calling program unit whose label corresponds to the nth asterisk in the dummy argument list of the subroutine. If control first encounters a RETURN without a number, or with a number outside the applicable range, a return to the point of call will occur.

For example:

```
      PROGRAM ALTRTRN
100   CALL PROC1 (J)
300   CALL PROC2 (K)
500   CALL PROC3 (J, K, *100, 4, *900)
700   GO TO 100
900   STOP
      END

      SUBROUTINE PROC3 (J,K,*,M,*)
      IF (I .EQ. J) RETURN
      IF ((I + J) .EQ. K) RETURN 1      /* Returns to stmt 100
      IF ((I + K) .EQ. J) RETURN M/2    /* Returns to stmt 900
      RETURN                            /* Returns to stmt 700
      END
```

Alternate returns are permitted following the CALL of a subroutine at a secondary entry point. Only the asterisks in the dummy argument list at the point of entry are counted.

SUBPROGRAM ARGUMENTS

Adjustable Subprogram Elements

The length of the value returned by a type CHARACTER function, the lengths of type CHARACTER dummy arguments in a subprogram, and the dimension bounds of an array dummy argument, can be made adjustable. An adjustable element will take on the length or bounds of the corresponding actual argument at each call. Such flexibility can considerably increase the versatility of a subprogram.

Adjustable Character Functions

To make a CHARACTER function adjust the length of its result, specify its length as an asterisk in parentheses:

```
CHARACTER*(*) FUNCTION CFUNC (A,B)
```

In each program unit referencing the adjustable function, use a type-statement to assign the CHARACTER type and a length to the name of the function. The length of the value returned at each function reference will be the one assigned to the function in the calling program unit.

Adjustable Character Arguments

To make a type CHARACTER dummy argument adjustable, specify its length to be (*) in a type-statement:

```
SUBROUTINE YORD (CVAR)
CHARACTER*(*) CVAR
```

CVAR will take on the length of the actual argument corresponding to it at each call.

Assumed-size Arrays

To create an assumed-size array, replace the upper bound of the last dimension specification in a fixed or adjustable dummy array declaration with an asterisk. That dimension will take on the upper bound associated with it in the corresponding actual array in the calling program unit.

Adjustable Array Dimensions

To create an adjustable array, pass the name of an existing array to an appropriately typed dummy argument in a subprogram. Dimension the dummy array using:

1. Integer variables passed to integer dummy arguments in the subprogram.
2. Integer variables from COMMON.

Expressions are permitted in adjustable array bound declarations, subject to the following restrictions:

- All variables must be INTEGER
- No array references
- No function references

Example:

```
REAL FUNCTION ARRTEST(ANAME, DIM1, DIM2)
IMPLICIT INTEGER (A-Z)
COMMON /BND/ DIM3,N
DIMENSION ANAME (DIM1, DIM2:N, 1:10, DIM3+12)
```

When control passes to a subprogram containing an adjustable array, the array bounds are determined before execution begins. The variables used may therefore be redefined or become undefined during execution without affecting the dimensional properties of the array.

> **Caution**
>
> Adjustable arrays do not represent dimension-by-dimension subsets of the original array, but are equivalenced to the original array as a whole. The adjustable array cannot be longer than the corresponding actual array.

Boundary Spanning Arrays as Arguments

The F77 compiler can produce two types of object code. Ordinary code can address only within a segment. Boundary-spanning code is capable of addressing across the boundary between one segment and the next.

Whenever an array extends across a segment boundary, all references to it must consist of boundary-spanning code. Those portions of it in segments higher than the one in which it begins are inaccessible to ordinary code.

Arrays in local static or dynamic storage present no problem. There may be at most one segment for all static variables, and another for all dynamic variables: hence no boundary-spanning is possible. Arrays in COMMON blocks under 128K bytes (one segment) long present no problem because such blocks are always loaded within a single segment.

An array in a COMMON block over one segment long (a large COMMON block) may or may not span a segment boundary, depending on its size and its location in the block. In practice, no array under one segment long should ever be placed in a large COMMON block. See the Note below.

When a program unit is compiled, the F77 compiler inspects any COMMON statements in it for COMMON block size and the presence of arrays. All references in the program unit to any array the compiler knows to be in a large COMMON block will automatically be compiled with boundary-spanning code. No special action is required of the programmer in this case.

However, when a dummy array occurs in a subprogram, the compiler is not aware of the storage status of any actual array that will become associated with it when the subprogram is called. Therefore, the compiler does not know whether to compile references to the dummy array with ordinary or boundary-spanning code. The programmer must specify to the compiler the correct action — in this case the use of the -BIG/-NO_BIG compiler option.

When a subprogram is compiled with -NO_BIG (the default), dummy array references within it will be compiled with ordinary code. The actual array passed to any dummy array in it must then be contained within one segment. When a subprogram is compiled with -BIG, all references it makes to any dummy array will be compiled with boundary-spanning code. The actual array passed to any dummy argument in it may then span a segment boundary, though it need not do so.

A dummy-array reference compiled with boundary-spanning code will execute correctly for any actual array, whether it spans a segment boundary or not. However, boundary-spanning code executes more slowly than ordinary code because it performs more complex address calculation. The -BIG option should therefore not be used unnecessarily.

Note

An array less than 128K bytes long should not be put in a large COMMON block, since this will cause the inefficiency of boundary-spanning code to be needlessly incurred in every reference to the array.

Character Arrays as Arguments

When a CHARACTER array that may cross a segment boundary is passed as an argument, the element size of the actual array and the dummy array must be the same. This is an F77 restriction required to insure that no element of the array can fall across a segment boundary. See the COMMON Statement in Chapter 3 for more information on COMMON block restrictions.

Subprograms as Arguments

Entire subprograms may be passed as arguments to other subprograms, where they may be referenced or passed again. The general method is as follows:

1. In the invoking program unit, name any intrinsic functions to be passed in an INTRINSIC statement, and any user supplied or library subprograms to be passed in an EXTERNAL statement.

2. In the actual argument list for each invocation, the subprograms which are to be passed to the invoked subprogram are named.

3. Place the following dummy names at the entry point to the called subprogram (either in its header or an ENTRY statement):

   - An untyped dummy subroutine name at each position corresponding to an actual argument that is a subroutine.

   - An appropriately typed dummy function name at each position corresponding to an actual argument that is a function.

4. In the invoked subprogram, use the appropriate dummy subprogram name wherever a reference to the corresponding actual subprogram is desired.

For example, suppose that the program called MAIN calls the subroutine SUB repeatedly, and each time passes one of the intrinsic functions DSIN and DCOS, as well as one of the user-supplied subroutines GREATER and LESSER. The code could be as follows:

```
PROGRAM MAIN
INTRINSIC  DSIN, DCOS
EXTERNAL  GREATER, LESSER
CALL SUB (DSIN, GREATER, 1.D0)
CALL SUB (DCOS, GREATER, 1.D0)
CALL SUB (DCOS, LESSER, 1.D0)
STOP
END

SUBROUTINE SUB (TRIG, COMPARE, NUM)
DOUBLE PRECISION TRIG, NUM
IF (TRIG(NUM) .GT. DTAN (NUM)) CALL COMPARE (NUM)
RETURN
END
```

Not all intrinsic functions can be passed as arguments. See the section F77 INTRINSIC FUNCTIONS discussed previously in this chapter before passing intrinsic functions.

PART III

Working With Prime F77

9
Compiling Your Program

Prime's F77 compiler translates the statements in your source program into an object (binary) module and produces an optional listing file. This binary module contains the machine code that is needed to link and execute your program. The optional listing file contains a compiler output listing that gives error and statistical information, and other helpful messages and information about your source program.

This chapter describes:

- How to compile FORTRAN programs
- How to specify options to the compiler
- Compiler error messages
- Compiler options

For information on using BIND to link your program, see Chapter 10.

COMPILING AN F77 PROGRAM

After you have entered your source program into the system using ED or EMACS, and have named your program with a .F77 suffix, you are ready to invoke the F77 compiler.

Invoking and Specifying Options to the Compiler

To invoke the FORTRAN 77 compiler from the PRIMOS command level, use the F77 command:

```
F77 pathname [-option 1] [-option2] ... [-option n]
```

pathname is the pathname of the source program you want to compile.

options are the names of the compiler options that you invoke on the command line. These options provide information and input while you compile, link, and execute your program. Every option must begin with a hyphen (-).

For example:

```
OK, F77 TEST -BIG
[F77 Rev. 19.4]
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]
OK,
```

You can specify more than one option on the command line, in any order. However, if you issue conflicting or redundant options, an error message will result.

Compiler Error Messages

During compilation, the compiler will output an error message each time it encounters an error in your program. The error messages, which are self-explanatory, will assist you in finding and correcting the errors in your program. For every error found, the compiler displays information about where the error occurred and the level of severity:

```
ERROR xxx SEVERITY y BEGINNING ON LINE zzz
Explanation of message
```

xxx	Error code
y	Level of severity
zzz	Line number where error begins
explanation	Description of the error, and possible remedies.

Errors are classified into four levels depending on the severity of the error. Table 9-1 describes each level of error.

Table 9-1
Error Message Severity Levels

Level 1

ERROR TYPE: Warning — a recoverable error, object file produced.

Example:

```
OK, F77 TEST
[F77 Rev. 19.4]

WARNING 250 SEVERITY 1 BEGINNING ON LINE 1
A program unit consisting of just an END statement has been
encountered.

0001 ERRORS [<> F77 Rev. 19.4]
MAX SEVERITY IS 1
OK,
```

Level 2

ERROR TYPE: Recoverable — the compiler will attempt corrective action.

Example:

```
OK, F77 RAISE
[F77 Rev. 19.4]

ERROR 274 SEVERITY 2 BEGINNING ON LINE 30
Missing END statement is supplied by the compiler.

0001 ERRORS [<RAISE> F77 Rev. 19.4]
MAX SEVERITY IS 2
OK,
```

Level 3

ERROR TYPE: Nonrecoverable — object file not produced.

Example:

```
OK, F77 PUZZLE
[F77 Rev. 19.4]

ERROR 369 SEVERITY 3 BEGINNING ON LINE 6
Invalid argument to the "ICHAR" intrinsic function.

0001 ERRORS [<.MAIN.> F77 Rev. 19.4]
MAX SEVERITY IS 3
ER!
```

Table 9-1 (continued)
Error Message Severity Levels

Level 4
ERROR TYPE: Abort the compilation. Example: OK, F77 POGO [F77 Rev. 19.4] Not found. Cannot open file "TEST.FILE" (OPEN) SOURCE LINE NUMBER 2 0001 ERRORS [<> F77 Rev. 19.4] MAX SEVERITY IS 4 ER!

End-of-Compilation Message

After the compilation process is complete, the compiler prints an end-of-compilation message at the terminal. Its format is:

```
0000 ERRORS [<.MAIN.> F77 Rev.  19.4]
```

After compilation, control returns to the PRIMOS level.

COMPILER OPTIONS

This section discusses the options available with the F77 compiler. Most of the options come in pairs, which act as switches to enable or disable a particular action. The Prime-supplied defaults are indicated by an asterisk. These defaults can be changed by your System Administrator.

Table 9-2 lists a summary of compiler options and abbreviations.

▶ -32I

The -32I option generates 32I-mode code, which is a segmented virtual mode that takes maximum advantage of Prime's 32-bit machine architecture (P450 and up).

▶ -32IX

The -32IX option is a new addressing capability added to 32I-mode code that can speed up the access of arrays larger than one segment. It also permits use of common blocks larger than one segment. This option gives the effect of using general registers as base registers.

Note

A program compiled with the -32IX option can be run only on a 2550, 9650, 9750, 9950, or 9955 Prime system.

▶ *-64V

The -64V option generates 64V-mode code, which is a segmented virtual addressing mode for 32-bit machines.

▶ *-ALLOW_PRECONNECTION / -NO_ALLOW_PRECONNECTION

Abbreviation: -APRE / -NAPRE

The -ALLOW_PRECONNECTION option allows for the preconnection of a listing output to a preopened file unit 2, or of a binary output to a preopened file unit 3. When files have been preconnected, the compiler displays a message indicating that preconnection has occurred.

For example:

```
OK, BINARY SNOW
OK, LISTING FLAKE
OK, F77 CIRCLE -LISTING
[F77 Rev. 19.4]
Note: Binary output will go to pre-opened unit 3 (preconnection). (F77)
    Explicit use of -Allow_PREconnection is recommended when using preconnection.
    Use -Allow_PREconnection on the command line to suppress this note.
    Use -No_Allow_PREconnection to avoid accidental preconnection.
Note: Listing output will go to pre-opened unit 2 (preconnection). (F77)
    Explicit use of -Allow_PREconnection is recommended when using preconnection.
    Use -Allow_PREconnection on the command line to suppress this note.
    Use -No_Allow_PREconnection to avoid accidental preconnection.
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]
0000 ERRORS [<SIRKLE> F77 Rev. 19.4]
OK,
```

When the -NO_ALLOW_PRECONNECTION option is used, no attempt to perform preconnection is made. The compiler automatically selects file units for listing and binary files, opens, and closes them.

▶ -BIG / *-NO_BIG

Abbreviation: -BIG / -NBIG

The -BIG option handles arrays spanning segment boundaries. A dummy array can become associated with any array, even if it crosses a segment boundary.

The -NO_BIG option specifies that a dummy array can become associated only with an array that does not cross a segment boundary.

See Arrays as Arguments in Chapter 8 for details.

▶ * -BINARY [pathname] / -NO_BINARY

Abbreviation: -B / -NB

The -BINARY option produces an object (binary) file with the name source-program.BIN. To write the object code to a different file, use the -BINARY option followed by pathname.

-NO_BINARY specifies that no binary object file is to be produced. Use this option when only a syntax check or listing is desired.

▶ -CLUSTER

Abbreviation: -CLU

-CLUSTER specifies that all routines in a source file be compiled as a cluster and optimized together. A cluster is a collection of program units in one source file that have been compiled together in order to maximize the optimizations that can be performed. Use of this option means that the compiler can make certain assumptions that are relevant to optimization. The compiler will check the validity of these assumptions when possible, but the responsibility for their validity rests with the user.

The assumptions the compiler will make are the following:

1. The file compiled with the -CLUSTER option is assumed to be a program with a single entrypoint. If the file has a main program in it, then that is the entrypoint. Otherwise, if the user has used the -MAIN option to specify a main entry procedure, then that routine is the program entrypoint. Otherwise, the first routine is taken to be the program entrypoint.

2. The compiler will not make any procedure or data entrypoints of the cluster visible outside the cluster

except the main procedure entrypoint. All other procedures may be QUICK-called or expanded inline. You must also specify -OPT 4 when using -CLUSTER for this purpose. The binaries from such a compilation cannot be combined with other modules that expect to call these procedures.

▶ -DCLVAR / *-NO_DCLVAR

Abbreviation: -DC / -NDC

The -DCLVAR option controls flagging of undeclared variables. A warning will be generated for any variable that is used in the program, but not included in a type statement.

For example:

```
OK, F77 FRIDAY -DCLVAR
[F77 Rev. 19.4]

ERROR 413 SEVERITY 2 BEGINNING ON LINE 4
"TOTAL" has been defined by the context in which it was used. Declare
all variables, function references, and subroutine references.

ERROR 413 SEVERITY 2 BEGINNING ON LINE 4
"J" has been defined by the context in which it was used. Declare
all variables, function references, and subroutine references.

0002 ERRORS [<.MAIN> F77 REV. 19.4]
MAX SEVERITY IS 2
OK,
```

-NO_DCLVAR specifies that no undeclared variable warnings be generated.

▶ -DEBUG / *-NO_DEBUG

Abbreviation: -DBG / -NDBG

The -DEBUG option generates full debugger (DBG) functionality code. With -DEBUG, the object file is modified so that it will run under the source level debugger. Execution time increases, and the code generated will not be optimized.

-NO_DEBUG causes no debugger code to be generated.

In Chapter 11 of this book you will find an introduction to the Source Level Debugger. For complete information on this separately priced product, see the Source Level Debugger User's Guide.

▶ -DO1 / *-NO_DO1

Abbreviation: -DO / -NDO

DO1 specifies that all DO loops will be of the FORTRAN IV type. This option is provided for upward compatibility of FTN programs.

The use of the NO_DO1 option specifies that FORTRAN 77 DO loops will be produced. These are described in Chapter 5. They differ significantly from those in FTN.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -NO_DO1.

▶ -D_STATEMENT / *-NO_D_STATEMENT

Abbreviation: -D_STMT / *-ND_STMT

The -D_STATEMENT option causes the F77 compiler to interpret all statements that begin with a D in column 1 as normal source coding. The "D Statements" are used primarily for debugging purposes and are valueable when used with applications that require continual maintenance.

If the -NO_D_STATEMENT is specified, or if the -D_STATEMENT option is not specified, the statements with a D in column 1 are treated as comments. Thus, with the minimum amount of effort, the user may switch into the debugging mode or the run mode, and out again by merely using or not using the -D_STATEMENT option. Reliance upon the DEBUGGER may be somewhat diminished.

The program fragment below illustrates the use of the -D_STATEMENT option. (The complete program is Sample Program #1 in Appendix B of this manual.)

```
*****************************************************************
*                  UPDATE 1 -D_STATEMENTS                       *
*             ONLY WHEN THE -D_STATEMENT OPTION IS              *
*             IS USED IN THE COMPILATION, WILL THE              *
*                    VALUE FOR SPORTY PRINT                     *

D     SPORTY = 2

D     PRINT*, SPORTY
```

▶ -DYNM

Abbreviation: -DY

-DYNM allocates local storage dynamically. The opposite option is -SAVE. Dynamic storage variables are kept in each program's stack. At each call to a subprogram, space for its dynamic variables is allocated. At RETURN, the space is freed, and the data lost. -DYNM allocates dynamic storage to all variables not SAVEd or in COMMON, and is used principally to save space in user memory.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -DYNM.

▶ -ERRLIST / *-NO_ERRLIST

Abbreviation: -ERRL / -NERRL

The -ERRLIST option controls the generation of an errors-only listing. A listing file named source-program.LIST will be generated. This file contains only the error messages for the current compilation. -ERRLIST has no effect when a full source listing is specified or implied.

The -NO_ERRLIST option causes an errors-only listing file to be generated. Does not override the -LISTING option.

▶ *-ERRTTY / -NO_ERRTTY

Abbreviation: -ERRT / -NERRT

The -ERRTTY option controls printing of error messages at the terminal. Error messages will be printed at the terminal during compilation.

The -NO_ERRTTY option causes no error messages to be printed. They will still be included in the source listing file, if any.

▶ -EXPLIST / *-NO_EXPLIST (Implies -LISTING)

Abbreviation: -EXP / -NEXP

The -EXPLIST option inserts a pseudo-assembly code listing into the source listing. Each statement in the source will be followed by the pseudo-PMA (Prime Macro Assembler) statements into which it was compiled. For information on PMA, see the Assembly Language Programmer's Guide.

The -NO_EXPLIST option causes no assembler statements to be printed in the listing.

