Absoft Fortran

Language Reference Manual

absoft
development tools and languages

Absoft Fortran

Language Reference Manual

absoft
development tools and languages

5119 Highland Road, PMB 398
Waterford, MI 48327
U.S.A.
Tel (248) 220-1190
Fax (248) 220-1194
support@absoft.com

All rights reserved. No part of this publication may be reproduced or used in any form by any means, without the prior written permission of Absoft Corporation.

THE INFORMATION CONTAINED IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE AND RELIABLE. HOWEVER, ABSOFT CORPORATION MAKES NO REPRESENTATION OF WARRANTIES WITH RESPECT TO THE PROGRAM MATERIAL DESCRIBED HEREIN AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. FURTHER, ABSOFT RESERVES THE RIGHT TO REVISE THE PROGRAM MATERIAL AND MAKE CHANGES THEREIN FROM TIME TO TIME WITHOUT OBLIGATION TO NOTIFY THE PURCHASER OF THE REVISION OR CHANGES. IN NO EVENT SHALL ABSOFT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE PURCHASER'S USE OF THE PROGRAM MATERIAL.

U.S. GOVERNMENT RESTRICTED RIGHTS — The software and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. The contractor is Absoft Corporation, 5119 Highland Road, PMB 398, Waterford, Michigan 48327.

ABSOFT CORPORATION AND ITS LICENSOR(S) MAKE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. ABSOFT AND ITS LICENSOR(S) DO NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL ABSOFT, ITS DIRECTORS, OFFICERS, EMPLOYEES OR LICENSOR(S) BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF ABSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. Absoft and its licensor(s) liability to you for actual damages for any cause whatsoever, and regardless of the form of the action (whether in contract, tort, (including negligence), product liability or otherwise), will be limited to $50.

Absoft, the Absoft logo, Fx3, and Pro Fortran are trademarks of Absoft Corporation
Apple, the Apple logo, and OS X are registered trademarks of Apple Computer, Inc.
AMD64 and Opteron are trademarks of AMD Corporation
Macintosh is a trademarks of Apple Computer, Inc., used under license.
Pentium, Pentium Pro, and Pentium II are trademarks of Intel Corp.
Windows 95/98/NT/ME/2000/XP/Windows 7 and Windows 8 are trademarks of Microsoft Corp.
All other brand or product names are trademarks of their respective holders.

Copyright © 1991-2022 Absoft Corporation and its licensor(s).
All Rights Reserved
Printed and manufactured in the United States of America. 1.0.22020622

Fortran Language Reference
Contents

ABSOFT FORTRAN I

CHAPTER 1 INTRODUCTION 1

Introduction to This Manual 1

Introduction to Absoft Fortran 1

Compatibility 1

Conventions Used in this Manual 2

CHAPTER 2 THE FORTRAN PROGRAM 3

Character Set 3

Symbolic Names 4

Keywords 4

Labels 5

Statements 5
- Executable Statements 5
- Nonexecutable Statements 6
- Statement Format 6
  - FORTRAN 77 Fixed Format 7
  - Fortran 90 Free Format 9
  - VAX FORTRAN Tab-Format 10
- Multiple Statement Lines 10
- Statement Order 11
- INCLUDE Statement 11
- Conditional Compilation Statements 12

type and KIND 14

Data Items 14
- Constants 15
  - Character Constant 15
  - Logical Constant 16
  - Integer Constant 17
    - Alternate Integer Bases 17
  - Real Constant 18
  - Complex Constant 19
  - Hollerith Constant 19
- Variables 20
- Substrings 20

Storage 20

Numeric Storage Unit 21
Character Storage Unit 21
Storage Sequence 21
Storage Association 22
Storage Definition 22

CHAPTER 3 SPECIFICATION AND DATA STATEMENTS 25

Type Statements 25
Arithmetic and Logical Type Statements 25
Character Type Statement 29

DERIVED TYPES 30
Structure Constuctor 31
Derived Type Component Reference 31

Pointers 32
Pointer Aliases 33
Pointer Bounds Remapping 33
Arrays of Pointers 34
ASSOCIATED() Function 34
NULL() Function 34
NULLIFY Statement 34

ALLOCATABLE Statement 35

ASYNCHRONOUS Statement 35

AUTOMATIC Statement 35

DIMENSION Statement 35

COMMON Statement 36

CONTAINS Statement 36

EQUIVALENCE Statement 37
Equivalence of Arrays 38
Equivalence of Substrings 38
COMMON and EQUIVALENCE Restrictions 38