Figure 9-1 contains a sample listing created with -EXP.

```
OK, F77 FRIDAY -EXPLIST
[F77 Rev. 19.4]
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]


OK, SLIST FRIDAY.LIST
SOURCE FILE: MONTH>DAY>FRIDAY.F77
COMPILED ON: 850211  AT: 21:11  BY: F77 REV. 19.4
Options selected:  FRIDAY -EXPLIST
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
    "-Full_OPTimize" means "-OPTimize 4", and default is "-OPTimize 2".
Options used(* follows those that are not default):
     64V Allow_PREconnection No_BIG Binary No_DClvar No_DeBuG No_DOl DYnm
     No_ERRList ERRTty EXPlist* No_FRN No_FTN_Entry INTL Listing* LOGL MAp
     OFFset* OPTimize(2) No_OverFlow No_PBECB No_PRODuction No_RAnge
     SIlent(-1) TIME No_STATistics No_Store_Owner_Field UPcase XRef*

    1   c
    2         INTEGER*2 ARRAY(5), TOTAL
    3         DATA ARRAY/10,200,40,55,78/
    4         TOTAL=0

000033:     140040        CRA
000034:     04.000053S    STA   SB%+53

    5         DO 100 J=1,5

000035: 005414.000000     LDL   PB%+0
000037: 011415.000054S    STL   SB%+54

    6             TOTAL=TOTAL+ARRAY(J)

000041: 005415.000054S    LDL   SB%+54
000043: 011415.000056S    STL   SB%+56
000045:     02.000053S    LDA   SB%+53
000046:     15.000057S    LDX   SB%+57
000047:     06.000423L    ADD   LB%+423,X
000050:     04.000053S    STA   SB%+53
                  .
                  .
                  .


OK,
```

Pseudoassembly Code Listing
Generated with -EXPLIST Option

Figure 9-1

▶ -EXTENDED_CHARACTER_SET / *-NO_EXTENDED_CHARACTER_SET

Abbreviation: -ECS / -NECS

Prior to Revision 21.0 of F77, the high-order bit of an ASCII-7 character representation was always set. Thus a 40 (octal) and 240 (octal) passed to the intrinsic function CHAR both returned the SPACE character. At Revision 21.0, F77 supports Prime Extended Character Set (Prime ECS), so each of these arguments can produce a unique character; 40 (octal) returns NBSP (NO-BREAK SPACE), and 240 (octal) returns SP (SPACE). The full Extended Character Set is printed in Appendix A.

To take advantage of this new capability, the -EXTENDED_CHARACTER_SET (-ECS) compiler option must be used. This option causes all unique arguments submitted to the CHAR intrinsic function to return unique characters.

-NO_EXTENDED_CHARACTER_SET causes the CHAR function to map the argument into the range of values 128 through 255. Thus CHAR X will give the expected character when X is in the range of 128 through 255 (pre-Revision 21.0 characters). Should X be in the range of 000 to 128 (new characters in Prime ECS), then CHAR will map the value of X into the range of values, 128 through 255. For instance, if X = 34, CHAR(X) should return the Cent Sign but instead it will return # or the same as if X = 163.

▶ -FRN / *-NO_FRN

Abbreviation: -FRN / -NFRN

The Floating Point Round option improves the accuracy of calculations involving single-precision real numbers. Such numbers are REAL or REAL*4 in F77.

When the -FRN option has been given, all single-precision numbers are rounded each time they are moved from a register to main storage. The method of rounding is: if the last bit of the mantissa is 1, add a 1 to the second-to-last bit, then set the last bit to 0. This rounding reduces loss of accuracy in the low-order bits when many calculations are performed on the same number.

The -FRN option does not affect double-precision real numbers (REAL*8, DOUBLE PRECISION) or quadruple floating point precision numbers (REAL*16). It causes a slight increase in execution time, and should therefore be used only when maximum accuracy is a major consideration.

-NO_FRN will cause no rounding to be performed.

▶ -FTN_ENTRY / *-NO_FTN_ENTRY

Abbreviation: -FTNE / -NFTNE

The use of the -FTN_ENTRY option considers that all calls where procedure names are being passed as actual arguments are calls to FTN procedures.

-NO_FTN_ENTRY considers all calls where procedure names are being passed as actual arguments are calls to a non FTN procedure.

▶ -FULL_HELP

Abbreviation: -FH

The -FULL_HELP option is similar to the -HELP option, except that in addition to the usage summary, a description of the meaning of each compiler option is given. The -HELP option is described below.

▶ -FULL_OPTIMIZE

Abbreviation: -FOPT

-FULL_OPTIMIZE ensures that the maximum amount of optimization available is used. A note in the listing file will show the current level of optimization implied by the use of this option. The default of -FULL_OPTIMIZE is equivalent to OPT 6; however, -CLUster must be specified additionally to achieve inline expansions.

▶ -HELP

Abbreviation: -H

The -HELP option produces information on using the F77 compiler. The compiler displays a usage summary and a list of all options available.

▶ -INPUT pathname

Abbreviation: -I

This is an obsolete option. -INPUT is an alternative way to specify the source of the compilation. pathname specifies the name of the source program. If pathname is TTY, then input will come from the terminal. This option should not be used if the pathname immediately follows the F77 command. This is the same as the -SOURCE option.

▶ *-INTL / -INTS

These options determine default lengths for type INTEGER data items whose length is not explicitly declared.

The -INTL option specifies that every type INTEGER data item, including constants and parameters, will be compiled as INTEGER*4 unless the item has been explicitly declared INTEGER*2 in a type statement.

The -INTS option specifies that every such data item will become INTEGER*2 unless it is explicitly declared INTEGER*4. A constant will remain INTEGER*4 under -INTS if:

- Its value lies outside the INTEGER*2 range.
- Its representation, including leading zeros, contains more than five decimal or six octal digits.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -INTL.

▶ -LISTING [destination] / *-NO_LISTING

Abbreviation: -L / -NL

The -LISTING option controls the creation of the source listing file. If you do not specify destination, the listing file is named source-program.LIST. destination must be one of the following:

TTY	The listing will be displayed at the terminal.
SPOOL	The listing will be spooled directly to the line printer. Default SPOOL arguments are in effect.
pathname	The listing will be written to the file named pathname.LIST

-NO_LISTING causes no listing file to be created.

▶ *-LOGL / -LOGS

The -LOGL and -LOGS options determine default lengths for type LOGICAL data items whose length is not explicitly declared, and for the logical constants.

-LOGL specifies that every type LOGICAL data item will be compiled as LOGICAL*4 unless the item has been explicitly declared LOGICAL*2 in a type statement. This is the default.

-LOGS specifies that every type LOGICAL data item will be compiled as LOGICAL*2 unless it is explicitly declared LOGICAL*4 in a type statement.

The F77 compiler acts as a standard-conforming compiler only when it is invoked with -LOGL.

▶ -MAIN program entry-name

-MAIN specifies a top-level routine as the main program entrypoint. This option is used in conjunction with the -CLUSTER option.

▶ -MAP / *-NO_MAP (Implies -LISTING)

Abbreviation: -MA / -NMA

The -MAP option produces a listing file that contains a reference map of data and procedure names. To get a full cross-reference of usage information for each symbolic name, use the -XREF option.

NO_MAP produces a listing file that includes only the program and error messages without a variable reference map.

▶ -MAPWIDE [decimal integer] (Implies -LISTING)

Abbreviation: -MAPW

-MAPWIDE specifies the width in number of characters of the cross-reference map that appears in the listing file, as well as the width of the options list that appears at the beginning of the listing file. The legal range of values for the decimal integer argument is from 80 to 160 inclusive. The default width of the cross-reference map, if -MAPWIDE is not specified is 80, provided that a listing is being produced. The default width if -MAPWIDE is specified without an argument is 108.

▶ -MAX_GROWTH_PERCENT [decimal_integer]

Abbreviation: -MXGR

-MAX_GROWTH_PERCENT specifies a suggested limit to the growth of the size of a program due to optimization. decimal_integer is the limit to the growth expressed as a percent of the original program size. The default percent is 100; the percent cannot be 0. This size is a suggestion only; there is no guarantee that the growth in program size will not exceed it.

▶ -MAX_SUB_STATEMENTS_INLINE [decimal integer]

Abbreviation: -MSSI

-MAX_SUB_STATEMENTS_INLINE specifies the maximum number of executable source line statements in a subroutine in order for it to be expanded inline. The compiler optimizes for faster execution by avoiding the overhead of procedure calls through inserting their code inline, thereby having an effect on the size of programs.

If no argument is present, or if the option is omitted entirely, the number of source lines defaults to 20.

▶ -MAXERRORS [decimal integer]

Abbreviation: -MAXE

-MAXERRORS specifies the maximum number of compilation errors to be reported. If in a given compilation the specified maximum is reached, then an error message is issued and the compilation is aborted. The legal range for the number of errors to be reported is 1 to "infinity" (32767).

The default maximum number of errors, if -MAXERRORS is not specified, remains 100; the maximum number of errors that can be reported if -MAXERRORS is specified without a decimal argument, is "infinity".

▶ -NESTING / *-NO_NESTING (Implies -LISTING)

Abbreviation: -NE / *-NNE

The -NESTING option generates compiler listing files with numbers alongside each statement indicating the nesting level of the statement within the compilation unit.

When -NESTING is invoked, the -LISTING option is enabled and a column

of numbers appear between the statement ID numbers (at the lefthand margin) and the F77 statements. For example:

```
Statement   Nesting   F77
Number      Level     Stmt
------      -----     ----
1           0         J=10
2           0         DO 100 I = 1,10
3           1          K(I) = I
4           1          IF (I .EQ. 5) THEN
5           2           J = J + 2
6           1          ELSE
7           2           J =J - 1
8           1          ENDIF
9           1          PRINT *, K(I)
10          1          100 CONTINUE
11          0         END
```

-NO_NESTING disables the option.

▶ -OFFSET / *-NO_OFFSET (Implies -LISTING)

Abbreviation: -OFF / -NOFF

The -OFFSET option appends an offset map to the source listing. For each statement in the source program, the offset map gives the offset in the object file of the first machine instruction generated for that statement.

▶ *-OPTIMIZE [decimal-integer]

Abbreviation: -OPT

-OPTIMIZE controls the optimization phase of the compiler. Optimized code runs more efficiently that non-optimized code, but takes somewhat longer to compile. The decimal-integer that follows -OPTIMIZE specifies one of the following levels:

Level	Meaning
0	Performs no optimizations. Turns optimization off.

1	Replaces certain code patterns with more efficient ones.
2	Eliminates common subexpressions. (-OPT 2 is the default level of optimization.)
3	Moves invariant expressions outside of loops.
4	Performs strength reduction in loops, optimizes GOTOs and statement labels, optimizes certain conditional branches, performs loop test copy, performs inductive variable replacement, optimizes loop array addresses. When used with -CLUSTER, makes quick procedure calls where possible.
5	Performs constant propagation, integer constant folding, detects uninitialized variables, eliminates dead computations, optimizes discovered loops, performs straightline array addresses, performs strength reduction with SAVEd and COMMON variables.
6	Provides inline expansion of statement functions. When used with -CLUSTER, provides inline expansion of subroutines.

Note

Each optimization level performs all the optimizations of the next lower level, plus those that are listed.

The level of optimization that you select is identified in the optimization note of the compiler's listing output file.

▶ -OVERFLOW / *-NO_OVERFLOW

Abbreviation: -OVF/ -NOVF

The -OVERFLOW option enables the integer exception handling mechanism when integer arithmetic causes an integer to be larger than the data item to which it is assigned, or a divide by zero is encountered.

-OVERFLOW affects integer calculations only. It causes FIXEDOVERFLOW to be raised at runtime if the result will not fit.

-NO_OVERFLOW does not enable integer overflow conditions.

For example:

```
INTEGER*2 I, J, K
I = 10
J = 32767
K = 0
K = I + J
STOP
END
OK, F77 OVF -OVERFLOW
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]
OK, BIND -LOAD OVF -LI
[BIND rev 19.4]
BIND COMPLETE
OK, RESUME OVF

FIXEDOVERFLOW raised at 4345(3)/1012
(fixed binary)

ERROR raised at 4345(3)/1012
(no on-unit for FIXEDOVERFLOW)
```

Here is an example of a divide by zero encountered at runtime:

```
INTEGER*2 I,J,K
I=10
J=32767
K=0
I=J/K
STOP
END

OK, F77 ZERO -OVERFLOW
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]
OK, BIND -LOAD ZERO -LI
[BIND rev 19.4]
BIND COMPLETE
OK, R ZERO

ZERODIVIDE raised at 4351(3)/1011
(fixed binary)

ERROR raised at 4351(3)/1011
(no on-unit for ZERODIVIDE)
```

Note

If you specify the -OVERFLOW option under certain circumstances, you may receive a FIXEDOVERFLOW error message unexpectedly. If your program

1. Contains INTEGER*2 or INTEGER*4 variables, and

2. Contains an IF statement comparing these variables,

then the computation in the comparison may cause an overflow even though the integers themselves are within the INTEGER*2 or INTEGER*4 range of values. For example, the statements

```
INTEGER*2 I, J
I = 5
J = -32765
IF (I .LT. J) THEN WRITE (1,*) 'THIS WILL NEVER PRINT'
```

generate a runtime overflow message, because the compiler performs the computation

```
5 - (-32765)
```

which yields 32770. In most such cases the solution is either to declare the variables INTEGER*4 or to change the comparison. Rewriting the IF statement above as

```
IF (J .GE.  I) WRITE (1,*) 'ALL OK'
```

solves the problem.

▶ -PBECB / *-NO_PBECB

Abbreviation: -PBECB / -NPBECB

-PBECB causes F77 to place the Entry Control Block (ECB) of each subprogram it compiles into the procedure frame, except for BLOCKDATA subprograms that do not have an ECB. The compiler ignores this option when it is compiling a main program; it will always put a main program's ECB into the link frame.

The -PBECB option is useful for large programs that have many subprograms. On shared systems, users running programs compiled with this option have smaller working sets, and will demand less of system paging resources.

-NO_PBECB does not place Entry Control Blocks in the procedure frame.

Note

It is recommended that you do not use -PBECB when creating an EPF using BIND. Because EPFs have read-only procedure code, they cannot make full use of object files that have been compiled with this option. The procedure will not be shared between users.

▶ -PRODUCTION / *-NO_PRODUCTION

Abbreviation: -PROD / -NPROD

-PRODUCTION generates code for partial debugger functionality. -PRODUCTION is similar to DEBUG , except that the code generated will not permit insertion of statement break points. Execution time increases less than when DEBUG is given.

-NO_PRODUCTION causes no production-type code to be generated.

▶ -RANGE / *-NO_RANGE

Abbreviation: -RA / -NRA

The -RANGE option controls error checking for out-of-bounds values of array subscripts and character substring indexes. Error-checking code is inserted into the object file. If an array subscript or character substring index takes on a value outside the range specified when the referenced data item was declared, a runtime error will be generated. This option is not designed to work on assumed-size arrays.

-NO_RANGE causes no code to be generated to check for out-of-bounds values of subscripts and indexes.

▶ -SAVE

Abbreviation: -SA

-SAVE allocates storage statically. This option is the opposite of the -DYNM option, which allocates storage dynamically. Static storage variables are kept in the link frame. They exist at all times, and maintain their values until the program terminates.

All variables specified in a SAVE statement or initialized in a DATA or type-statement are static. All variables in COMMON are static. This option affects only variables not SAVEd or in COMMON.

▶ *-SILENT [decimal-integer]

Abbreviation: -SI

The -SILENT option, when used with a decimal argument, suppresses the printing of error and warning messages of the severity you specify in decimal-integer. The error and warning messages will be omitted from any listing files generated. Severity levels are listed in Table 9-1.

If no value is given, a value of 1 is assumed. The option header in the listing file will show the level of severity you specify in decimal-integer.

▶ -SOURCE pathname

Abbreviation: -S

This is an obsolete option. -SOURCE specifies the source of the compilation. pathname specifies the name of the source program. If pathname is TTY, then input will come from the terminal. This option should not be used if pathname immediately follows the F77 command.

▶ -SPACE

-SPACE specifies that space reduction is to be given preference over speed in optimization consideration. This option is the opposite of -TIME, which favors speed over space in reducing the size of optimized code.

▶ -STANDARD / *-NO_STANDARD

Abbreviation: -STAN / *-NSTAN

The -STANDARD option allows you to detect PRIME extensions in your program that might inhibit successful compilation on other commercially available FORTRAN 77 compilers.

When invoked, severity 1 warning messages will attempt to flag those portions of FORTRAN 77 code or items that violate ANSI standards which are detectable at compile time. The error messages will be in keeping with the current form and behavior of all error messages currently delivered by F77. See Appendix F for a list of ANSI Standard violations flagged when this option is used.

▶ -STATISTICS / *-NO_STATISTICS

Abbreviation: -STAT / -NSTAT

-STATISTICS displays a list of compilation statistics at the terminal after each phase of compilation. For each phase the list contains:

DISK	Number of reads and writes during the phase, excluding those needed to obtain the source file.
SECONDS	Elapsed real time.
SPACE	Internal buffer space used for symbol table, in 16K byte units.
NODES	The number of symbol table nodes that the compiler is using in the program.
PAGING	Disk I/O time.
CPU	CPU time in seconds, followed by the clock time when the phase was completed.

The -NO_STATISTICS option does not display compilation statistics at the terminal.

Here is an example of compilation statistics generated using the -STATISTICS option:

```
OK, F77 FRIDAY -STATISTICS
[F77 Rev. 19.4]
PHASE          DISK  SECONDS    SPACE    NODES   PAGING      CPU
FORTRAN           0        0        5       73     0.00     0.53  21:19:53
DECLARE           0        0        5       73     0.00     0.09  21:19:53
ALLOCATOR         1        1        5       88     0.00     0.17  21:19:54
VMODE             2        0        6      124     0.00     0.60  21:19:54
TOTAL             3        1        6      124     0.00     1.39  21:19:54

CODE SIZE:                  102
STATIC SIZE:                 34
SOURCE LINES:                16
LINES PER MIN:              671

0000 ERRORS [<.MAIN.> F77 Rev. 19.4]
OK,
```

▶ *-STORE_OWNER_FIELD / -NO_STORE_OWNER_FIELD

Abbreviation: -SOF/ -NSOF

-STORE_OWNER_FIELD stores the identity of the current program in a known place for use by traceback routines. This option is useful for

debugging F77 programs, since utilities such as DMSTK will have access to module names. Use of this option will increase the size of the generated code and linkage and will slightly degrade execution time of user's programs.

The -NO_STORE_OWNER_FIELD option omits this small code sequence for extremely time-critical programs.

▶ *-TIME

The -TIME option specifies that speed is to be given preference over space reduction in optimization consideration. This option is the opposite of -SPACE, which favors space over speed in reducing the size of optimized code.

▶ *-UPCASE / -LCASE

Abbreviation: -UP / -LC

The -UPCASE option treats all lowercase letters in the source program as uppercase, except in Hollerith and CHARACTER constants.

-LCASE distinguishes lower and uppercase characters in the source program. Keywords must be in uppercase only.

▶ -XREF / *-NO_XREF (Implies -LISTING)

Abbreviation: -XR / -NXR

The -XREF option appends a cross reference to the source listing. A cross reference lists, for every symbolic name, the number of every line on which the variable was referenced.

Also, the letter A or M may be appended to a line number. If the letter A appears, it indicates that the symbolic name is an argument. If the letter M appears, it indicates that the variable was modified (that it appeared on the left-hand side of an assignment operator).

On the printout of a cross reference listing, under the column SIZE (DEC), C refers to character, and H refers to half-words.

-NO_XREF does not generate a cross-reference listing.

Figure 9-2 contains an example of a cross-reference listing.

```
OK, F77 FRIDAY -XREF
[F77 Rev. 19.4]
0000 ERRORS [<.MAIN.> F77 Rev. 19.4]

OK, SLIST FRIDAY.LIST
SOURCE FILE: <MONTH>DAY>FRIDAY.F77
COMPILED ON: 850211  AT: 21:21  BY: F77 REV. 19.4
Options selected:  FRIDAY -XREF
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
    "-Full_OPTimize" means "-OPTimize 4", and default is "-OPTimize 2".
Options used(* follows those that are not default):
     64V Allow_PREconnection No_BIG Binary No_DClvar No_DeBuG No_DOl DYnm
     No_ERRList ERRTty No_EXPlist No_FRN No_FTN_Entry INTL Listing* LOGL MAp
     No_OFFset OPTimize(2) NO_OverFlow No_PBECB No_PRODuction No_RAnge
     SIlent(-l) TIME No_STATistics No_Store_Owner_Field UPcase XRef*

    1   c
    2         INTEGER*2 ARRAY(5), TOTAL
    3         DATA ARRAY/10,200,40,55,78/
    4         TOTAL=0
    5         DO 100 J=1,5
    6            TOTAL=TOTAL+ARRAY(J)
    7            IF(J.EQ.3) THEN
    8              WRITE(1,110) TOTAL
    9
   10    110       FORMAT(I4)
   11            ENDIF
   12    100   CONTINUE
   13          WRITE(1,200) TOTAL
   14    200   FORMAT('THE TOTAL OF ARRAY = ',I4)
   15          STOP
   16          END



EXTERNAL ENTRY POINTS

ENTRY POINT       PROGRAM UNIT        LINE        TYPE

.MAIN.                                2           ENTRY REF 2


MAIN PROGRAM .MAIN. ON LINE 2

SYMBOLIC          STORAGE     SIZE    LOC         ATTRIBUTES
NAME              CLASS       (DEC)   (OCT)

100               CONSTANT                        EXECUTABLE LABEL LINE 12 REF 5 12
110               CONSTANT                        FORMAT LABEL LINE 10 REF 8 10
200               CONSTANT                        FORMAT LABEL LINE 14 REF 13 14
ARRAY             STATIC      5H      000024      INTEGER*2 DIMENSION(5) INITIAL RE
                                                  2 3 6
J                 DYNAMIC     2H      000054      INTEGER*4 REF 5 6 7
```

Cross-reference Listing Generated Using -XREF
Figure 9-2

Table 9-2
Summary of Compiler Options and Abbreviations

	Option	Abbreviation	Significance
	-32I		Produce 32I mode code
	-321X		Produce optimized 32I mode code
*	-64V		Produce 64V mode code
*	-ALLOW_PRECONNECTION	-APRE	Use of preopened files
	-BIG	-BIG	Boundary-spanning code
*	-BINARY	-B	Creation of object file
	-CLUSTER	-CLU	Cluster routines for optimization
	-D_STATEMENT	-DSTMT	Treats statements with "D" in column 1 as source code
	-DCLVAR	-DC	Flag undeclared variables
	-DEBUG	-DBG	Debugger code
	-DO1	-DO	FTN DO loops
*	-DYNM	-DY	Dynamic storage default
	-EXTENDED_CHARACTER_SET	-ECS	CHAR returns ECS characters
	-ERRLIST	-ERRL	Create errors-only file
*	-ERRTTY	-ERRT	Write errors to terminal
	-EXPLIST	-EXP	Expanded source listing
	-FRN		Floating point round option
	-FTN_ENTRY	-FTNE	Procedure names passed as arguments, are being passed to FTN
	-FULL_HELP	-FH	Usage information, option list, and description
	-FULL_OPTIMIZE	-FOPT	Full optimization

* Denotes Default Option

Table 9-2 (continued)
Summary of Compiler Options and Abbreviations

	Option	Abbreviation	Significance
	-HELP	-H	Usage information and option list
	-INPUT	-I	Designate source file
*	-INTL	-INTL	Long integer default
	-INTS	-INTS	Short integer default
	-LCASE	-LC	No source-file case conversion
	-LIST	-L	Creation of source listing
*	-LOGL		Long logical-data default
	-LOGS		Short logical-data default
	-MAIN		Main entry point of program
	-MAP	-MA	Listing of data and procedure names
	-MAPWIDE	-MAPW	Specify width of cross-reference map and options list
	-MAXERRORS	-MAXE	Maximum number of errors allowed in a compilation
	-MAX_GROWTH_PERCENT	-MXGR	Specifies suggested limit to growth of program size
	-MAX_SUB_STATEMENTS_INLINE	-MSSI	Provides maximum number of executable statements in subroutines
	-NESTING	-NE	Provides nesting level numbers for statements
	-NO_ALLOW_PRECONNECTION	-NAPRE	No use of preopened files
*	-NO_BIG	-NBIG	No boundary-spanning code
	-NO_BINARY	-NB	No object file

* Denotes Default Option

Table 9-2 (continued)
Summary of Compiler Options and Abbreviations

	Option	Abbreviation	Significance
*	-NO_D_STATEMENT	-NDSTMT	Treats statements with "D" in column 1 as comments
*	-NO_DCLVAR	-NDC	Don't flag undeclared variables
*	-NO_DEBUG	-NDBG	No debugger code
*	-NO_DO1	-NDO	F77 DO loops
*	-NO_EXTENDED_CHARACTER_SET	-NECS	CHAR returns only 7-bit ASCII characters
*	-NO_ERRLIST	-NERRL	No errors-only file
	-NO_ERRTTY	-NERRT	No errors to terminal
*	-NO_EXPLIST	-NEXP	No expanded source listing
*	-NO_FRN	-NFRN	No floating point option
*	-NO_FTN_ENTRY	-NFTNE	Procedure names being passed as arguments are being passed to F77.
*	-NO_LISTING	-NL	No source listing
*	-NO_MAP	-NMA	No listing of data and procedure names
	-NO_NESTING	-NNE	Don't provide block level numbers for statements
*	-NO_OFFSET	-NOFF	No offsets in source listing
*	-NO_OVERFLOW	-NOVF	No integer overflow
*	-NO_PBECB	-NPBECB	Don't load ECBs into procedure frame
*	-NO_PRODUCTION	-NPROD	No production code
*	-NO_RANGE	-NRA	No range checking
*	-NO_STANDARD	-NSTAN	Don't flag ANSI standard violations

* Denotes Default Option

Table 9-2 (continued)
Summary of Compiler Options and Abbreviations

	Option	Abbreviation	Significance
*	-NO_STATISTICS	-NSTAT	Don't print statistics
*	-NO_STORE_OWNER_FIELD	-NSOF	No module names generated
*	-NO_XREF	-NXR	Doesn't generate cross reference
	-OFFSET	-OFF	Offsets in source listing
*	-OPTIMIZE	-OPT	Optimize object code
	-OVERFLOW	-OVF	Enables integer overflow
	-PBECB		Load ECBs into procedure frame
	-PRODUCTION	-PROD	Generate production code
	-RANGE	-RA	Check subscript ranges
	-SAVE	-SA	Static storage default
*	-SILENT	-SI	Suppress warning messages (default is level 1)
	-SOURCE	-S	Designate source file
	-SPACE		Space over time in optimization
	-STANDARD	-STAN	Flag ANSI standard violations
	-STATISTICS	-ST	Print compiler statistics
	-STORE_OWNER_FIELD	-SOF	Module names are generated into program code for debugging use
*	-TIME		Time over space in optimization
*	-UPCASE	-UP	Convert to uppercase
	-XREF	-XR	Generate cross-reference

* Denotes Default Option

10

Linking and Executing Your Program

After you have successfully created an object file using the F77 compiler, you are ready to link and execute your program using the PRIMOS utilities BIND and RESUME. This chapter discusses:

- How to use BIND to link your program.
- How to use the RESUME command to get your program running.

BIND

BIND is a linking utility that creates an executable program format runfile from an object file. This runfile, known as an Executable Program Format (EPF), is the executable version of your program. Some advantages of EPFs are:

- They are dynamic; they can execute in any segment, or segments, of PRIMOS.
- They do not need to use the same segment each time they are invoked.
- They are RESUMEable.
- Several programs can exist in memory at one time without overwriting each other.
- Programs can call other programs.

For more information on EPFs, see the Programmer's Guide to BIND and EPFs.

USING BIND

You can use BIND to create an EPF in one of two ways:

1. Interactively by invoking subcommands of BIND. (This is the subsystem form.)

2. Directly from the PRIMOS command line. (This is the command form.)

Note

To allow BIND to operate most efficiently, your object file should have a .BIN suffix. If you follow the compilation steps as explained in Chapter 9, the F77 compiler will automatically generate an object file with a .BIN suffix. Naming conventions are discussed in ABOUT THIS BOOK.

Using BIND Interactively

To invoke BIND interactively, type the command

```
BIND [EPF-filename]
```

where:

EPF-filename is the name of the EPF that you want BIND to create. BIND saves the runfile in a directory that you specify with the name EPF-filename.RUN. If you do not specify the EPF-filename, BIND adds the .RUN suffix to the first object file that you load, saving the runfile in the directory you specify.

You have now entered the BIND subsytem. You know you're there when you see the colon prompt. You will see this prompt each time you press the carriage return until you leave BIND. That's when your system prompt will return.

Here is an example of using BIND interactively:

```
OK, BIND MYF77_PROG          /*invoke BIND and name runfile
[BIND rev 19.4]              /*BIND version number
: LOAD TEST                  /*load TEST.BIN
: LOAD SUB1                  /*load SUB1.BIN
: LI                         /*load system libraries
BIND COMPLETE                /*load completion message
: FILE                       /*save the EPF, and,
OK,                          /*return control to PRIMOS
```

In this example, the EPF is filed in your directory and has the name MYF77_PROG.RUN. If you do not specify a filename for the EPF when you invoke BIND, BIND automatically takes the name of the first object file that you load and adds a .RUN suffix.

Using BIND From the Command Line

To create an EPF runfile from the PRIMOS command line, type the command:

```
BIND [EPF-filename] [ -options]
```

where:

EPF-filename is used in the same way as when you invoke BIND interactively as a subsystem.

options given on the command line correspond to internal BIND commands. You must precede each option with a hyphen.

Here is an example of using BIND on the command line:

```
OK, BIND MYF77_PROG -LOAD TEST SUB1 -LI
    BIND rev 19.4]
    BIND COMPLETE
    OK,
```

In the above example, BIND creates an EPF that has the name MYF77_PROG.RUN containing the linked object files, TEST.BIN and SUB1.BIN. At this point, BIND displays the message BIND COMPLETE and returns control to PRIMOS.

Basic BIND Commands

You can link your F77 program modules with the following basic BIND commands:

- LOAD
- LIBRARY
- FILE
- MAP
- QUIT
- HELP

For a brief description and example of each of these commands, see Table 10-1.

Table 10-1
BIND: Basic Commands

Command	Description	Example
LOAD	Loads one or more object files	: Lo test
LIBRARY	Loads one or more libraries	: Li MYLIB
FILE	Saves runfile	: File MYPROG
MAP	Checks for unresolved references	: MAP -UN
QUIT	Ends bind session without saving runfile	: QUIT
HELP	Gives online help on bind commands and options	: HELP -LOAD

Using the LOAD command: This command links your program, starting with the main procedure and followed by subprograms in any order. The LOAD command has the following format:

```
LOAD pathname-1 [pathname-2 pathname-3...]
```

Each pathname is the name of an object file that you want to bind to the current EPF. If there is no existing EPF, and you do not specify one on the command line, BIND creates one with the name you give in the first pathname.

For example:

```
: LOAD HOME                    /*links the object file
                                 with the name HOME.BIN, and
                                 files the runfile HOME.RUN
                                 in your directory.
```

Using the LIBRARY Command: This command is used for linking the standard system libraries needed by F77, as well as libraries that you create. The LIBRARY command has the following format:

```
LIBRARY [library-name-1 library-name-2 library-name-3...]
```

When you issue the LIBRARY command without specifying a library-name, BIND automatically links the standard system libraries that are kept in a directory called LIB. To link libraries that you have created, you must specify the pathname of the file in library-name.

For example:

```
: LI SAMPLE>MYLIB              /*links the file MYLIB.BIN
                                 from the directory SAMPLE.
                                 (You must link the libraries
                                 created by you before
                                 system libraries.)

: LI                           /*links the standard system
                                 libraries
```

Once the standard system libraries have been linked, you should receive a BIND COMPLETE message. If you don't, use the MAP command to identify any unresolved references.

Using the FILE Command: The FILE command saves the EPF runfile to disk. The FILE command has the following format:

```
FILE [EPF-filename]
```

BIND responds to the FILE command by processing the EPF and filing the runfile in your dirctory with a .RUN suffix. If you already have an EPF with the same name, BIND will overwrite the existing EPF. When BIND completes processing the EPF, control is returned to PRIMOS.

For example:

```
OK, BIND                        /*invoke BIND interactively
[BIND REV 19.4]                 /*BIND version number
: LOAD TEST                     /*link TEST.BIN
: LI                            /*link standard system libraries
BIND COMPLETE                   /*BIND completion message
: FILE NEW_TEST                 /*file the EPF in your directory
                                  with the filename NEW_TEST.RUN
OK,                             /*return control to PRIMOS
```

If you are using BIND from the command line, you do not have to specify the file command. By default, BIND will add the FILE command to the end of the command line:

```
OK,BIND -LO TEST -LI
[BIND rev 19.4]
BIND COMPLETE
OK,
```

BIND automatically adds a FILE command and saves the runfile in your directory with the name TEST.RUN.

Using the MAP Command: Use the MAP command to get a load map of any unresolved subroutine, program, or common block references. The MAP command has the following format:

```
MAP [pathname] [option]
```

pathname specifies that the map is written to a file instead of being printed at your terminal. The most useful option you can use with the MAP command is the -UNDEFINED option. This option lists only the unresolved references in your program. For a complete list of options that you can use with this command, see the Programmer's Guide to BIND and EPFs.

For example:

```
: MAP MYMAP                    /*writes a standard map of
                                 your program to the file
                                 MYMAP

: MA -UN                       /*displays a list of the
                                 unresolved references
```

Using the QUIT Command: You can use the QUIT command to return to PRIMOS without completing the binding process. The QUIT command used as an option on the command line causes BIND not to create an EPF. The QUIT command has the following format:

```
QUIT
```

The QUIT command ends a BIND session without saving the current EPF. BIND asks you for a verification if the EPF is not filed before returning to PRIMOS.

For example:

```
OK, BIND                       /*invoke BIND interactively
: LOAD TEST                    /*link TEST.BIN
: LI                           /*link system libraries
BIND COMPLETE                  /*BIND completion message
: QUIT                         /*terminate binding session
EPF not filed, ok to quit? ('Yes', 'Y', 'No', 'N'): Y  /*verify
OK,                            /*return control to PRIMOS
```

Using the HELP Command: BIND has a built-in help facility that you can use when you are working interactively. The HELP command has the form:

```
HELP [command-name] [-LIST]
```

If you use the -LIST option, BIND will display a list of all the commands available. To get help on a specific command, use this command followed by the command name.

For example:

```
OK, BIND
[BIND rev 19.4]
: HELP FILE
FILE [<epfname>]
     will exit to PRIMOS after filing the EPF.
     If <epfname> is specified, the EPF will be named <epfname>.RUN
: Q
OK,
```

RESUME

Once your program has been compiled, and an EPF runfile has been created with BIND, you are ready to run your program using the RESUME command. This command has the following format:

```
RESUME [EPF-filename]
```

The RESUME command looks in your directory for the file you specify in EPF-filename having .RUN suffix and begins execution of the EPF. The finename must appear in the directory with a .RUN suffix or PRIMOS returns with a NOT FOUND message. After you create the runfile you will be able to run the program.

For example:

```
OK, RESUME MYF77_PROG
```

11

Finding and Correcting Runtime Errors

This chapter introduces you to Prime's Source Level Debugger, DBG. The Debugger is a separately priced product that you can use to help you find out why your program failed at runtime. Working interactively with the Debugger, you can stop the execution of your program at critical points and examine the contents of program variables to see if they're correct.

The DBG commands that you will learn about in this chapter are:

- DBG
- RESTART
- SOURCE
- BREAKPOINT
- CONTINUE
- : , TYPE, and LET
- WATCH
- HELP and QUIT

This chapter will not teach you everything about the Debugger. You will only learn enough to debug a simple F77 program. For a more detailed discussion on this powerful tool, see the Source Level Debugger's User's Guide.

HOW TO USE THE DEBUGGER

Before you can begin using DBG, you must first complete the following steps:

1. Create and edit your program using ED or EMACS.
2. Compile your program using the -DEBUG compiler option.
3. Link your program using BIND.

Let's use a program called TEST.F77 to illustrate steps 2 and 3.

```
OK, F77 TEST -DEBUG                      /*Compile your program
[F77 Rev. 19.4]                            with the -DEBUG option.
0000 ERRORS [<.MAIN.> F77-REV 19.2]        No errors.

OK, BIND -LOAD  TEST -LI                 /*Link your program
[BIND rev 19.4]                            with BIND.  Again, no
BIND COMPLETE                              errors.

OK, RESUME TEST                          /*Execute your program
                                           with RESUME.
 THIS IS AN F77 TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5
 X + Y=    5.00
**** STOP

OK,
```

As you can see, even though there were no errors at compilation or load time, your program produces the wrong results. At this point you can use the Debugger to find out what happened at runtime.

Entering the Debugger

To enter the Debugger from the PRIMOS command level, type the following command:

```
DBG EPF-filename
```

where:

EPF-filename is the name of the executable program file you want to debug.

Once you have entered the Debugger subsystem, DBG will prompt with a right angle bracket prompt. You will also see some Debugger software identification information displayed. Here is an example of invoking the Debugger to debug TEST.F77.

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

>
```

At this point the debugger is waiting for you to enter commands in response to the right angle bracket prompt.

RUNNING YOUR PROGRAM WITHIN THE DEBUGGER

To start the execution of your program from within the Debugger, use the RESTART command. This command restarts program execution at any point within the Debugger. Here is an example of the RESTART command:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> RESTART
 THIS IS AN F77 PROGRAM TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5
 X + Y=    5.00
**** STOP


Program stop at $MAIN\9.
>
```

You can use this command whenever you want to restart execution of your program at the beginning of the main procedure.

Note

If you use the RESTART command after execution of your program begins, the variables initialized in a DATA statement will not be reinitialized.

LOOKING AT YOUR SOURCE PROGRAM

If you want to take a look at the source code of your program without leaving the Debugger subsystem, use the SOURCE command. This command, with it's EDITOR-like subcommands, allow you to move around your source program. The SOURCE command has the following format:

```
SOURCE source-command [argument]
```

where:

source-command is any EDITOR subcommand that can be used with SOURCE.

argument is an EDITOR source subcommand object that may or may not be used. A line number or text string are examples of argument.