Enumerations 38

EXTERNAL Statement 39

IMPLICIT Statement 39

INTENT Statement 40

INTRINSIC Statement 41

NAMELIST Statement 42

OPTIONAL Statement 42

PARAMETER Statement 43

POINTER Statement 43

POINTER Statement (Cray-Style) 44

PRIVATE And PUBLIC Statements 44

PROCEDURE Statement 45

RECORD Statement 46

SAVE Statement 46

SEQUENCE Statement 47

STDCALL Statement 47

STRUCTURE Declaration 47

UNION Declaration 49

TARGET Statement 50

VALUE Statement 50

VOLATILE Statement 50

DATA Statement 51
Implied DO List In A DATA Statement 52

CHAPTER 4 ARRAYS 53

Arrays 53

Array Declarator 53

Array Subscript 54

Array Valued Expressions 56

Assumed Shape Array 56

Automatic Array 57

Deferred shape Array 57

Pointer Arrays 58

Array sections 59

Array Constructor 60

vector subscript 60

array conformance **61**

array operations **62**

array valued functions **62**

CHAPTER 5 MODULES AND INTERFACES 63

modules **63**
Module Structure 63
Module Use 64
PRIVATE And PUBLIC Statements 64

interfaces **64**
Abstract Interfaces 68

CHAPTER 6 EXPRESSIONS AND ASSIGNMENT STATEMENTS 69

Arithmetic Expressions **69**
Data Type of Arithmetic Expressions 70
Arithmetic Constant Expression 71

Character Expressions **72**

Relational Expressions **72**

Logical Expressions **73**

Operator Precedence **74**

Arithmetic Assignment Statement **75**

Logical Assignment Statement **75**

Character Assignment Statement **76**

ASSIGN Statement **76**

CHAPTER 7 CONTROL STATEMENTS 77

GOTO Statements **77**
Unconditional GOTO 77
Computed GOTO 77
Assigned GOTO 77

IF Statements **78**
Arithmetic IF 78
Logical IF 78
Block IF 78

Loop Statements **79**
Basic DO loop 79
DO Loop Execution 80
DO WHILE 81

Infinite DO....81
END DO statement....81
EXIT statement....82
CYCLE statement....82

CONTINUE Statement....82
Labeled DO loops....82

BLOCK CASE....83
Execution of a block CASE statement....84
Block CASE Example....85

STOP Statement....85

PAUSE Statement....85

END Statement....86

WHERE....86

FORALL....87

CHAPTER 8 INPUT/OUTPUT AND FORMAT SPECIFICATION....89

Records....89
Formatted Record....89
Unformatted Record....90
Endfile Record....90

Files....90
File Name....90
File Position....90
File Access....90
Internal Files....91
Input/Output Editing....91

I/O Specifiers....92
Unit Specifier....92
Implicit File Connections....93
Format Specifier....93
Namelist Specifier....94
Record Specifier....94
Error Specifier....94
End of File Specifier....94
Asynchronous Specifier....95
Blank Specifier....95
Decimal Specifier....95
Delim Specifier....95
ID Specifier....95
Pad Specifier....96
Pos Specifier....96
Sign Specifier....96
I/O Status Specifier....97
I/O Message Specifier....97
Partial Record Specifier....97

I/O List **98**
Implied DO List In An I/O List 98

DATA TRANSFER STATEMENTS **98**
READ, WRITE and PRINT 98
ACCEPT and TYPE 100
Unformatted Data Transfer 100
Formatted Data Transfer 100
Printing 100

OPEN Statement **101**

CLOSE Statement **104**

BACKSPACE Statement **104**

REWIND Statement **105**

ENDFILE Statement **105**

FLUSH Statement **106**

WAIT Statement **106**

INQUIRE Statement **107**

ENCODE and DECODE Statements **111**

Giving a FORMAT Specification **112**

FORMAT and I/O List Interaction **113**

Input Validation **114**

Integer Editing **115**
I Editing 115
B, O, and Z Editing 115

Floating Point Editing **116**
F Editing 116
E and D Editing 116
EN Editing 117
ES Editing 117
G Editing 118
P Editing 118

Character and Logical Editing **119**
A Editing 119
L Editing 119

Sign Control Editing **120**

Blank Control Editing **120**

Decimal Symbol Editing **120**

Positional Editing **120**
X Editing 120
T, TL, and TR Editing 121
Slash Editing 121
Dollar Sign and Backslash Editing 121

Colon Editing **122**

Apostrophe and Hollerith Editing **122**
Apostrophe Editing 122
H Editing 122

Q Editing **122**

Variable Format Expressions **123**

List Directed Editing **123**
List Directed Input 123
List Directed Output 124

Namelist Directed Editing **124**
Namelist Directed Input 125
Namelist Directed Output 127

CHAPTER 9 PROGRAMS, SUBROUTINES, AND FUNCTIONS 129

Programs **129**

Subroutines **130**
Subroutine Arguments 130

Functions **131**
External Functions 131
Statement Functions 132
Intrinsic Functions 133

Recursion **133**

Pure Procedures **133**

Elemental Procedures **134**

ENTRY Statement **134**

RETURN Statement **134**

Passing Procedures in Dummy Arguments **135**

Passing Return Addresses in Dummy Arguments **135**

Common Blocks **135**

BLOCK DATA **135**

CHAPTER 10 INTRINSIC PROCEDURES.. 137

Intrinsic Procedure summary **137**
numeric inquiry functions 137
array inquiry functions 137
conversion functions 138
numeric computation functions 138
character computation functions 139
bit computation functions 140
array computation functions 140
intrinsic subroutines 141
pointer functions 141

Alphabetical listing of intrinsic procedures **141**

Intrinsic Procedure Details **145**

APPENDIX A ASCII TABLE 179

APPENDIX B EXCEPTIONS AND IEEE ARITHMETIC 183

IEEE_FEATURES **183**
IEEE_FEATURES_TYPE 183

IEEE_ARITHMETIC **184**
IEEE_CLASS_TYPE 184
IEEE_ROUND_TYPE 185
Subroutines and Functions 185

IEEE_EXCEPTIONS **189**
IEEE_FLAG_TYPE 190
IEEE_STATUS_TYPE 190
Subroutines and Functions 190

Examples **192**

APPENDIX C LANGUAGE BINDING MODULES 193

ISO_FORTRAN_ENV MODULE **193**

ISO_C_Binding Module **194**
KIND Parameters 194
CHARACTER CONSTANTS 195
POINTER CONSTANTS 195
TYPE DEFINITIONS 195
PROCEDURES 195

APPENDIX D ERROR MESSAGES 199

APPENDIX E TECHNICAL SUPPORT 203

APPENDIX F FLOATING POINT NUMBERS....205

APPENDIX G EXTENSIONS TO INTRINSIC FUNCTIONS....207

CHAPTER 1

Introduction

INTRODUCTION TO THIS MANUAL

This is the common language reference manual for the Absoft Fortran 77, Fortran 90, and Fortran 95 implementations. Operating systems supported on the platforms include: Windows, Linux, and Macintosh OS X. Absoft Fortran compilers in these environments are 100% source compatible and most control options are identical. Options relevant only to a specific environment are noted as such.

INTRODUCTION TO ABSOFT FORTRAN

Absoft Fortran is a complete implementation of the FORTRAN programming languages: FORTRAN 77, Fortran 90, and Fortran 95. It also completely implements ISO Technical Reports TR15580 and TR15581. The microprocessor-based computers of today are vastly more powerful and sophisticated than their predecessors. They offer more RAM, faster clock speeds, advanced scheduling and multiple core capabilities at very low prices..

Absoft Fortran is designed especially for these modern CPUs. It is not an evolutionary descendent of older compiler technology. Absoft Fortran brings a complete software development tool set with exceptional flexibility and improved ease-of-use to modern personal computers.

COMPATIBILITY

Absoft Fortran provides excellent compatibility with legacy code developed on mainframes and workstations. Most popular VAX/VMS statement extensions are accepted, as well as several from IBM/VS, Cray, Sun, and others. See the chapter **Porting Code** in the *Pro Fortran User Guide* for additional compatibility information.

CONVENTIONS USED IN THIS MANUAL

There are a few typographic and syntactic conventions used throughout this manual for clarity.