Table 11-1 gives a list of the most frequently used source EDITOR subcommands that you will use with the SOURCE command. For more information on the source EDITOR commands, see THE NEW USER'S GUIDE TO EDITOR AND RUNOFF.

Table 11-1

SOURCE Subcommands (From ED)
(abbreviations are underlined)

Subcommand	Description
TOP	Position line pointer to top of file.
BOTTOM	Position pointer to bottom of file.
PRINT	Print line(s).
WHERE	Print current line number.
POINT	Position to specific line.
NEXT	Move line pointer forward or backward.
LOCATE	Locate line with the specified text string.
FIND	Locate line with the specified text string beginning in a given column.

The following example shows how to use the SOURCE command with the PRINT, TOP, POINT, and NEXT subcommands:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> SOURCE PRINT 23
    1:        WRITE(1,*)'THIS IS AN F77 PROGRAM TO DBG TEST'
    2:        WRITE(1,*)'ENTER A VALUE FOR X:  '
    3:        READ(1,*)X
    4:        WRITE(1,*)'ENTER A VALUE  FOR Y:  '
    5:        READ(1,*)Y
    6:        XY = X
    7:        WRITE(1,10)XY
    8: 10     FORMAT(1X,'X + Y= ',1X,F6.2)
    9:        STOP
   10:        END
BOTTOM
> SOURCE TOP
> SOURCE POINT 5
    5:        READ(1,*)Y
> SOURCE NEXT
    6:        XY = X
>
```

By looking at your source program, you notice that you have coded your program incorrectly. The expression in line number 6 is not correct; it should read XY = X + Y. When you reedit, recompile and relink your program, the value for XY will be correct. However, for the sake of illustrating a few more basic debugger commands, we're not going to change the source code.

STOPPING EXECUTION OF YOUR PROGRAM

By using the BREAKPOINT command, you can suspend the execution of your program and take a look at the data at that point. The BREAKPOINT command has the following format:

```
BREAKPOINT breakpoint-identifier
```

where:

breakpoint-identifier is the number of the line in your program where you want to suspend execution. Execution will stop immediately before the line you specify in breakpoint-identifier. You can find the source line number by using the SOURCE command.

If you try to suspend execution on a non-executable statement, the Debugger will issue an error message.

Here is an example of how the BREAKPOINT command works:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> SOURCE PRINT 23
    1:        WRITE(1,*)'THIS IS AN F77 PROGRAM TO DBG TEST'
    2:        WRITE(1,*)'ENTER A VALUE FOR X:  '
    3:        READ(1,*)X
    4:        WRITE(1,*)'ENTER A VALUE  FOR Y:  '
    5:        READ(1,*)Y
    6:        XY = X
    7:        WRITE(1,10)XY
    8: 10     FORMAT(1X,'X + Y= ',1X,F6.2)
    9:        STOP
   10:        END
BOTTOM
> BREAKPOINT 6
> RESTART
 THIS IS AN F77 PROGRAM TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5

**** breakpointed at $MAIN\6
```

You have suspended the execution of your program just before line number 6.

CONTINUING EXECUTION OF YOUR PROGRAM

To begin program execution once again after setting a breakpoint in your program, use the CONTINUE command.

Here is an example of how to use the CONTINUE command:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> BREAKPOINT 6
> RESTART
 THIS IS AN F77 PROGRAM TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5

**** breakpointed at $MAIN\6
> CONTINUE
 X + Y=    5.00
**** STOP


Program stop at $MAIN\9.
>
```

EXAMINING AND MODIFYING DATA

There are three useful commands for looking at and changing the data in your program: the colon (:), TYPE, and LET commands.

Using the : Command

The : command is used to look at the value of a variable or expression while your program is suspended. To use this command, specify a variable or expression after you issue the : character. The : must be followed by a space.

Here is an example of how the : command works:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1   (5-March-1984)

> BREAKPOINT 6
> RESTART
 THIS IS AN F77 PROGRAM TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5

**** breakpointed at $MAIN\6
> : X
X = 5.000000E+00
> : Y
Y = 5.000000E+00
>
```

In the above example you can see that both X and Y, when evaluated with the : command, have the value that was entered for them during program execution.

Using the TYPE Command

The TYPE command is used to evaluate the data type of a variable or expression. Frequently, a program fails due a data type mismatch.

Here is an example of the how to use the TYPE command:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1   (5-March-1984)

> BREAKPOINT 6
> RESTART
 THIS IS AN F77 PROGRAM TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5

**** breakpointed at $MAIN\6
> TYPE X
real*4 automatic
> TYPE Y
real*4 automatic
>
```

Using the LET Command

The LET command allows you to assign a new value to a variable. By doing this, you can test to see what would happen to the execution of your program with these new values. To use the LET command, you assign an expression to a variable with an equals sign(=).

Here is how the LET command works:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> BREAKPOINT 6
> RESTART
 THIS IS AN F77 TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5

**** breakpointed at $MAIN\6
> : X
X = 5.000000E+00
> LET X = 10
> : X
X = 1.000000E+01
> CONTINUE
 X + Y=    10.00
**** STOP


Program stop at $MAIN\9.
>
```

VALUE TRACING

While you're in the Debugger subsystem, you can use the WATCH command to watch or trace a variable to see how it changes during program execution.

Here is an example of the WATCH command:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> RESTART
 THIS IS A F77 TO DBG TEST
 ENTER A VALUE FOR X:
5
 ENTER A VALUE  FOR Y:
5
 X + Y=     5.00
**** STOP


Program stop at $MAIN\9.
> WATCH X
> RESTART
 THIS IS AN F77 TO DBG TEST
 ENTER A VALUE FOR X:
6
The value of $MAIN\X has been changed at $MAIN\4
  from 5.000000E+00
  to   6.000000E+00
 ENTER A VALUE  FOR Y:
5
 X + Y=     6.00
**** STOP


Program stop at $MAIN\9.
>
```

GETTING HELP

If you run into trouble while you are working within the Debugger subsystem, you can use the HELP command to get online help. The HELP command has the following format:

```
        -LIST
HELP    -SYM_LIST
        command-name
        -syntax-symbol
```

where:

-LIST prints a list of all DBG commands.

-SYM_LIST prints a list of syntax symbols.

command-name prints the syntax of command-name.

syntax-symbol prints the definition of syntax symbol.

For example:

```
OK, DBG TEST

**Dbg**   revision 1.1 - 19.1  (5-March-1984)

> HELP TYPE                    /*command line syntax of
TYPE  <expression>              TYPE command.
> HELP EXPRESSION              /*definition of syntax
<EXPRESSION>:                   symbol.
any valid expression in the default evaluation language
>
```

HOW TO LEAVE THE DEBUGGER

When you are ready to leave the Debugger and return to PRIMOS level, use the QUIT command.

Here is an example of the QUIT command:

```
> QUIT
OK,
```

You are now at PRIMOS command level.

FOR MORE INFORMATION...

This chapter only gave you an introduction to using the Debugger. There are many more features available with the Debugger that will greatly expedite and simplify the process of debugging your programs. For a thorough discussion of the Debugger, see the Source Level Debugger User's Guide.

12
Optimizing F77 Programs

This chapter presents programming suggestions for improving the performance of F77 programs. Some are reminders of good coding practice. Others take advantage of implementation techniques in the F77 compiler. All offer some speedup in program execution.

Multidimensional Arrays

Reference memory as sequentially as possible. For multidimensional arrays, the leftmost subscript varies the fastest in FORTRAN 77. For addressing large portions of an array, paging time and working set size can be significantly reduced by indexing the leftmost subscript the fastest (e.g., in the innermost loop). Thus,

```
      DIMENSION ARRAY (100,100)
      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 array as ARRAY (I, J) = 3.0.

If the program can be designed efficiently without multidimensional arrays, memory addressing can be more efficient. For more than one dimension, this saves one multiply per effective address calculation; i.e., number-of-multiplies = number-of-dimensions - 1. For instance, the example above could be written as:

```
      DIMENSION ARRAY (100,100)
      DIMENSION INITARRAY (1)
      EQUIVALENCE (ARRAY(1,1), INITARRAY(1))

      DO 10 I = 1, 10000
         INITARRAY(I) = 3.0
   10 CONTINUE
```

saving considerable CPU time.

Loading and Memory Allocation

Paging time can be significantly reduced by loading subprograms by frequency of use (rather than, say, alphabetically). The main program must always be loaded first for BIND to work properly.

A suitable loading scheme would allocate memory as:

```
MAIN
 ..
 ..
END

 .
 .                    most common subroutines
 .

 .
 .                    occasionally used subroutines
 .

 .
 .                    infrequently used subroutines
```

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.

For more information on using BIND, see the Programmer's Guide to BIND and EPFs.

Function Calls

When using function calls, eliminate redundant invocations of user-supplied functions. For example:

```
TEMP = FUNC(X)
A = TEMP * TEMP
```

is faster than:

```
A = FUNC(X) * FUNC(X)
```

Make sure that the function has no side effects which might modify the argument(s) or anything else in the environment.

This practice is not necessary with intrinsic functions unless optimization of the program unit is prevented by the -NO_OPTIMIZE compiler option, because the F77 optimizer eliminates redundant intrinsic function calls.

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. (These routines are listed in the Subroutines Reference Guide.) 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?
 * 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 short integer named N. For printing out text efficiently, use the TNOU/TNOUA routines. See the Subroutines Reference Guide 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 is 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, resulting in faster execution of statements using the named constants. The compiler does not fold normal variables initialized by DATA statements into constants.

Library Calls

Some applications library routines are not optimized for time critical operations. The get and store character routines (GCHR$A, etc.) are convenient, but comparatively slow. Some applications library routines are by definition slow, because they use lower-level routines which can more efficiently be called directly. Avoid using the MAX and MIN functions when execution time must be minimized.

Applications library subroutines are designed to perform acceptably at any task for which they might be called. When one particular task is often required in a program, a user-supplied routine which is maximally efficient at that one task can be substituted. See Chapter 8 and the EXTERNAL Statement in Chapter 3.

Remember the 80/20 rule, which states: "80 percent of a program's time is spent in 20 percent of the code." Therefore, standard library routines are adequate in the non-time-critical 80 percent of the program.

Integer Division

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
```

Compiler Options

The following compiler options allow your program to execute faster and more efficiently:

- -OPTIMIZE
- -CLUSTER
- -TIME
- -DYNM

Use of the -OPTIMIZE Option: This option allows you the choice of the following levels of optimization:

0: Perform no optimizations. Turns optimization off.

1: Code pattern replacement.

2: Common subexpression elimination. (This is the default value.)

3: Loop invariant removal.

4: Strength reduction of some common operations including indexing of large arrays. Elimination of unreachable code.

Use of the -CLUSTER Option: When the -CLUSTER option is specified on the command line for F77 in addition to optimization level 4, all the subroutines in the file being compiled will become candidates to be made quick.

Internally-nested procedures will be made quick, that is, called by a Jump to Subroutine instruction rather than a Procedure call, if conditions allow. The conditions under which a procedure will be made quick are that it be called from only one place in your program. For example, procedure C can be quick if it's called from procedure A. However, if procedure C is also called from procedure B, where B is a separate procedure from A, then C cannot be quick.

Use of the TIME Option: This option specifies that time is to be given preference over space in optimization consideration.

The -TIME option is the default.

Use of the -DYNM Option: Prime F77 programs run more efficiently when local variables are placed in the stack through the use of the -DYNM option (the default). These variables are not guaranteed to be valid after a return.

Conclusion

These are some of the more common guidelines for programming in Prime F77. If you keep these ideas in mind while writing, or while "fine tuning" FORTRAN 77 programs, your programs will generally be smaller and faster. Some of these rules are not necessarily permanent. As Prime F77 evolves more optimizations, you will have more freedom to choose programming 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 text editor commands, others may require extensive changes to the source code.

Only specific techniques that can be described fairly briefly are mentioned in this chapter. Many other examples of good programming practice, and an excellent discussion of the more general aspects of good programming, appear in the following text:

Kernighan and Plauger, The Elements of Programming Style, McGraw-Hill, 1974

APPENDIXES

A

Prime Extended Character Set

As of Revision 21.0, Prime has expanded its character set. The basic character set remains the same as it was before Revision 21.0; it is the ANSI ASCII 7-bit set, with the 8th bit always on. However, the 8th bit is now significant; when it is turned off, it signifies a different character. Thus the size of the character set has doubled from 128 characters to 256 characters. This expanded character set is called the Prime Extended Character Set (Prime ECS).

The pre-Revision 21.0 character set is a proper subset of Prime ECS. These characters have not changed. Software written before Revison 21.0 continues to run exactly as it did before. Software written at Revision 21.0 that does not use the new characters requires no special coding to use the old ones.

Prime ECS support is automatic at Revison 21.0. You can begin to use characters that have the 8th bit turned off. However, the extra characters are not available on most printers and terminals. Check with your System Administrator to find out whether you can take advantage of the new characters in Prime ECS.

Table A-1 shows the Prime Extended Character Set. The pre-Revision 21.0 character set consists of the characters with decimal values 128 through 255 (octal values 200 through 377). The characters added at Revision 21.0 all have decimal values less than 128 (octal values less than 200).

SPECIFYING PRIME ECS CHARACTERS

Direct Entry

On terminals that support Prime ECS, you can enter the printing characters directly; the characters appear on the screen as you type them. For information on how to do this, see the appropriate manual for your terminal.

A terminal supports Prime ECS if

1. It uses ASCII-8 as its internal character set, and
2. The TTY8 protocol is configured on your asynchronous line.

If you do not know whether your terminal supports Prime ECS, ask your System Administrator.

On terminals that do not support Prime ECS, you can enter any of the ASCII-7 printing characters (characters with a decimal value of 160 or higher) directly by typing them.

Octal Notation

If you use the Editor (ED), you can enter any Prime ECS character by typing:

```
^octal-value
```

where octal-value is the three-digit octal number given in Table A-1. You must type all three digits, including leading zeroes.

Before you use this method to enter any of the ECS characters that have decimal values between 32 and 127, first specify the following ED command:

```
MODE CKPAR
```

This command permits ED to print as ^nnn any characters that have a first bit of 0.

Character String Notation

You can specify Prime ECS characters on any terminal by using one of the notations shown below. However, the characters themselves can only appear on a terminal that supports Prime ECS. Other terminals will not display the new characters correctly.

The following rules describe how to specify Prime ECS characters in character strings.

1. You can specify printing characters in character strings by enclosing them in single quotation marks ('). For example:

   ```
   'Quoted string'
   ```

   You can enter the characters using either direct entry or octal notation as described in the beginning of this section.

2. You can specify any character in Prime ECS that has a mnemonic as follows:

   ```
   \(mnemonic)
   ```

   where mnemonic is the Prime mnemonic shown for that character in Table A-1. You can specify the mnemonic with either uppercase or lowercase characters. Some characters have more than one mnemonic; you may use any one of these. In the table, the alternatives are separated by a slash character (/). For example:

   ```
   'A string'\(FF)'with a form feed in it'
   ```

   The compiler interprets the above example as a single character string.

3. You can specify certain frequently used nonprinting characters as

   ```
   \(abbreviation)
   ```

   where abbreviation is one of the following:

Abbreviation	Meaning
B	Backspace
E	Escape
F	Form feed
L	Line feed
N	New line
R	Carriage return
T	Horizontal tab
V	Vertical tab

For example:

```
'A string'\F'with a form feed in it'
```

4. You can specify control characters as

```
\^character
```

where ^character is listed under "Graphic" in Table A-1. For example:

```
'A string'\^L'with a form feed in it'
```

A character specified with a backslash (that is, with notation 2, 3, or 4)

- Must appear outside quotation marks
- Specifies a character string of length 1
- Can be specified by itself, or with one or more additional backslash-notation characters, or juxtaposed with one or more quoted character strings.

Spaces between the Prime ECS character specification and the character string are not significant, but there must be no spaces within the character specification itself.

Program Example

The following program example writes a string that is specified by Prime ECS syntax:

```
      PROGRAM ECS_STRING
*
      CHARACTER*12 STRING
*
      STRING = \(CR) 'HELLO' \n 'THERE'
*
      PRINT*, STRING
*
      STOP
      END
```

This program produces the following output:

```
HELLO
     THERE
**** STOP
```

SPECIAL MEANINGS OF PRIME ECS CHARACTERS

PRIMOS, or an applications program running on PRIMOS, may interpret some Prime ECS characters in a special way. For example, PRIMOS interprets ^P as a process interrupt. ED, the Editor, interprets the backslash (\) as a logical tab. If you wish to make use of the Prime ECS backslash character in a file you are editing with ED, you must define another character as your logical tab.

For a detailed description of how PRIMOS interprets the following Prime ECS characters, see the discussion in the Prime User's Guide of special terminal keys and special characters: ^ \ " ? ^P ^S ^Q _ and ;.

F77 PROGRAMMING CONSIDERATIONS

Remember that identifiers and program names may contain only letters, numbers, and the dollar sign and underscore characters ($ and _). These characters form a subset of the ASCII-7 character set.

Character strings, however, can contain any character in Prime ECS. Such strings can be declared as constants, written, read, or assigned to CHARACTER variables.

You can use notations 2, 3, and 4, described above, in any quoted string in your program. Thus, you can use these rules in constant declarations, assignment, and write or print statements.

You cannot use notations 2, 3, and 4 in identifiers or in terminal or file input. Therefore, if your terminal does not support Prime ECS, you can enter as terminal input only those characters with decimal values greater than 127 (octal values greater than 177).

The new characters in Prime ECS, decimal values 000 through 127 (octal values 000 through 177) specified with notations 2, 3, and 4 above, cannot be juxtaposed with Hollerith-style constants. They may not be used in FORMAT statements, or used in runtime formats.

PRIME EXTENDED CHARACTER SET TABLE

Table A-1 contains all of the Prime ECS characters, arranged in ascending order. This order provides both the collating sequence and the way that comparisons are done for character strings. For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal values. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^character; for example, ^P represents the character produced when you type P while holding the control key down.

Characters with decimal values from 000 to 031 and from 128 to 159 are control characters.

Characters with decimal values from 032 to 127 and from 160 to 255 are graphic characters.

The pre-Revision 21.0 character set consists of the characters with decimal values 128 through 255 (octal values 200 through 377). The characters added at Revision 21.0 all have decimal values less than 128 (octal values less than 200).

Table A-1
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	RES1	Reserved for future standardization	0000 0000	000	00	000
	RES2	Reserved for future standardization	0000 0001	001	01	001
	RES3	Reserved for future standardization	0000 0010	002	02	002
	RES4	Reserved for future standardization	0000 0011	003	03	003
	IND	Index	0000 0100	004	04	004
	NEL	Next line	0000 0101	005	05	005
	SSA	Start of selected area	0000 0110	006	06	006
	ESA	End of selected area	0000 0111	007	07	007
	HTS	Horizontal tabulation set	0000 1000	008	08	010
	HTJ	Horizontal tab with justify	0000 1001	009	09	011
	VTS	Vertical tabulation set	0000 1010	010	0A	012
	PLD	Partial line down	0000 1011	011	0B	013
	PLU	Partial line up	0000 1100	012	0C	014
	RI	Reverse index	0000 1101	013	0D	015
	SS2	Single shift 2	0000 1110	014	0E	016
	SS3	Single shift 3	0000 1111	015	0F	017
	DCS	Device control string	0001 0000	016	10	020
	PU1	Private use 1	0001 0001	017	11	021
	PU2	Private use 2	0001 0010	018	12	022
	STS	Set transmission state	0001 0011	019	13	023
	CCH	Cancel character	0001 0100	020	14	024
	MW	Message waiting	0001 0101	021	15	025
	SPA	Start of protected area	0001 0110	022	16	026
	EPA	End of protected area	0001 0111	023	17	027
	RES5	Reserved for future standardization	0001 1000	024	18	030
	RES6	Reserved for future standardization	0001 1001	025	19	031
	RES7	Reserved for future standardization	0001 1010	026	1A	032
	CSI	Control sequence introducer	0001 1011	027	1B	033
	ST	String terminator	0001 1100	028	1C	034
	OSC	Operating system command	0001 1101	029	1D	035
	PM	Privacy message	0001 1110	030	1E	036

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	APC	Application program command	0001 1111	031	1F	037
	NBSP	No-break space	0010 0000	032	20	040
¡	INVE	Inverted exclamation mark	0010 0001	033	21	041
¢	CENT	Cent sign	0010 0010	034	22	042
£	PND	Pound sign	0010 0011	035	23	043
¤	CURR	Currency sign	0010 0100	036	24	044
¥	YEN	Yen sign	0010 0101	037	25	045
¦	BBAR	Broken bar	0010 0110	038	26	046
§	SECT	Section sign	0010 0111	039	27	047
¨	DIA	Diaeresis, umlaut	0010 1000	040	28	050
©	COPY	Copyright sign	0010 1001	041	29	051
ª	FOI	Feminine ordinal indicator	0010 1010	042	2A	052
«	LAQM	Left angle quotation mark	0010 1011	043	2B	053
¬	NOT	Not sign	0010 1100	044	2C	054
	SHY	Soft hyphen	0010 1101	045	2D	055
®	TM	Registered trademark sign	0010 1110	046	2E	056
¯	MACN	Macron	0010 1111	047	2F	057
°	DEGR	Degree sign	0011 0000	048	30	060
±	PLMI	Plus/minus sign	0011 0001	049	31	061
²	SPS2	Superscript two	0011 0010	050	32	062
³	SPS3	Superscript three	0011 0011	051	33	063
´	AAC	Acute accent	0011 0100	052	34	064
µ	LCMU	Lowercase Greek letter µ, micro sign	0011 0101	053	35	065
¶	PARA	Paragraph sign, Pilgrow sign	0011 0110	054	36	066
·	MIDD	Middle dot	0011 0111	055	37	067
¸	CED	Cedilla	0011 1000	056	38	070
¹	SPS1	Superscript one	0011 1001	057	39	071
º	MOI	Masculine ordinal indicator	0011 1010	058	3A	072
»	RAQM	Right angle quotation mark	0011 1011	059	3B	073
¼	FR14	Common fraction one-quarter	0011 1100	060	3C	074

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
½	FR12	Common fraction one-half	0011 1101	061	3D	075
¾	FR34	Common fraction three-quarters	0011 1110	062	3E	076
¿	INVQ	Inverted question mark	0011 1111	063	3F	077
À	UCAG	Uppercase A with grave accent	0100 0000	064	40	100
Á	UCAA	Uppercase A with acute accent	0100 0001	065	41	101
Â	UCAC	Uppercase A with circumflex	0100 0010	066	42	102
Ã	UCAT	Uppercase A with tilde	0100 0011	067	43	103
Ä	UCAD	Uppercase A with diaeresis	0100 0100	068	44	104
Å	UCAR	Uppercase A with ring above	0100 0101	069	45	105
Æ	UCAE	Uppercase diphthong Æ	0100 0110	070	46	106
Ç	UCCC	Uppercase C with cedilla	0100 0111	071	47	107
È	UCEG	Uppercase E with grave accent	0100 1000	072	48	110
É	UCEA	Uppercase E with acute accent	0100 1001	073	49	111
Ê	UCEC	Uppercase E with circumflex	0100 1010	074	4A	112
Ë	UCED	Uppercase E with diaeresis	0100 1011	075	4B	113
Ì	UCIG	Uppercase I with grave accent	0100 1100	076	4C	114
Í	UCIA	Uppercase I with acute accent	0100 1101	077	4D	115
Î	UCIC	Uppercase I with circumflex	0100 1110	078	4E	116
Ï	UCID	Uppercase I with diaeresis	0100 1111	079	4F	117
Ð	UETH	Uppercase Icelandic letter Eth	0101 0000	080	50	120
Ñ	UCNT	Uppercase N with tilde	0101 0001	081	51	121
Ò	UCOG	Uppercase O with grave accent	0101 0010	082	52	122
Ó	UCOA	Uppercase O with acute accent	0101 0011	083	53	123

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
Ô	UCOC	Uppercase O with circumflex	0101 0100	084	54	124
Õ	UCOT	Uppercase O with tilde	0101 0101	085	55	125
Ö	UCOD	Uppercase O with diaeresis	0101 0110	086	56	126
×	MULT	Multiplication sign used in mathematics	0101 0111	087	57	127
Ø	UCOO	Uppercase O with oblique line	0101 1000	088	58	130
Ù	UCUG	Uppercase U with grave accent	0101 1001	089	59	131
Ú	UCUA	Uppercase U with acute accent	0101 1010	090	5A	132
Û	UCUC	Uppercase U with circumflex	0101 1011	091	5B	133
Ü	UCUD	Uppercase U with diaeresis	0101 1100	092	5C	134
Ý	UCYA	Uppercase Y with acute accent	0101 1101	093	5D	135
Þ	UTHN	Uppercase Icelandic letter Thorn	0101 1110	094	5E	136
ß	LGSS	Lowercase German letter double s	0101 1111	095	5F	137
à	LCAG	Lowercase a with grave accent	0110 0000	096	60	140
á	LCAA	Lowercase a with acute accent	0110 0001	097	61	141
â	LCAC	Lowercase a with circumflex	0110 0010	098	62	142
ã	LCAT	Lowercase a with tilde	0110 0011	099	63	143
ä	LCAD	Lowercase a with diaeresis	0110 0100	100	64	144
å	LCAR	Lowercase a with ring above	0110 0101	101	65	145
æ	LCAE	Lowercase diphthong ae	0110 0110	102	66	146
ç	LCCC	Lowercase c with cedilla	0110 0111	103	67	147
è	LCEG	Lowercase e with grave accent	0110 1000	104	68	150
é	LCEA	Lowercase e with acute accent	0110 1001	105	69	151
ê	LCEC	Lowercase e with circumflex	0110 1010	106	6A	152

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
ë	LCED	Lowercase e with diaeresis	0110 1011	107	6B	153
ì	LCIG	Lowercase i with grave accent	0110 1100	108	6C	154
í	LCIA	Lowercase i with acute accent	0110 1101	109	6D	155
î	LCIC	Lowercase i with circumflex	0110 1110	110	6E	156
ï	LCID	Lowercase i with diaeresis	0110 1111	111	6F	157
ð	LETH	Lowercase Icelandic letter Eth	0111 0000	112	70	160
ñ	LCNT	Lowercase n with tilde	0111 0001	113	71	161
ò	LCOG	Lowercase o with grave accent	0111 0010	114	72	162
ó	LCOA	Lowercase o with acute accent	0111 0011	115	73	163
ô	LCOC	Lowercase o with circumflex	0111 0100	116	74	164
õ	LCOT	Lowercase o with tilde	0111 0101	117	75	165
ö	LCOD	Lowercase o with diaeresis	0111 0110	118	76	166
÷	DIV	Division sign used in mathematics	0111 0111	119	77	167
ø	LCOO	Lowercase o with oblique line	0111 1000	120	78	170
ù	LCUG	Lowercase u with grave accent	0111 1001	121	79	171
ú	LCUA	Lowercase u with acute accent	0111 1010	122	7A	172
û	LCUC	Lowercase u with circumflex	0111 1011	123	7B	173
ü	LCUD	Lowercase u with diaeresis	0111 1100	124	7C	174
ý	LCYA	Lowercase y with acute accent	0111 1101	125	7D	175
þ	LTHN	Lowercase Icelandic letter Thorn	0111 1110	126	7E	176
ÿ	LCYD	Lowercase y with diaeresis	0111 1111	127	7F	177

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	1000 0000	128	80	200
^A	SOH/TC1	Start of heading	1000 0001	129	81	201
^B	STX/TC2	Start of text	1000 0010	130	82	202
^C	ETX/TC3	End of text	1000 0011	131	83	203
^D	EOT/TC4	End of transmission	1000 0100	132	84	204
^E	ENQ/TC5	Enquiry	1000 0101	133	85	205
^F	ACK/TC6	Acknowledge	1000 0110	134	86	206
^G	BEL	Bell	1000 0111	135	87	207
^H	BS/FE0	Backspace	1000 1000	136	88	210
^I	HT/FE1	Horizontal tab	1000 1001	137	89	211
^J	LF/NL/FE2	Line feed	1000 1010	138	8A	212
^K	VT/FE3	Vertical tab	1000 1011	139	8B	213
^L	FF/FE4	Form feed	1000 1100	140	8C	214
^M	CR/FE5	Carriage return	1000 1101	141	8D	215
^N	SO/LS1	Shift out	1000 1110	142	8E	216
^O	SI/LS0	Shift in	1000 1111	143	8F	217
^P	DLE/TC7	Data link escape	1001 0000	144	90	220
^Q	DC1/XON	Device control 1	1001 0001	145	91	221
^R	DC2	Device control 2	1001 0010	146	92	222
^S	DC3/XOFF	Device control 3	1001 0011	147	93	223
^T	DC4	Device control 4	1001 0100	148	94	224
^U	NAK/TC8	Negative acknowledge	1001 0101	149	95	225
^V	SYN/TC9	Synchronous idle	1001 0110	150	96	226
^W	ETB/TC10	End of transmission block	1001 0111	151	97	227
^X	CAN	Cancel	1001 1000	152	98	230
^Y	EM	End of medium	1001 1001	153	99	231
^Z	SUB	Substitute	1001 1010	154	9A	232
^[	ESC	Escape	1001 1011	155	9B	233
^\	FS/IS4	File separator	1001 1100	156	9C	234
^]	GS/IS3	Group separator	1001 1101	157	9D	235
^^	RS/IS2	Record separator	1001 1110	158	9E	236
^_	US/IS1	Unit separator	1001 1111	159	9F	237
	SP	Space	1010 0000	160	A0	240
!		Exclamation mark	1010 0001	161	A1	241
"		Quotation mark	1010 0010	162	A2	242
#	NUMB	Number sign	1010 0011	163	A3	243
$	DOLR	Dollar sign	1010 0100	164	A4	244
%		Percent sign	1010 0101	165	A5	245
&		Ampersand	1010 0110	166	A6	246

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	1010 0111	167	A7	247
(		Left parenthesis	1010 1000	168	A8	250
)		Right parenthesis	1010 1001	169	A9	251
*		Asterisk	1010 1010	170	AA	252
+		Plus sign	1010 1011	171	AB	253
,		Comma	1010 1100	172	AC	254
–		Minus sign	1010 1101	173	AD	255
.		Period	1010 1110	174	AE	256
/		Slash	1010 1111	175	AF	257
0		Zero	1011 0000	176	B0	260
1		One	1011 0001	177	B1	261
2		Two	1011 0010	178	B2	262
3		Three	1011 0011	179	B3	263
4		Four	1011 0100	180	B4	264
5		Five	1011 0101	181	B5	265
6		Six	1011 0110	182	B6	266
7		Seven	1011 0111	183	B7	267
8		Eight	1011 1000	184	B8	270
9		Nine	1011 1001	185	B9	271
:		Colon	1011 1010	186	BA	272
;		Semicolon	1011 1011	187	BB	273
<		Less than sign	1011 1100	188	BC	274
=		Equal sign	1011 1101	189	BD	275
>		Greater than sign	1011 1110	190	BE	276
?		Question mark	1011 1111	191	BF	277
@	AT	Commercial at sign	1100 0000	192	C0	300
A		Uppercase A	1100 0001	193	C1	301
B		Uppercase B	1100 0010	194	C2	302
C		Uppercase C	1100 0011	195	C3	303
D		Uppercase D	1100 0100	196	C4	304
E		Uppercase E	1100 0101	197	C5	305
F		Uppercase F	1100 0110	198	C6	306
G		Uppercase G	1100 0111	199	C7	307
H		Uppercase H	1100 1000	200	C8	310
I		Uppercase I	1100 1001	201	C9	311
J		Uppercase J	1100 1010	202	CA	312
K		Uppercase K	1100 1011	203	CB	313
L		Uppercase L	1100 1100	204	CC	314
M		Uppercase M	1100 1101	205	CD	315
N		Uppercase N	1100 1110	206	CE	316

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	1100 1111	207	CF	317
P		Uppercase P	1101 0000	208	D0	320
Q		Uppercase Q	1101 0001	209	D1	321
R		Uppercase R	1101 0010	210	D2	322
S		Uppercase S	1101 0011	211	D3	323
T		Uppercase T	1101 0100	212	D4	324
U		Uppercase U	1101 0101	213	D5	325
V		Uppercase V	1101 0110	214	D6	326
W		Uppercase W	1101 0111	215	D7	327
X		Uppercase X	1101 1000	216	D8	330
Y		Uppercase Y	1101 1001	217	D9	331
Z		Uppercase Z	1101 1010	218	DA	332
[	LBKT	Left bracket	1101 1011	219	DB	333
\	REVS	Reverse slash, backslash	1101 1100	220	DC	334
]	RBKT	Right bracket	1101 1101	221	DD	335
^	CFLX	Circumflex	1101 1110	222	DE	336
_		Underline, underscore	1101 1111	223	DF	337
`	GRAV	Left single quote, grave accent	1110 0000	224	E0	340
a		Lowercase a	1110 0001	225	E1	341
b		Lowercase b	1110 0010	226	E2	342
c		Lowercase c	1110 0011	227	E3	343
d		Lowercase d	1110 0100	228	E4	344
e		Lowercase e	1110 0101	229	E5	345
f		Lowercase f	1110 0110	230	E6	346
g		Lowercase g	1110 0111	231	E7	347
h		Lowercase h	1110 1000	232	E8	350
i		Lowercase i	1110 1001	233	E9	351
j		Lowercase j	1110 1010	234	EA	352
k		Lowercase k	1110 1011	235	EB	353
l		Lowercase l	1110 1100	236	EC	354
m		Lowercase m	1110 1101	237	ED	355
n		Lowercase n	1110 1110	238	EE	356
o		Lowercase o	1110 1111	239	EF	357
p		Lowercase p	1111 0000	240	F0	360
q		Lowercase q	1111 0001	241	F1	361
r		Lowercase r	1111 0010	242	F2	362
s		Lowercase s	1111 0011	243	F3	363
t		Lowercase t	1111 0100	244	F4	364

Table A-1 (Continued)
Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	1111 0101	245	F5	365
v		Lowercase v	1111 0110	246	F6	366
w		Lowercase w	1111 0111	247	F7	367
x		Lowercase x	1111 1000	248	F8	370
y		Lowercase y	1111 1001	249	F9	371
z		Lowercase z	1111 1010	250	FA	372
{	LBCE	Left brace	1111 1011	251	FB	373
|	VERT	Vertical line	1111 1100	252	FC	374
}	RBCE	Right brace	1111 1101	253	FD	375
~	TIL	Tilde	1111 1110	254	FE	376
	DEL	Delete	1111 1111	255	FF	377

B

F77 Programming Examples

SAMPLE PROGRAM #1

```
SOURCE FILE: <PUPS>DOGS>SAMPLE1.F77
COMPILED ON: 850212  AT: 14:01  BY: F77 REV. 19.4
Options selected:  SAMPLE -LISTING
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
    "-Full_OPTimize" means "-OPTimize 4", and default is "-OPTimize 2".
Options used(* follows those that are not default):
     64V Allow_PREconnection No_BIG Binary No_DClvar No_DeBuG No_DO1 DYnm
     No_ERRList ERRTty No_EXPlist No_FRN No_FTN_Entry INTL Listing* LOGL MAp
     No_OFFset OPTimize(2) No_OverFlow No_PBECB No_PRODuction No_RAnge
     SIlent(-1) TIME No_STATistics No_Store_Owner_Field UPcase No_XRef

    1         PROGRAM DEMO                          /*PROGRAM STATEMENT*/
    2   C
    3   ****************************************************************
    4   *                                                              *
    5   *  SAMPLE PROGRAM TO DEMONSTRATE THE VARIOUS FEATURES OF       *
    6   *  FORTRAN 77, AND A TYPICAL F77 COMPILER SOURCE LISTING.      *
    7   *                                                              *
    8   *                                                              *
    9   *                                                              *
   10   ****************************************************************
   11   C
   12   C
   13   C*****  PARAMETER STATEMENTS
   14   C
   15         INTEGER ONE,FOUR,TEN,FORTY    /* DCL TYPE BEFORE USE  */
```

```
      PARAMETER ONE = 1,
     *          FOUR = 4,
     *          TEN = 10,
     *          FORTY=TEN*FOUR    /* NOTE USE OF EXPRESSION */
C
C*****  THE CHARACTER DATA TYPE IS NEW TO FORTRAN 77.
C
      CHARACTER*4 FILE
      CHARACTER*12 FNAME, FORM*8
      CHARACTER*80 BUFFER              /*DEFINE INPUT BUFFER*/
      DIMENSION IN_ARRAY(80)          /* DEFINE INTEGER ARRAY*/
C
C*****  ARRAY DCL'S, USING LOWER BOUNDS AND 7 DIMENSIONS
C
      DIMENSION A(-5:5, 6, 0:9)
      DIMENSION B(1, 2, 3, 4, 5, 6, 7)
      CHARACTER C(0:FOUR, TEN)*5      /* CHAR ARRAYS ALLOWED */
                                     /*  NOTE USE OF PARAMETERS */
C
C*****  LOGICAL VARIABLES - NOTE *1, *2 AND *4 FORMS.
C*****  THESE ARE NOT FORTRAN 77, BUT ARE SUPPORTED FOR
C*****  COMPATIBILITY WITH IBM.  NOTE DATA INITIALIZATION
C*****  IN A TYPE STATEMENT.
C
      LOGICAL EXISTS, OPND
      LOGICAL*1 LOG1
      LOGICAL*2 LOG2/.TRUE./, LOG2B
      LOGICAL*4 LOGICALFOUR           /* UP TO 32 CHAR NAMES */
C
C*****  COMPLEX*16 IS NOT FORTRAN 77, BUT IS AN EXTENSION FOR
C*****  COMPATIBILITY WITH IBM FORTRAN.
C
      COMPLEX*16 DCOMPVAR
C
C*****  USE OF DOUBLE PRECISION TYPE DECLARATION.
C
      DOUBLE PRECISION D1, D2, D3, D4
C
C*****  EXTERNAL STATEMENT USED TO INSURE THAT AN EXTERNAL
C*****  FUNCTION WILL BE USED INSTEAD OF THE INTRINSIC.
C*****  IT COULD ALSO BE USED TO INSURE THAT ANY FUNCTION
C*****  USED WILL NOT BE MISINTERPRETED AS AN INTRINSIC EVEN
C*****  THOUGH SOME VENDOR MAY HAVE ADDED A FUNCTION OF THAT
C*****  NAME TO THE LIST OF INTRINSICS, ENHANCING PORTABILITY.
C
      EXTERNAL IFIX
C
C*****  BEGINNING OF EXECUTABLE CODE.  THE PURPOSE OF THIS
C*****  ROUTINE IS TO OPEN SOME FILES, AND THEN CHECK
C*****  THAT THE FILES WERE CORRECTLY OPENED.  THIS DEMON-
C*****  STRATES SOME OF THE NEW I/O FEATURES OF FORTRAN 77.
C
```