- `[]` square brackets indicate that a syntactic item is optional.
- ... indicates a repetition of a syntactic element.
- Term definitions are underlined.
- **-option** font indicates a compiler option.
- *Italics* is used for emphasis and book titles.
- On-screen text such as menu titles and button names look the same as in pictures and on the screen (e.g. the **File** menu).
- The modifier keys on PC keyboards are Shift, Alt, and Control. They are always used with other keys and are referenced as in the following:

Shift-G	press the Shift and 'G' keys together
Alt-F4	press the Alt and F4 function keys together
Control-C	press the Control and 'C' keys together

- Unless otherwise indicated, all numbers are in decimal form.
- FORTRAN examples appear in the following form:

```
PROGRAM HELLO
WRITE (*,*) "Hello World!"
END PROGRAM HELLO
```

- Language features that have been deleted or are categorized as obsolescent from the ISO Fortran standard are highlighted in salmon in this manual. They are no longer part of the language. While they are supported, their use should be avoided.
- Extensions to the Fortran language are highlighted in gray in this manual. Extensions are provided only for compatibility with legacy Fortran programs. Their use should be avoided.

CHAPTER 2

The Fortran Program

Fortran source programs consist of one program unit called the main program and any number of related program units called subprograms. A program or program unit is constructed as an ordered set of statements that describes procedures for execution and information to be used by the Fortran compiler during the compilation of a source program. Every program unit is written using the Fortran character set and follows a prescribed statement line format. A program unit may be one of the following:

- Main program
- Subroutine subprogram
- Function subprogram
- External subprogram
- Module
- Block Data subprogram

This chapter describes the format of Fortran programs and the data objects that may be manipulated by them.

CHARACTER SET

The compiler's character set consists of the following characters:

- Uppercase and lowercase letters: `A-Z, a-z`
- Digits: `0-9`
- Blank character
- Special characters: `+-*/=.,()$'_:%!<>"&`

Any of these characters, as well as the remaining printable ASCII characters, may appear in character and Hollerith constants (see below).

SYMBOLIC NAMES

A symbolic name is used to identify a Fortran entity, such as a variable, array, program unit, module, or labeled common block. The first character of a symbolic name must be a letter. Addition characters may be letters, digits, or the underscore (_) character. A symbolic name may contain up to 63 significant characters. Symbolic names of greater than 63 characters are acceptable, but only the first 63 characters are significant to the compiler

The blank character may not be used within a Fortran 95 source format symbolic name. The blank character is not significant in a Fortran 77 source format symbolic name but may be used as a separator.

Global symbolic names are known to every program unit within an executable program and therefore must be unique. The names of main programs; subroutine, function, block data subprograms; modules; and common blocks are examples of global symbolic names.

Local symbolic names are known only within the program unit in which they occur. The names of variables, arrays, symbolic constants, statement functions, contained subprograms, and dummy procedures are local symbolic names.

KEYWORDS

A keyword is a sequence of characters that has a predefined meaning to the compiler. A keyword is used to identify a statement or serve as a separator in a statement. Some typical statement identifiers are `READ`, `FORMAT`, and `REAL`. Two separators are `TO` and `THEN`.

There are no reserved words in Fortran, therefore a symbolic name may assume the exact sequence of characters as a keyword. The compiler determines the meaning of a sequence of characters through the context in which the characters are used. A surprising example of a keyword/symbolic name exchange in Fortran 77 is:

Statement	Meaning
`DO10I=1,7`	Control statement
`DO 10 I = 1. 7`	Assignment statement

Note that the embedded blanks are not significant in Fortran 77 source format nor are they required as separators for the compiler to determine that the first statement is the initial statement of a `DO` loop. The absence of a comma in the second statement informs the compiler that an assignment is to be made to the variable whose symbolic name is `DO10I`.

In Fortran 95 source format, blanks are required as separators.

LABELS

A statement label may appear on a Fortran statement initial line. Actual placement of a label on the initial line is governed by rules described later in this chapter in the section **Statement Format**. A statement label is used for reference in other statements. The following considerations govern the use of the statement label:

- The label is an unsigned integer in the range of 1 to 99999.
- Leading zeros not significant to the compiler.
- A label must be unique within a program unit.
- A label is not allowed on a continuation line.
- Labels may appear in any numeric order.

Examples of labels:

```
1101
88
10
010
```

The last two examples produce the same label.

STATEMENTS

Individual statements deal with specific aspects of a procedure described in a program unit and are classified as either executable or nonexecutable. The proper usage and construction of the various types of statements is described in the following chapters.