```
      FILE = 'FILE'      /* ASSIGN ASCII STRING TO CHAR VAR */
      SOME_NUMBER = 64.2
C
C*****  THIS IS THE MAIN LOOP
C
      DO 10 I=1,SQRT(SOME_NUMBER)*8   /* REAL EXPR FOR DO PARM
      FNAME = FILE//CHAR(I)           /* CHAR CONCATENATION */
C
C*****  NEW OPEN STATEMENT WITH KEYWORDS.
C
      OPEN (FILE = FNAME,
     *      UNIT = I,
     *      STATUS = 'UNKNOWN',
     *      ACCESS = 'SEQUENTIAL',
     *      ERR = 100)
C
C*****  NEW INQUIRE STATEMENT
C
      INQUIRE (UNIT = I,
     *         EXIST = EXISTS,
     *         OPENED = OPND,
     *         NAME = C(I+1,4),  /* EXPRESSION IN ARRAY REF */
     *         ERR = 101)
C
C*****  AN EXAMPLE OF A BLOCK IF-THEN-ELSE
C
      IF (EXISTS .AND. OPND) THEN
                WRITE (1,*) FNAME, ' EXISTS AND IS OPENED'
                                  /* LIST DIRECTED I/O WITH */
                                        /* CHAR CONSTANT */
           ELSE
                PRINT *, FNAME, ' NOT OPENED, NO ERROR RAISED'
                                     /* NEW PRINT STATEMENT */
           END IF
   10 CONTINUE
      GO TO 1000
C
C*****  END OF MAIN LOOP.  ERROR ROUTINES FOLLOW.
C
  100 WRITE (1, '(A, A, A, I3)')'ERROR ON OPEN OF ', FNAME,
     * 'ON UNIT ', I
                              /* FORMAT EMBEDDED IN I/O STMT */
      STOP 'ERROR'
  101 CONTINUE
      FORM = '    (A, I3)'                /* DEFINE FORMAT */
      WRITE (1, FORM) 'ERROR ON INQUIRE ON UNIT ',I
                            /* CHAR VAR REPRESENTS FORMAT */
```

```
115         STOP 'ERROR'
116   C
117    1000 INT_RANDOM = IFIX(3.1)       /* USE EXTERNAL FUNCTION */
118   C
119   C*****  THIS NEXT CALL DEMONSTRATES THE ALTERNATE RETURN.
120   C
121         CALL ALTRET (I, $5001, $5002)
122         INT_RANDOM = 0
123         GO TO 6000
124    5001 CONTINUE                              /* ALT RETURN #1 */
125         GO TO 6000
126    5002 CONTINUE                              /* ALT RETURN #2 */
127         INT_RANDOM = 2
128   C
129   C*****  ANOTHER EXAMPLE OF THE BLOCK-IF, BUT WITH MULTIPLE
130   C*****  BRANCHES.  ALSO, MULTIPLE ENTRY POINTS OF THE
131   C*****  SUBROUTINE MULTIN ARE USED.
132   C
133    6000 IF (INT_RANDOM .EQ. 0) THEN
134                   CALL MULTIN (I, INT_RANDOM)
135              ELSE IF (INT_RANDOM .EQ. 1) THEN
136                   CALL MULT1 (I)
137              ELSE IF (INT_RANDOM .EQ. 2) THEN
138                   CALL MULT2 (INT_RANDOM)
139              ELSE
140                   INT_RANDOM = 1
141              END IF
142   C
143   C*****  NEXT IS AN EXAMPLE OF INTERNAL FILES.  FIRST, READ AN
144   C*****  80 CHAR RECORD INTO BUFFER.  ASSUMING IT IS ALL
145   C*****  NUMBERS, IT CAN BE 'READ' INTERNALLY INTO ANOTHER
146   C*****  INTEGER VARIABLE.  INTERNAL FILES HAVE THE SAME
147   C*****  FUNCTIONALITY AS ENCODE/DECODE.
148   C
149          READ (5, '(A80)') BUFFER
150          READ (UNIT=BUFFER, FMT='(80I1)') IN_ARRAY
151   C
152   C*****  THIS IS AN EXAMPLE OF GENERIC TYPING OF INTRINSICS.
```

```
153   C*****  IT IS NO LONGER NECESSARY TO USE DIFFERENT FUNCTION
154   C*****  NAMES FOR THE SAME FUNCTION FOR DIFFERENT DATA TYPES.
155   C
156         D1 = 2.2                        /* DEFINE DOUBLE PREC VARS */
157         D2 = 3.6
158         D3 = 4.9
159         D4 = D1 + D2 + D3
160         SINGLE = 31.3134                              /* SINGLE PREC */
161         SINGLE=SQRT(D1)/ABS(D2)+SQRT(D3)*SQRT(D4)/SQRT(D_SINGLE)
162         END
```

EXTERNAL ENTRY POINTS

ENTRY POINT	PROGRAM UNIT	LINE	TYPE
DEMO		1	ENTRY

MAIN PROGRAM DEMO ON LINE 1

SYMBOLIC NAME	STORAGE CLASS	SIZE (DEC)	LOC (OCT)	ATTRIBUTES
10	CONSTANT			EXECUTABLE LABEL LINE 102
100	CONSTANT			EXECUTABLE LABEL LINE 107
1000	CONSTANT			EXECUTABLE LABEL LINE 117
101	CONSTANT			EXECUTABLE LABEL LINE 111
5001	CONSTANT			EXECUTABLE LABEL LINE 124
5002	CONSTANT			EXECUTABLE LABEL LINE 126
6000	CONSTANT			EXECUTABLE LABEL LINE 133
A	DYNAMIC	1320H	000054	REAL*4 DIMENSION(-5:5 6,0:9)
ABS	INTRINSIC			
ALTRET	CONSTANT			SUBROUTINE
B	DYNAMIC	10080H	002524	REAL*4 DIMENSION(1,2,3,4,5,6, 7)
BUFFER	DYNAMIC	80C	026264	CHARACTER*80
C	DYNAMIC	250C	026334	CHARACTER*5 DIMENSION(0:4,10)
CHAR	INTRINSIC			
D1	DYNAMIC	4H	026532	REAL*8
D2	DYNAMIC	4H	026536	REAL*8
D3	DYNAMIC	4H	026542	REAL*8
D4	DYNAMIC	4H	026546	REAL*8

```
DCOMPVAR        DYNAMIC     8H      026552      COMPLEX*16
D_SINGLE        DYNAMIC     2H      026562      REAL*4
EXISTS          DYNAMIC     2H      026564      LOGICAL*4
FILE            DYNAMIC     4C      026566      CHARACTER*4
FNAME           DYNAMIC     12C     026570      CHARACTER*12
FORM            DYNAMIC     8C      026576      CHARACTER*8
FORTY                       2H                  INTEGER*4 NAMED CONSTANT 40
FOUR                        2H                  INTEGER*4 NAMED
CONSTANT 4
I               DYNAMIC     2H      026602      INTEGER*4
IFIX            CONSTANT                        INTEGER*4 FUNCTION
INT_RANDOM      DYNAMIC     2H      026604      INTEGER*4
IN_ARRAY        DYNAMIC     160H    026606      INTEGER*4 DIMENSION(80)
LOG1            DYNAMIC     1C      027046      LOGICAL*1
LOG2            STATIC      1H      000030      INITIAL LOGICAL*2
LOG2B           DYNAMIC     1H      000053      LOGICAL*2
LOGICALFOUR     DYNAMIC     2H      027050      LOGICAL*4
MULT1           CONSTANT                        SUBROUTINE
MULT2           CONSTANT                        SUBROUTINE
MULTIN          CONSTANT                        SUBROUTINE
ONE                         2H                  INTEGER*4 NAMED CONSTANT 1
OPND            DYNAMIC     2H      027052      LOGICAL*4
SINGLE          DYNAMIC     2H      027054      REAL*4
SOME_NUMBER     DYNAMIC     2H      027056      REAL*4
SQRT            INTRINSIC
TEN                         2H                  INTEGER*4 NAMED
CONSTANT 10
```

```
C
  163   C
  164   C
  165   ****************************************************************
  166   *                                                              *
  167   *  THIS IS AN EXTERNAL FUNCTION OF THE SAME NAME AS THE        *
  168   *  INTRINSIC IFIX, AND DOES THE SAME THING, SO AS TO           *
  169   *  DEMONSTRATE THAT BY USING THE EXTERNAL STATEMENT ONE        *
  170   *  SUBSTITUTE ONE'S OWN VERSION OF A FUNCTION.                 *
  171   *                                                              *
  172   ****************************************************************
  173   C
  174   C
  175         INTEGER FUNCTION IFIX(RVAR)
  176         IFIX = RVAR
  177         RETURN
  178         END


EXTERNAL ENTRY POINTS

ENTRY POINT      PROGRAM UNIT        LINE       TYPE

IFIX                                 175        INTEGER*4 FUNCTION


 FUNCTION IFIX ON LINE 175

SYMBOLIC         STORAGE     SIZE   LOC        ATTRIBUTES
NAME             CLASS       (DEC)  (OCT)

RVAR             DUMMY ARG   2H     POS 1      REAL*4
```

```
C
  179   C
  180   C
  181   ****************************************************************
  182   *                                                              *
  183   *  THIS SUBROUTINE DEMONSTATES ALTERNATE RETURNS.              *
  184   *                                                              *
  185   ****************************************************************
  186   C
  187   C
  188         SUBROUTINE ALTRET (I, *, *)
  189         RETURN I                  /* IF I = 1, RETURNS TO 5001 */
  190                                   /* IF I = 2, RETURNS TO 5002 */
  191                                  /* OTHERWISE, RETURNS NORMALLY */
  192         END


EXTERNAL ENTRY POINTS

ENTRY POINT       PROGRAM UNIT        LINE        TYPE

ALTRET                                188         SUBROUTINE


SUBROUTINE ALTRET ON LINE 188

SYMBOLIC          STORAGE     SIZE    LOC         ATTRIBUTES
NAME              CLASS       (DEC)   (OCT)

I                 DUMMY ARG   2H      POS 1       INTEGER*4
```

```
C
  193   C
  194   C
  195   ****************************************************************
  196   *                                                              *
  197   *  THIS SUBROUTINE IS AN EXAMPLE OF A SUBROUTINE WITH          *
  198   *  MULTIPLE ENTRY POINTS.                                      *
  199   *                                                              *
  200   ****************************************************************
  201   C
  202   C
  203         SUBROUTINE MULTIN (I, INT_RANDOM)
  204   C
  205         I = 0
  206         INT_RANDOM = 13
  207         RETURN
  208   C
  209   C*****  SECONDARY ENTRY POINT.  NOTE THAT THE ARG LIST NEED
  210   C*****  NOT MATCH THAT AT THE HEADER STATEMENT.
  211   C
  212         ENTRY MULT1 (I)
  213         I = 15
  214         RETURN
  215   C
  216   C*****  NEXT ENTRY POINT
  217   C
  218         ENTRY MULT2 (INT_RANDOM)
  219         INT_RANDOM = INT_RANDOM**2
  220         RETURN
  221         END


EXTERNAL ENTRY POINTS

ENTRY POINT      PROGRAM UNIT       LINE       TYPE

MULT1            MULTIN             212        SUBROUTINE
MULT2            MULTIN             218        SUBROUTINE
MULTIN                              203        SUBROUTINE


SUBROUTINE MULTIN ON LINE 203

SYMBOLIC         STORAGE     SIZE   LOC        ATTRIBUTES
NAME             CLASS       (DEC)  (OCT)

I                DUMMY ARG   2H     POS 1      INTEGER*4
INT_RANDOM       DUMMY ARG   2H     -V-        INTEGER*4
```

SAMPLE PROGRAM #2

```
SOURCE FILE: <PUPS>DOGS>SAMPLE2.F77
COMPILED ON: 850212  AT: 14:01  BY: F77 REV. 19.4
Options selected:  SAMPLE2 -LISTING
Optimization note: Currently "-OPTimize" means "-OPTimize 2",
    "-Full_OPTimize" means "-OPTimize 4", and default is "-OPTimize 2".
Options used(* follows those that are not default):
     64V Allow_PREconnection No_BIG Binary No_DClvar No_DeBuG No_DOl DYnm
     No_ERRList ERRTty No_EXPlist No_FRN No_FTN_Entry INTL Listing* LOGL MAp
     No_OFFset OPTimize(2) No_OverFlow No_PBECB No_PRODuction No_RAnge
     SIlent(-1) TIME No_STATistics No_Store_Owner_Field UPcase No_XRef
```

```
    1   C
    2   ****************************************************************
    3   *                                                              *
    4   *                                                              *
    5   *  THIS SAMPLE PROGRAM DEMONSTRATES THE USE OF QUAD FLOATING *
    6   *  POINT ARITHMETIC, A NEW PRIME FEATURE. PROGRAMS THAT USE  *
    7   *  REAL*16 DATA TYPE CAN ONLY BE EXECUTED ON PRIME MACHINES  *
    8   *  THAT SUPPORT QUAD PRECISION (REV. 19.2 AND HIGHER).       *
    9   *                                                              *
   10   *  THE SUBPROGRAM EXAMINES 3 REAL VALUES PASSED FROM THE     *
   11   *  MAIN PROGRAM AND RETURNS THAT VALUE WHICH HAS THE LARGEST *
   12   *  ABSOLUTE VALUE.                                           *
   13   *                                                              *
   14   ****************************************************************
   15   C
   16   C*****  USE OF QUAD PRECISION TYPE DECLARATION
   17   C
   18         REAL*16 Q1, Q2, Q3
   19   C
   20         Q1 = 4.2Q1                    /* DEFINE REAL*16 VARS   */
   21         Q2 = 2.33Q2
   22         Q3 = 6.9Q0
   23   C
   24         READ*, Q1, Q2, Q3
   25   C
   26         CALL BIG(Q1, Q2, Q3, ANS)      /* CALL SUBROUTINE BIG */
   27         PRINT*, ANS
   28         END
```

```
EXTERNAL ENTRY POINTS

ENTRY POINT      PROGRAM UNIT        LINE       TYPE

.MAIN.                               18         ENTRY


MAIN PROGRAM .MAIN. ON LINE 18

SYMBOLIC         STORAGE     SIZE    LOC        ATTRIBUTES
NAME             CLASS       (DEC)   (OCT)

ANS              DYNAMIC     2H      000054     REAL*4
BIG              CONSTANT                       SUBROUTINE
Q1               DYNAMIC     8H      000056     REAL*16
Q2               DYNAMIC     8H      000066     REAL*16
Q3               DYNAMIC     8H      000076     REAL*16
C
   29   C
   30   ****************************************************************
   31   *                                                              *
   32   *  THE SUBROUTINE BIG USES THE INTRINSIC FUNCTION QABS TO      *
   33   *  FIND WHICH VALUE HAS THE LARGEST MAGNITUDE AND RETURNS      *
   34   *  THE RESULT TO THE MAIN PROGRAM.                             *
   35   *                                                              *
   36   ****************************************************************
   37   C
   38         SUBROUTINE BIG(X, Y, Z, BIQABS)
   39         REAL*16 X, Y, Z, BIGABS
   40         IF (QABS(X).GT.QABS(Y)) THEN
   41             IF (QABS(X).GT.(Z)) THEN
   42                 BIQABS = X
   43             ELSE
   44                 BIQABS = Z
   45             END IF
   46         ELSE
   47             IF (QABS(Y).GT.QABS(Z)) THEN
   48                 BIQABS = Y
   49             ELSE
   50                 BIQABS = Z
   51             END IF
   52         END IF
   53         RETURN
   54         END
```

```
EXTERNAL ENTRY POINTS

ENTRY POINT      PROGRAM UNIT        LINE        TYPE

BIG                                  38          SUBROUTINE


SUBROUTINE BIG ON LINE 38

SYMBOLIC         STORAGE     SIZE    LOC         ATTRIBUTES
NAME             CLASS       (DEC)   (OCT)

BIGABS           DYNAMIC     8H      000070      REAL*16
BIQABS           DUMMY ARG   2H      POS 4       REAL*4
QABS             INTRINSIC
X                DUMMY ARG   8H      POS 1       REAL*16
Y                DUMMY ARG   8H      POS 2       REAL*16
Z                DUMMY ARG   8H      POS 3       REAL*16
```

C
Converting FTN Programs to F77

The techniques required for converting FTN programs to F77 are described in this appendix.

The simplicity of converting FTN programs to F77 results from two factors:

- The designers of FTN used preliminary documents released by ANSI during the development of FORTRAN 77. The information in these documents was used to make FTN's extensions to FORTRAN 66 identical to those of the future FORTRAN 77 wherever this could be accomplished without violating the FORTRAN 66 standard.

- F77 includes all FTN constructs, except the obsolete TRACE statement, that are absent from but compatible with FORTRAN 77.

The result is that many FTN program units can be compiled in F77 with no changes. Most of the other programs can be converted with only minor changes.

The program unit which cannot easily be converted to F77 can usually be left in FTN form and called by other units that are written in F77. See USING AN FTN PROGRAM UNIT IN AN F77 PROGRAM.

PROGRAM CONVERSION

Any project converting FTN programs to F77 should have available:

- This guide
- The Prime User's Guide
- The FORTRAN Reference Guide
- The ANSI Standard for FORTRAN 77

Conversion of a program to F77 need not be an all or none process. Due to the similarity of FTN and F77, each unit of an FTN program can be dealt with separately when the program as a whole is converted.

The first step in converting an FTN program unit to F77 is to compile it in F77 and see what, if any, error messages result. Due to the detailed and prescriptive information given by an F77 error message, the messages produced should give an assessement of the changes needed.

The second step is to check the FTN program unit for constructs that are common to and syntactically the same in FTN and F77, and therefore generate no syntax errors, but which have different requirements or results in the two languages due to differences between the ANSI standards. Such constructs are called "optionally acceptable FTN constructs" and "reimplemented FTN constructs." These terms are defined, and all such constructs are described, under PRODUCING AN F77-COMPATIBLE PROGRAM UNIT.

The first and/or second steps should be iterated until the program unit compiles correctly with all optionally acceptable and reimplemented constructs dealt with as necessary.

The third step is a thorough check of the converted program unit. Before it is accepted as correct, it should pass the same tests it was required to pass before being accepted in its original version.

> **Caution**
>
> The fact that a program unit compiles without error in F77 does not mean it will produce the same results in F77 that it did in FTN. Identical results can be achieved only if all optionally acceptable and reimplemented constructs have been correctly dealt with.

DEGREES OF PROGRAM UNIT CONVERSION

Conversion of a program unit to F77 is not an all or none matter. Three degrees of conversion of an FTN program unit can be distinguished:

- The unit may be left in FTN, but may reference and be called by other units that are in F77. This conversion is contextual. The unit per se remains an FTN program unit.

- The unit may be recompiled in F77, but retain certain optionally acceptable FTN constructs that violate the FORTRAN 77 standard. The F77 compiler will compile them correctly only if it is invoked with appropriate options, as described below. A program unit of this type is termed an F77-compatible program unit.

- The unit may be completely converted to standard-conforming F77. It is then termed an F77-standard program unit.

There is no need for all units of a converted program to be converted to the same degree.

USING AN FTN PROGRAM UNIT IN AN F77 PROGRAM

An FTN program unit may reference and be referenced by an F77 program unit. See the comments in Chapter 1 under INTERFACE TO OTHER LANGUAGES. The following additional restrictions apply.

- An F77 program unit cannot pass a subprogram as an argument to an FTN program unit, nor can an FTN unit pass a subprogram to an F77 unit.

- An F77 function returning a COMPLEX*8 value cannot be referenced by an FTN program unit, nor can an FTN function returning a COMPLEX*8 value be referenced by an F77 program unit.

- Data of types that exist in F77 but not in FTN cannot be passed as arguments.

- An F77 subroutine cannot use the F77 alternate return mechanism (that is, RETURN (expression)) if it will be called by an FTN program unit. The F77 subroutine must use the alternate mechanism (that is, GO TO (dummy variable)).

- F77 cannot pass unaligned arguments to FTN program units.

Any program unit for which no modifications to use the added power of F77 are contemplated, and which can be invoked by an F77 program unit, can be left in FTN indefinitely. No F77 program unit can reference or be referenced by any program unit that was compiled in R-mode. An FTN unit in R-mode must be recompiled into V-mode before it can become part of an F77 program. A few rarely used R-mode FTN constructs are not available in V-mode. See Unsupported FTN Constructs.

PRODUCING AN F77-COMPATIBLE PROGRAM UNIT

The information needed to convert an FTN program unit to an F77-compatible program unit falls into four categories:

- Constructs that are compiled differently by the FTN and F77 compilers, but which will be compiled in the FTN manner by the F77 compiler if the compiler is invoked with appropriate options (optionally acceptable FTN constructs).
- Constructs that are compiled differently by the FTN and F77 compilers, and which cannot be compiled in the FTN manner by the F77 compiler (reimplemented FTN constructs).
- Constructs that exist in FTN but not in F77 (unsupported FTN constructs).
- Constructs that exist in FTN and are not part of FORTRAN 77, but have been added to F77 for compatibility (obsolete FTN constructs).

Optionally Acceptable FTN Constructs

The various compiler options mentioned below are fully defined in Chapter 9.

The optionally acceptable FTN constructs, and their F77 versions, are as follows. In each case, the F77 version conforms to the FORTRAN 77 standard, while the FTN version does not.

FTN DO Loops: An FTN DO loop always executes once, and permits extended DO ranges. An F77 DO loop can execute zero times and forbids extended DO ranges. This difference can be insidious because all FTN DO loops are syntactically correct in F77. There are also other differences, but these do not affect program unit conversion. The two types of loop are fully compared under the DO Statement in Chapter 6.

To cause the F77 compiler to produce FTN-type DO loops, invoke it with the -DO1 option.

Short Integers: In FTN, the type INTEGER without a *(length) specification is synonymous with INTEGER*2 (short integer), and integer constants are stored as INTEGER*2 unless they are too big or contain too many digits. (See Chapter 2.) In F77, INTEGER is synonymous with INTEGER*4 (long integer) and integer constants are stored as INTEGER*4.

To cause the F77 compiler to produce short integers in the manner of the FTN compiler, invoke it with the -INTS option.

The FTN compiler has the -INTL option, which causes it to treat integer data in the manner described for F77. A program unit that was normally compiled with -INTL in FTN requires no special action regarding integer data when converted to F77.

Short Logical Data: In FTN, logical data always occupies two bytes (LOGICAL*2); there is no LOGICAL*4 type. In F77, the type LOGICAL without a *(length) specification is synonymous with LOGICAL*4, and logical constants are stored as LOGICAL*4.

To cause the F77 compiler to produce short logical data (except where LOGICAL*4 has been explicitly specified) invoke it with the -LOGS option.

Static Storage Default: Both the FTN and F77 compilers offer the -DYNM/-SAVE option. In FTN, the default is -SAVE, so that all data is static. In F77, the default is -DYNM, so that all data is dynamic unless explicitly declared static. This dynamic storage property is required by the FORTRAN 77 standard.

If the correct operation of an FTN program unit is dependent on some or all of its data being static by default, the -SAVE option must be given explicitly when it is compiled in F77.

A program unit that was normally compiled with -DYNM in FTN requires no special action regarding storage class when converted to F77.

Reimplemented FTN Constructs

Most of the effort required in converting an FTN program unit to F77 will concern reimplemented constructs. Each instance of such a construct must be examined, and modified if necessary, to be sure it will produce the results desired when run under F77.

The reimplemented FTN constructs and their F77 versions are as follows. In each case where standard conformance is involved the F77 version conforms to the FORTRAN 77 standard, while the FTN version conforms to the FORTRAN 66 standard.

Note

> Most reimplemented constructs are syntactically identical in FTN and F77. No error messages will result when such constructs are encountered. They must be found by inspecting the source code.

Listing Control: In FTN, the interaction between the compiler options that create the source listing and the program statements that turn

source listing generation on and off is somewhat different than in F77. The two charts below illustrate the difference. Note that in F77, FULL LIST is an obsolete synonym for LIST.

FTN	-LIST NO	-LIST YES	-EXPLIST
NO LIST	NO LISTING	NO LISTING	FULL LISTING
LIST	NO LISTING	NORMAL LISTING	FULL LISTING
FULL LIST	NO LISTING	FULL LISTING	FULL LISTING

F77	-NO_LISTING	-LISTING	-EXPLIST
NO LIST	NO LISTING	NO LISTING	NO LISTING
LIST	NO LISTING	NORMAL LISTING	FULL LISTING
FULL LIST	NO LISTING	NORMAL LISTING	FULL LISTING

Global Mode: FTN assigns the global mode to those names that are not explicitly typed and whose first appearance in the program follows the global mode statement. F77 assigns the global mode to all names that are not explicitly typed, whether or not they follow the global mode statement.

Intrinsic Functions: FTN treats IFIX, FLOAT, and IDINT as generic functions, not restricting their argument to a particular type. F77 provides the INT and REAL generic functions, but treats IFIX, FLOAT, and IDINT as specific functions requiring a particular type.

FTN allows LOGICAL*2 arguments in the following intrinsics: LS, RS, SHFT, LT, RT, AND, OR, NOT, and XOR. F77 allows only INTEGER*2 and INTEGER*4 arguments.

FORTRAN 77 introduces a number of new intrinsic functions. Their names may conflict with those of user-supplied subprograms. To cause such a duplicate name to refer to the user-supplied subprogram, specify it in an EXTERNAL statement. The similarly-named intrinsic will then be unavailable to that program unit.

Intrinsics in Constant Expressions: FTN allows a subset of the intrinsic functions in constant expressions. F77 does not allow this practice.

Input/Output: In FTN, an unformatted sequential file must consist of fixed length records. In F77, such a file may consist of either fixed or variable length records.

In FTN, BACKSPACE works only on tape files. In F77, it will work on all formatted sequential files and on fixed length unformatted sequential files.

In FTN, a READ or WRITE can access more than one record. In F77, a READ or WRITE always accesses a single record (slash editing excepted).

The method for increasing maximum record length has been greatly simplified in F77. Use of ATTDEV is no longer required. The F77 method is described under INCREASING MAXIMUM RECORD LENGTH in Chapter 6.

Extra Parentheses in I/O Statements: FTN ignores extra parentheses in I/O lists, while F77 considers them syntax errors. Prohibiting the extra parentheses prevents certain ambiguities that could otherwise arise in an I/O list.

Blanks in Format Lists: FTN allows blanks as well as commas to separate format list descriptors. F77 ignores blanks in format lists unless they are in a character or Hollerith constant.

Slash Edit-Control Descriptor: In FTN, execution of the statement:

```
      WRITE (N,100)
  100 FORMAT (/)
```

will cause one blank record to be written. In F77, two blank records will be written.

STOP and PAUSE Statements: In FTN, the number (if any) printed by a STOP or PAUSE statement will be in octal form. F77 prints such a number in decimal.

The FTN STOP statement has no effect on I/O units. The F77 STOP statement closes any I/O units used by the program.

Unsupported FTN Constructs

The only frequently used FTN construct not supported in F77 is the TRACE statement, which was used in conjunction with the -TRACE compiler option (also unsupported) as a debugging tool.

When assistance in debugging an F77 program is required, use the far more powerful Source Level Debugger, available from Prime as a separately priced item. For complete information on using the debugger, see the Source Level Debugger User's Guide.

Certain specialized FTN constructs are dependent on FTN compiler options that are not supported by the F77 compiler. When one of these constructs has been used in an FTN program unit being converted to F77, it must be replaced with an equivalent F77 construct, or eliminated entirely. The options are:

The -32R and -64R options: A few FTN constructs are available only in R-mode: the commonly used ones are multi-level alternate returns, and variable-length argument lists. Methods that provide the same results and work in V- and I-mode can always be found.

The -SPO Option: The FTN constructs dependent on the -SPO option are not enumerated here, as they are of interest only to certain specialized users who need no additional information. If there is no alternative to using an -SPO construct, be sure that the program unit is otherwise callable from F77, and keep it in FTN form.

Obsolete FTN Constructs

The following features of FTN are not standard in FORTRAN 77. F77 has been extended to accept them, but they are considered obsolete techniques. Do not use them in new programs.

The obsolete techniques will always produce the same results in F77 as in FTN. They are mentioned here so that those converting FTN programs to F77 will know that, despite their nonstandard status, they can be ignored during the conversion process. They are not explained here, because they are properly part of FTN, not F77. For information on them, see the FORTRAN Reference Guide.

The obsolete features are:

- The format nOddd... for octal constants
- The ENCODE and DECODE statements for in-storage type conversion
- Hollerith strings
- Indexing a multi-dimensional array with a one-subscript reference in an EQUIVALENCE statement
- Alternate returns using a GO TO to a statement-label dummy variable
- Use of "$" instead of "*" to denote a statement label constant

- Extended DO ranges, except when the F77 compiler is invoked with the -DO1 option for generation of FTN type DO loops. If an extended DO range is present in a program compiled with -NODO1 (the default) no error will be detected, but unpredictable results will occur. See Chapter 9 for more on the -DO1/-NO_DO1 option.

PRODUCING AN F77 STANDARD PROGRAM UNIT

An F77 standard unit is a converted FTN unit that contains no optionally acceptable constructs. Such a unit must compile without errors and give the expected results when compiled with the default options -NO_DO1, -INIL, -LOGL, and -DYNM.

With respect to reimplemented, unsupported, and obsolete FTN constructs, the task of producing an F77 standard program unit is identical to that of producing an F77 compatible program unit.

Elimination of Optionally Acceptable Constructs

To eliminate an FTN program unit's dependence on the similarity of INTEGER with INTEGER*2 and LOGICAL with LOGICAL*2, the following steps can be taken. Where INTEGER*2 or LOGICAL*2 data is specifically desired, modify or create the appropriate type-statement. Where INTEGER*4 and LOGICAL*4 will do, be sure that use of the longer data types will not cause mismatch of arguments in subprogram invocations, or unexpected results in mixed-type expressions and assignments.

Elimination of dependence on FTN type handling of DO loops is accomplished as follows:

1. Eliminate any extended DO ranges. The simplest way is to substitute an appropriate subprogram invocation.

2. Where the program unit's logic is unalterably dependent on the one trip property of the FTN DO loop (which is only rarely the case) insert appropriate conditional statements into the source code to insure that the trip will occur.

Existing conditional statements serving only to prevent the compulsory one trip if the DO test is already satisfied when control reaches the loop can be left in or deleted as desired. They merely duplicate the normal action of an F77 DO loop.

Elimination of a program unit's dependence on the -SAVE option is accomplished by naming all data items that must be static in a SAVE statement in the program unit. See the SAVE Statement in Chapter 3.

D

Memory Formats for F77

Prime computers use a 16-bit memory halfword. All FORTRAN 77 data types except CHARACTER occupy either 32 bits or some multiple of 32 bits. CHARACTER data occupies one byte per character.

F77 includes the INTEGER*2, LOGICAL*2, and LOGICAL*1 types for compatibility with FTN; these occupy 16, 16, and 8 bits respectively. These types should never be used in new programs.

Figure D-1 summarizes the sizes and internal bit-usages of the F77 data types. Detailed descriptions of each type are presented below.

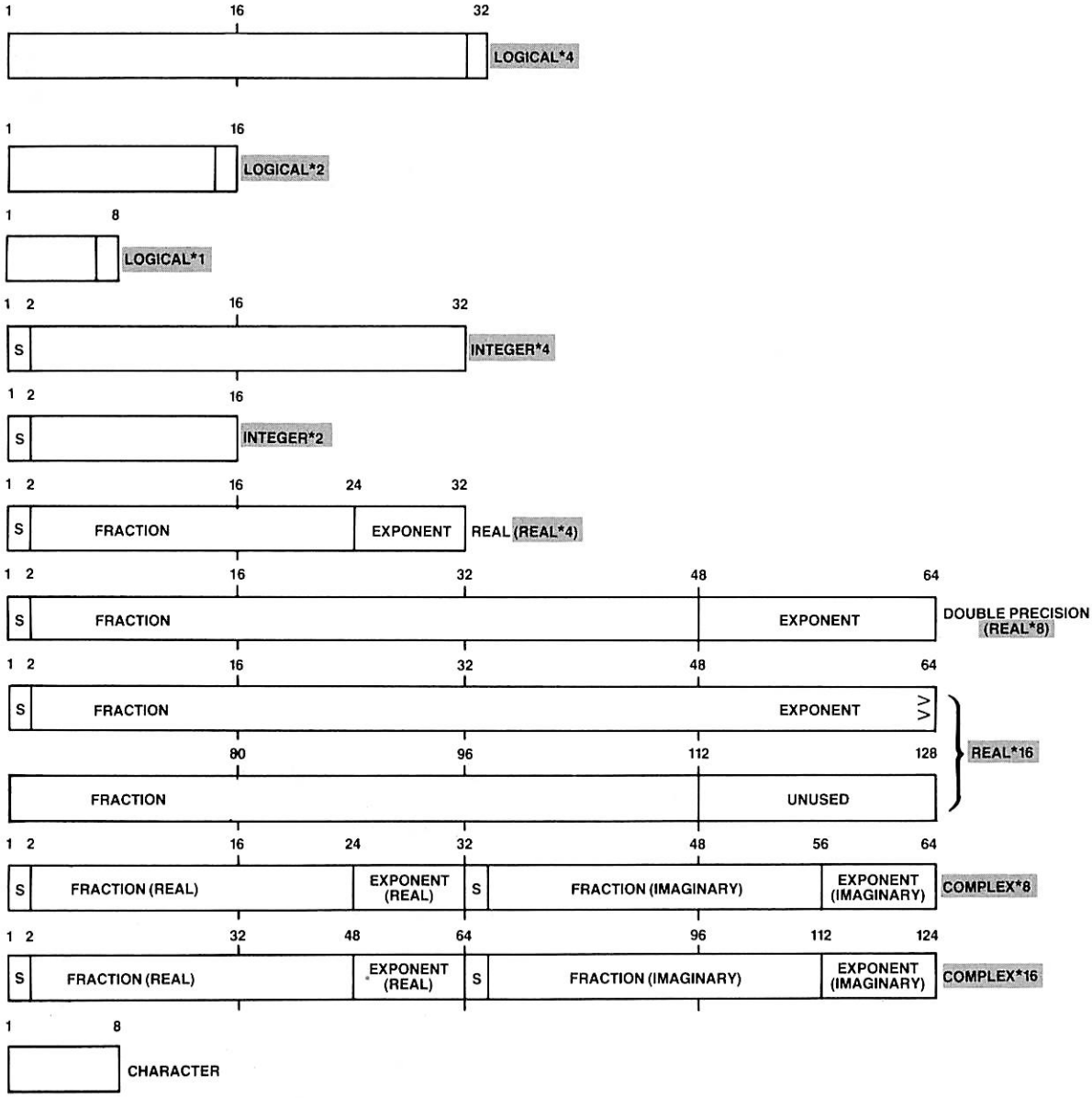
```
DATA TYPES

    LOGICAL*4  32 bits.  Bits 1-31=0   Bit 32: 0=.FALSE.
                                                1=.TRUE.

    LOGICAL*2  16 bits.  Bits 1-15=0   Bit 16: 0=.FALSE.
                                                1=.TRUE.

    LOGICAL*1   8 bits.  Bits 1-7=0    Bit  8: 0=.FALSE.
                                                1=.TRUE.