Executable Statements

Executable statements specify actions and cause the Fortran compiler to generate executable program instructions. There are 3 types of executable statements:

- Assignment statements
- Control statements
- Input/Output statements

Nonexecutable Statements

Nonexecutable statements are used as directives to the compiler: start a new program unit, allocate variable storage, initialize data, set the options, etc. There are 9 types of nonexecutable statements:

- Specification statements
- Data initialization statements
- `FORMAT` statements
- Function defining statements
- Subprogram statements
- Main program statements
- Module statements
- Interface statements
- Compiler directives

Statement Format

A FORTRAN statement consists of one or more source records referred to as a statement line. Historically, a record is equivalent to a card. In current source file formats, a record is one line of text terminated by an end of record character (generally a line feed, or carriage return-line feed pair).

Two statement formats are supported by the compiler and are described in the following sections.

FORTRAN 77 fixed format is selected if the file name ends with `.f`, `.for`, or the compiler option **–f fixed** is specified.

Fortran 90 free format is selected if the file name ends with `.f90`, `.f95`, or the compiler option **–f free** is specified.

FORTRAN 77 Fixed Format

A FORTRAN 77 statement line consists of 80 character positions or columns, numbered 1 through 80 that are divided into 4 fields.

The **–W132** compiler option may be used to expand the statement field to column 132.

Field	Columns	
Statement label	1-5	
Continuation	6	
Statement	7-72	
	7-132	(using –W132 compiler option)
Identification	73-80	
	132+	(using –W132 compiler option)

The Identification field is available for any purpose the programmer may desire and is ignored by the Fortran compiler. Historically this field has been used for sequence numbers and commentary. The statement line itself may exceed the column of the last field; the compiler ignores all characters beyond the last field.

There are four types of source lines in Fortran:

Comment Line – used for source program annotation and formatting. A comment line may be placed anywhere in the source program and assumes one of the forms:

- Column 1 contains the character C or an asterisk. The remainder of the line is ignored by the compiler.
- Column 1 contains an exclamation point. The remainder of the line is ignored by the compiler.
- The line is completely blank.
- An exclamation point not contained within a character constant designates all characters including the exclamation point through the end of the line to be commentary.
- Column 1 contains the character D, d, X, or x and the conditional compilation **-x** compiler option is not on.

Comment lines have no effect on the object program and are ignored by the Fortran compiler.

End Line – the last line of a program unit.

- The word END must appear within the statement field.
- Each Fortran program unit must have an END line as its last line to inform the compiler that it is at the physical end of the program unit.
- An END line may follow any other type of line.

Initial Line – the first and possibly only line of each statement.

- Columns 1 through 5 may contain a statement label to identify the statement.
- Column 6 must contain a zero or a blank.
- Statement field contains all or part of the statement.

Continuation Line – used when additional characters are required to complete a statement originating on an initial line.

- Columns 1 through 5 must be blank.
- Column 6 must contain a character other than zero or blank.
- Statement field contains the continuation of the statement.
- There may be only 999 continuation lines per statement, for a total of 1000 lines per statement.

Fortran 90 Free Format

A Fortran 90 free format statement line consists of 132 character positions. In this source format, there are no "fields" in which labels, statements, or comments must appear.

A statement label must appear before the statement on a line; it may be in *any* column:

```
100 I=123
J=456
200 I=123
       300      I=123
400
    I=123
```

Comments in Fortran 90 format begin with an exclamation point character "!" in any column not in character context. The letter C in column 1 does not indicate a comment.

```
! This entire line is a comment
A=1 !A trailing comment
                     ! Blank line
C="ab!cd"!The exclamation point in quotes does not begin a comment
```

To continue a statement across multiple lines, the ampersand character (&) is used according to the following rules:

- The "&" as the *last* non-blank character on a line signifies the line is continued on the next line. Comment lines may not be continued. A comment, beginning with "!", may appear after the "&" when not in character context.
- The "&" as the *first* non-blank character on the next line will cause continuation to begin after the "&". Otherwise, continuation begins with the first character. When continuing character context, the next line *must* begin with a "&" as the first non-blank character.
- There may be only 999 continuation lines per statement, for a total of 1000 lines per statement.

The following valid program demonstrates Fortran 90 continuation:

```
! Fortran 90 example
character s,&
          t,&
         &u
s="This string &
  &contains NO &
  &ampersand symbols" !Comment
t="One ampersand:&&
  &"
           1&
               &00 I=1 !This line has label 100
```

VAX FORTRAN Tab-Format

A VAX FORTRAN tab-format statement line consists of 80 character positions or columns with fields similar to those in the Fortran format. The TAB character is used to begin the continuation and statement fields rather than having them tied to a specific column. The tab-format is primarily useful for entering FORTRAN source with many editors since it is generally easier to hit the TAB key once as opposed to hitting the space bar multiple times before a statement. A TAB character elsewhere in a FORTRAN statement is treated as spaces. Tab-format may be freely mixed with fixed format source.