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.
```



Internal Representations of Prime F77 Data Types
Figure D-1

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 (halfword 1, halfword 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 = fraction. Bits 25-32 = exponent. The fraction 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:

$$-1 \leq M < -1/2 \text{ or } 1/2 \leq M < 1$$
$$0 \leq E \leq 255$$

Zero is represented as M = 0, E = 0.

The value range, in octal (halfword1, halfword2) is:

('100000, '000377) [SeeNote] 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 = fraction. Bits 49-64 = exponent. The fraction 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 ≤ M <-1/2 or 1/2 ≤ M < 1
-32768 ≤ E ≤ 32767.

Zero is represented as M = 0, E = 0

The value range, in octal (halfword1, halfword2, halfword3, halfword4) 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.

REAL*16 112 bits. Bit 1 = sign bit. Bits 2-48 = fraction. Bits 49-64 = exponent. Bits 65-112 are additional fraction bits. Bits 113-128 are unused (set to 0). The fraction and sign are treated as a 2's complement number and the exponent is a signed, excess 128 exponent. A REAL*16 floating point number is represented as:

$$N = M * 2^{(E-128)}$$

where:

$$-1 \leq M < -1/2 \text{ or } 1/2 \leq M < 1$$
$$-32768 \leq E \leq 32767$$

Zero is represented as M = 0, E = 0

The value range, in octal (halfword1, halfword2,...halfword8) is:

('100000,'000000,'000000,'000000,'077777,'000000,'000000,'000000) to
('077777,'177777,'177777,'077777,'177777,'177777,'177777,'000000)

corresponding to -1*2**32639 and (1-e)*2 32639 [See Note]

The values closest to zero, in octal, are:

('137777,'177777,'177777,'100000,'177777,'177777,'177777,'000000) and
('040000,'000000,'000000,'100000,'000000,'000000,'000000,'000000)

corresponding to (-1/2+e)*2**-32896 and 1/2*2**-32896 [See Note]

Normalization ensures that bits 1 and 2 are different and is achieved by shifting left 1 bit at a time. The effective precision is between 112 and 113 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.

COMPLEX*16 128 bits. Same as COMPLEX, except that two DOUBLE PRECISION entities are used.

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.)

Each character occupies one byte. The length of a CHARACTER item may be up to 32767 characters.

E
SHORTCALL Examples

This appendix contains examples of the SHORTCALL statement for both V mode and I mode.

V-MODE EXAMPLES

This section contains an example of a FORTRAN 77 main program and three Prime Macro Assembler (PMA) routines that are called from the FORTRAN 77 program using the Prime SHORTCALL Interface. The F77 program and the PMA routines are all V-mode programs. For information about writing the PMA programs and how the interface works, see the Assembly Language Programmer's Guide. The SHORTCALL statement for F77 is discussed in Chapter 3 of this manual.

The V-Mode Programs

The F77 main program, TEST_SHORTCALL.F77 is as follows:

```
C Test Program -- Requires FOO.PMA, BAR.PMA, and NUM.PMA
      PROGRAM TEST_SHORTCALL

       PARAMETER (IPAR = 2)
       INTEGER I, J, K, FOO,NUM
C Three PMA routines are SHORTCALLED: FOO (one argument),
C BAR (no argument), and NUM (two arguments)
```

```
C
        SHORTCALL FOO(IPAR*4), BAR, NUM(2)
C
C Initialize I, J, and K
        I = 1
        J = 130
        K = 3
        I = FOO(J)
        WRITE (1, 10) I
10      FORMAT ('The value of I should be 13.  I =' I4)
C       PRINT *, 'The value of I should = 13, it is    ',I

        CALL BAR
        I = NUM(J,K)
        WRITE (1, 20) J, K, I
20      FORMAT ('J =', I4, '  K =', I4, '  I = J + K;  I =', I4)
C       PRINT *, 'J = ',J,'   K = ',K,'  I = J+K, I = ',I
        STOP
        END
```

TEST_SHORTCALL.F77 calls the PMA routines FOO, BAR, and NUM. FOO and NUM are functions. FOO.PMA, BAR.PMA, and NUM.PMA are as follows:

```
* FOO.PMA
* Initialize V-mode program
*
        SEG
        RLIT
* Define FOO's entry point name
        SUBR FOO
        DYNM ARG1(2)
*
* Save address of first (and only) argument
FOO     STL ARG1
* Load argument into L register
        LDL ARG1,*
* Divide argument by 10
        DIV =10
* Return to calling program
        JMP XB%
        END
```

```
* BAR.PMA
* Initialize V-mode program
        SEG IMPURE
        RLIT
* Define BAR's entry point name
        ENT BAR
        EXT TNOU
*
        DYNM TEMP(2)

* Save return address in TEMP
BAR     EAL XB%
        STL TEMP
* Use PCL to print out a message
        PCL PTNOU,*
           AP =C' BAR subroutine was SHORTCALLed!!',S
           AP =32,SL
* Restore return address
        EAXB TEMP,*
* Return to calling program
        JMP XB%
*
PTNOU   IP TNOU
*
        END


* NUM.PMA
* Initialize V-mode program
        SEG
* Define NUM's entry point name
        SUBR NUM
        DYNM ARG1(2), ARG2(2)
*
* Save address of first argument in ARG1
NUM   STL ARG1
* Save return address in ARG2
        EAL XB%
        STL ARG2
* Load pointer to first argument in XB
        EAXB ARG1,*
* Load first argument into L register
        LDL XB
* Add second argument to first argument
        ADL XB
* Restore return address
        EAXB ARG2,*
* Return to calling program
        JMP XB%
        END
```

Compiling, Linking, and Executing the V-Mode Programs

To execute the programs, compile TEST_SHORTCALL and assemble FOO, BAR, and NUM. Link the four programs, creating TEST_SHORTCALL.RUN. If you invoke the programs with

```
r test_shortcall
```

the result is

```
The value of I should be 13.  I =  13
 BAR subroutine was SHORTCALLed!
J = 130  K =   3  I = J + K;  I = 133
**** STOP
```

I-MODE EXAMPLES

The I-mode FORTRAN 77 program CALLQF.F77 calls two PMA functions, SQUARE and CUBE, using the Prime SHORTCALL Interface.

The I-Mode Programs

CALLQF.F77 calls two PMA functions; CALLQF.F77 must be compiled with the -32I option.

```
C Test Program for I mode SHORTCALL
C Requires SQCUBE.PMA
        PROGRAM CALLQF

C The PMA program SQCUBE with two entry points
C SQUARE and CUBE is SHORTCALLED; both SQUARE
C and CUBE are functions.
C
        INTEGER X, Y, SQUARE, CUBE
        SHORTCALL SQUARE, CUBE
C
        X = SQUARE(13)
        WRITE (1, 10) X
10      FORMAT ('The value of X should be 169.  X =' I4)
        Y = CUBE(10)
        WRITE (1, 20) Y
20      FORMAT ('The value of Y should be 1000.  Y =' I5)
        STOP
        END
```

The I-mode program SQCUBE.PMA contains the two functions SQUARE and CUBE that are SHORTCALLed by CALLQF.F77.

```
* SQCUBE.PMA
*
          SEGR
          ENT     SQUARE        declare entry to SQUARE
          ENT     CUBE          declare entry to CUBE
CUBE      L       6,R1
          M       6,R1
          PIM     6             position for next multiply
          JMP     SQ1           jump to next multiply
SQUARE    L       6,R1
SQ1       M       6,R1
          PIM     6             position for return
          L       2,6           move into R2 (where F77 expects it)
          JMP     R0
          END
```

Compiling, Linking, and Executing the I-Mode Programs

To execute the programs, compile CALLQF with the -32I option and assemble SQCUBE. Link the two programs, creating CALLQF.RUN. If you invoke the programs with

```
r callqf
```

the result is

```
The value of X should be 169.  X = 169
The value of Y should be 1000.  Y = 1000
**** STOP
```

G

The Search Rules Facility

INCLUDE FILES AND THE SEARCH RULES FACILITY

As of Revision 21.0, the PRIMOS search rules facility enables you to establish an INCLUDE$ search list. An INCLUDE$ search list is a list of directories to be searched whenever an INCLUDE statment or $INSERT directive is processed by the compiler. Although there are several kinds of search lists, this appendix explains only the INCLUDE$ search list. For complete information about PRIMOS search rules, see the Advanced Programmer's Guide, Volume II.

When you specify a file in an INCLUDE statement or $INSERT directive, you must ordinarily give as much of the file pathname as PRIMOS needs to locate the file. If you often use the INCLUDE statements or $INSERT directives to refer to files, and if the files are kept in a number of different directories, keeping track of the file pathnames can be difficult. Now, however, you can locate a file by supplying only a filename and using the search rules facility to provide the full pathname.

Establishing Search Rules

To establish the search rules, perform the following steps:

1. Create a template file called

   [yourchoice.]INCLUDE$.SR

This file should contain a list of the pathnames of the directories that contain the files you often refer to when using INCLUDE and $INSERT statements. List the directories in the order in which you want them to be searched. For example, you might create a file called MY.INCLUDE$.SR that contains the following directory names:

```
<SYS1>MASTER_DIR>INSERT_FILES
<SYS2>ME
```

2. Activate the template file with the SET_SEARCH_RULES (SSR) command. For example, if your file is named MY.INCLUDE$.SR, type

```
OK, SSR MY.INCLUDE$
```

This command sets your INCLUDE$ search list. This search list may contain system search rules and administrator search rules in addition to the rules you specified in MY.INCLUDE$.SR.

When you give the SSR command shown in step 2, PRIMOS copies the contents of MY.INCLUDE$.SR into your INCLUDE$ search list. If you have no special system or administrator search rules, your INCLUDE$ search list appears as follows when you give the LIST_SEARCH_RULES (LSR) command:

```
List: INCLUDE$
Pathname of template: <MYSYS>ME>F77>MY.INCLUDE$.SR

[home_dir]
<SYS1>MASTER_DIR>INSERT_FILES
<SYS2>ME
```

[home_dir], your current attach point, is the system default. It is always the first directory searched, unless you remove it from the list or change the order of evaluation by using the NO_SYSTEM option of the SSR command. Additional search rules, established as system-wide defaults by your system administrator, may also appear at the beginning of your INCLUDE$ search list. The above search rules would initiate the search in [home_dir], then search <SYS1>MASTER_DIR>INSERT_FILES, and lastly <SYS2>ME.

The SET_SEARCH_RULES and LIST_SEARCH_RULES commands are described in the PRIMOS Commands Reference Guide. For more information about establishing search rules, see the Advanced Programmer's Guide, Volume II.

Using Search Rules

Once you have set the search list, any INCLUDE or $INSERT statement in a program can give just the filename rather than the full pathname of the file. PRIMOS then searches the contents of the directories in the INCLUDE$ search list for the filename specified in the INCLUDE or $INSERT statement. If PRIMOS finds the file, it stops searching and returns the full pathname of the file to the compiler. The compiler then uses this pathname to locate the file and inserts its contents into the source program.

Using [referencing_dir]

The Advanced Programmer's Guide describes several expressions you can use in your list of search rules. One of these, [referencing_dir], has a special meaning for INCLUDE$ search lists. Like [home_dir], [referencing_dir] is a variable that PRIMOS replaces with a directory pathname. [referencing_dir] always evaluates to the pathname of the directory from which the request for an INCLUDE or $INSERT file is made. Thus, if an INCLUDE or $INSERT statement is located in a program, [referencing_dir] evaluates to the pathname of the directory that contains the program.

[referencing_dir] may be useful if all of the following three circumstances hold:

- You are compiling a program that is not in your current directory.
- The directory containing the program is not in your search rules list.
- Your program contains one or more INCLUDE or $INSERT statements.

Under the above circumstances, the search for the INCLUDE or $INSERT file succeeds only if [referencing dir] is in your list of search rules.

You can also use [referencing_dir] for programs that contain nested INCLUDE or $INSERT statements. INCLUDE or $INSERT statements are nested if the file specified by an INCLUDE or $INSERT statement also contains an INCLUDE or $INSERT statement. If nested INCLUDE or $INSERT statements specify files that are located in the same directory as the file in which they are nested, putting [referencing_dir] at the top of your list of search rules could speed up the search somewhat.

H

Alphabetic Summary of F77 Intrinsic Functions

This appendix is a quick reference to the F77 Intrinsic Functions. The functions are presented alphabetically according to how they are referenced. In most instances the Specific Name of the function is used, but in those instances where there is no Specific Name, the list uses its Generic Name. For a detailed discussion of the Intrinsic Functions see Chapter 8.

Alphabetic Summary of F77 Intrinsic Functions

How Referenced	Number of Arguments	Argument Type	Result Type	Generic Name
ABS	1	Real	Real	ABS
ACOS	1	Real	Real	ACOS
AIMAG	1	Complex	Real	
AINT	1	Real	Real	AINT
ALOG	1	Real	Real	LOG
ALOG10	1	Real	Real	LOG10
AMAX0	>= 2	Integer	Real	MAX
AMAX1	>= 2	Real	Real	MAX
AMIN0	>= 2	Integer	Real	
AMIN1	2	Real	Real	MIN
AMOD	2	Real	Real	MOD
AND	Any	Integer	Integer	
ANINT	1	Real	Real	ANINT
ASIN	1	Real	Real	ASIN

Alphabetic Summary of F77 Intrinsic Functions

How Referenced	Number of Arguments	Argument Type	Result Type	Generic Name
ATAN	1	Real	Real	ATAN
ATAN2	2	Real	Real	ATAN2
CABS	1	Complex	Real	ABS
CCOS	1	Complex	Complex	COS
CDABS	1	Complex*16	Double	ABS
CDCOS	1	Complex*16	Complex*16	COS
CDEXP	1	Complex*16	Complex*16	EXP
CDLOG	1	Complex*16	Complex*16	LOG
CDSIN	1	Complex*16	Complex*16	SIN
CDSQRT	1	Complex*16	Complex*16	SQRT
CEXP	1	Complex	Complex	EXP
CHAR	1	Integer	Character	
CLOG	1	Complex	Complex	LOG
CMPLX	1 or 2	Integer	Complex	CMPLX
CMPLX	1 or 2	Real	Complex	CMPLX
CMPLX	1 or 2	Double	Complex	CMPLX
CMPLX	1 or 2	Real*16	Complex*16	CMPLX
CMPLX	1 or 2	Complex	Complex	CMPLX
CMPLX	1 or 2	Complex*16	Complex	CMPLX
CONJG	1	Complex	Complex	CONJG
COS	1	Real	Real	COS
COSH	1	Real	Real	COSH
CSIN	1	Complex	Complex	SIN
CSQRT	1	Complex	Complex	SQRT
DABS	1	Double	Double	ABS
DACOS	1	Double	Double	ACOS
DASIN	1	Double	Double	ASIN
DATAN	1	Double	Double	ATAN
DATAN2	2	Double	Double	ATAN2
DBLE	1	Integer	Double	DBLE
DBLE	1	Real	Double	DBLE
DBLE	1	Double	Double	DBLE
DBLE	1	Complex	Double	DBLE
DBLEQ	1	Real*16	Double	DBLE
DCMPLX	1 or 2	Integer	Complex*16	DCMPLX
DCMPLX	1 or 2	Real	Complex*16	DCMPLX
DCMPLX	1 or 2	Double	Complex*16	DCMPLX
DCMPLX	1 or 2	Real*16	Complex*16	DCMPLX
DCMPLX	1 or 2	Complex	Complex*16	DCMPLX
DCMPLX	1 or 2	Complex*16	Complex*16	DCMPLX
DCONJG	1	Complex*16	Complex*16	CONJG
DCOS	1	Double	Double	COS
DCOSH	1	Double	Double	COSH
DDIM	2	Double	Double	DIM
DEXP	1	Double	Double	EXP
DIM	2	Real	Real	DIM
DIMAG	1	Complex*16	Double	
DINT	1	Double	Double	AINT

Alphabetic Summary of F77 Intrinsic Functions

How Referenced	Number of Arguments	Argument Type	Result Type	Generic Name
DLOG	1	Double	Double	LOG
DLOG10	1	Double	Double	LOG10
DMAX1	>= 2	Double	Double	MAX
DMIN1	2	Double	Double	MIN
DMOD	2	Double	Double	MOD
DNINT	1	Double	Double	ANINT
DPROD	2	Real	Double	
DREAL	1	Complex*16	Double	DBLE
DREAL	1	Complex*16	Double	
DSIGN	2	Double	Double	SIGN
DSIN	1	Double	Double	SIN
DSINH	1	Double	Double	SINH
DSQRT	1	Double	Double	SQRT
DTAN	1	Double	Double	TAN
DTANH	1	Double	Double	TANH
EXP	1	Real	Real	EXP
FLOAT	1	Integer	Real	REAL
IABS	1	Integer	Integer	ABS
ICHAR	1	Character	Integer	
IDIM	2	Integer	Integer	DIM
IDINT	1	Double	Integer	INT
IDNINT	1	Double	Integer	NINT
IFIX	1	Real	Integer	INT
INDEX	2	Character	Integer	
INT	1	Integer	Integer	INT
INT	1	Real	Integer	INT
INT	1	Complex	Integer	INT
INT	1	Complex*16	Integer	INT
INTL	1	Integer	Integer*4	INTL
INTL	1	Real	Integer*4	INTL
INTL	1	Double	Integer*4	INTL
INTL	1	Real*16	Integer*4	INTL
INTL	1	Complex	Integer*4	INTL
INTL	1	Complex*16	Integer*4	INTL
INTS	1	Integer	Integer*2	INTS
INTS	1	Real	Integer*2	INTS
INTS	1	Double	Integer*2	INTS
INTS	1	Real*16	Integer*2	INTS
INTS	1	Complex	Integer*2	INTS
INTS	1	Complex*16	Integer*2	INTS
IQINT	1	Real*16	Integer	INT
IQNINT	1	Real*16	Real*16	NINT
ISIGN	2	Integer	Integer	SIGN
LEN	1	Character	Integer	
LGE	2	Character	Logical	
LGT	2	Character	Logical	
LLE	2	Character	Logical	
LLT	2	Character	Logical	

Alphabetic Summary of F77 Intrinsic Functions

How Referenced	Number of Arguments	Argument Type	Result Type	Generic Name
LOC	1	Any but CHAR or LOG*1	Integer*4	
LS	2	Integer	Integer	
LT	2	Integer	Integer	
MAX0	>= 2	Integer	Integer	MAX
MAX1	>= 2	Real	Integer	
MIN0	2	Integer	Integer	MIN
MIN1	>= 2	Real	Integer	
MOD	2	Integer	Integer	MOD
NINT	1	Real	Integer	NINT
NOT	1	Integer	Integer	
OR	Any	Integer	Integer	
QABS	1	Real*16	Real*16	ABS
QACOS	1	Real*16	Real*16	ACOS
QASIN	1	Real*16	Real*16	ASIN
QATAN	1	Real*16	Real*16	ATAN
QATAN2	2	Real*16	Real*16	ATAN2
QCOS	1	Real*16	Real*16	COS
QCOSH	1	Real*16	Real*16	COSH
QDIM	2	Real*16	Real*16	DIM
QEXP	1	Real*16	Real*16	EXP
QINT	1	Real*16	Real*16	AINT
QLOG	1	Real*16	Real*16	LOG
QLOG10	1	Real*16	Real*16	LOG10
QMAX1	>= 2	Real*16	Real*16	MAX
QMIN1	2	Real*16	Real*16	MIN
QMOD	2	Real*16	Real*16	MOD
QNINT	1	Real*16	Real*16	ANINT
QPROD	2	Double	Real*16	
QSIGN	2	Real*16	Real*16	SIGN
QSIN	1	Real*16	Real*16	SIN
QSINH	1	Real*16	Real*16	SINH
QSQRT	1	Real*16	Real*16	SQRT
QTAN	1	Real*16	Real*16	TAN
QTANH	1	Real*16	Real*16	TANH
REAL	1	Real	Real	REAL
REAL	1	Real*16	Real	REAL
REAL	1	Complex	Real	REAL
REAL	1	Complex*16	Real	REAL
REAL	1	Complex	Real	
RS	2	Integer	Integer	
RT	2	Integer	Integer	
SHFT	2 or 3	Integer	Integer	
SIGN	2	Real	Real	SIGN
SIN	1	Real	Real	SIN
SINH	1	Real	Real	SINH
SNGL	1	Double	Real	REAL

Alphabetic Summary of F77 Intrinsic Functions

How Referenced	Number of Arguments	Argument Type	Result Type	Generic Name
SQRT	1	Real	Real	SQRT
TAN	1	Real	Real	TAN
TANH	1	Real	Real	TANH
XOR	Any	Integer	Integer	

INDEX

Index

SURVEY

READER RESPONSE FORM

DOC4029-4LA FORTRAN 77 Reference Guide Fourth Edition

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

Readability: ___hard to understand ___average ___very clear

Technical level: ___too simple ___about right ___too technical

Technical accuracy: ___poor ___average ___very good

Examples: ___too many ___about right ___too few

Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

_____

_____

_____

4. What faults or errors gave you problems? _____

_____

_____

_____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760