A statement label must appear anywhere on a line before the first TAB character:

```
1(TAB) WRITE(*,*) "This line has label 1"
 (TAB) WRITE(*,*) "No label on this line"
1  2  3(TAB) WRITE(*,*) "This line has label 123"
```

Comments in VAX FORTRAN tab-format begin with a C or asterisk in column 1 or an exclamation point character "!" in any column not in character context. Having a D or X in column 1 will also comment an entire line unless the **-x** compiler option is on for conditional compilation:

```
! Full-line comment
 (TAB) I=123  ! Statement begins after TAB
D      J=456  ! Compiled only with x option
```

To continue a statement across multiple lines, the continuation line must have a non-zero digit after the first tab. Note that the initial line cannot start with a digit – no FORTRAN statement begins with a digit:

```
 (TAB) WRITE(*,*) "This line spans
 (TAB) 1multiple lines because
 (TAB) 2 of the non-zero continuation digit after first
 (TAB) 3 tab character on each line"
```

The **–W132** compiler option will expand the statement field to column 132.

Multiple Statement Lines

Multiple statements may be placed on the same line by separating them with a semicolon (;).

```
I=10; J=10; N(I,J)=0
```

Statement Order

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA or MODULE</td></tr>
<tr><td colspan="3">USE</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>PARAMETER
and
DATA</td><td>Derived-Type definitions, Interface Blocks, Type Declaration statements, statement function statements, and specification statements</td></tr>
<tr><td>DATA</td><td>Executable Statements</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">Internal or module procedures</td></tr>
<tr><td colspan="3">END</td></tr>
</table>

Required Statement Order

INCLUDE Statement

This statement is a compiler directive and is provided as a convenience for copying standard declaration statements and documentation sections directly into a source file at compile time. The syntax of this statement is:

```
INCLUDE filespec
```

where: *filespec* is a standard file specification presented as a character constant (i.e. enclosed in quotation marks or apostrophes).

INCLUDE statements may be nested up to 10 files deep.

The **-I** compiler option, described in the chapter **Using the Compilers** in the *Pro Fortran User Guide*, is useful for specifying directories and search paths for include files.

Conditional Compilation Statements

In addition to the previously described limited capability for conditional compilation available by placing a D or an X in column one, a complete set of compiler directives is also provided which gives dynamic control over the compilation process. These compiler directives are specified with a dollar sign ($) in column 1 and take the following forms:

```
$DEFINE name [value]
$UNDEFINE name

$IF expr
$ELSE
$ELIF expr
$ENDIF
```

`name` is the *case sensitive* symbolic name of a variable that is used only in conditional compilation directives. It can have the same name as any variable used in standard FORTRAN statements without conflict. Conditional compilation variables exist from the point they are defined until the end of the file unless they are undefined with the `$UNDEFINE` directive. `value` is an integer constant expression that is used to give the variable a value. If `value` is not present, the variable is given the value of 1.

`expr` is any expression using constants and conditional compilation variables which results in a logical value. Also provided is the logical function `DEFINED` which is used to determine if a variable has actually been defined. Consider the following:

```
$IF DEFINED(debug)
      WRITE (*,*) "iter=",iter
$ENDIF
```

In this case, you are interested in displaying the value of the variable `iter` only during the debugging stages of program development. To turn this feature on, all that is required is to define `debug` before the conditional compilation clause in the source file:

```
$DEFINE debug
```

A more complicated example:

```
      $DEFINE precision 8

      $IF precision == 4
            REAL a,b,c,d(100),pi
            pi = atan(1.0)*4.0
      $ELIF precision == 8
            DOUBLE PRECISION a,b,c,d(100),pi
            pi = datan(1d0)*4d0
      $ENDIF
```

Conditional compilation variables that are not defined can only be referenced as arguments to the `DEFINED` function. Any other use will result in a compile error.

Inline comments are ***not*** allowed.

Conditional compilation statements are particularly useful for managing large groups of include files with nested dependencies. Suppose you are using an include file named "graphics.inc" that declares certain structures which are dependent on another include file named "types.inc". If you add the statement `$DEFINE TYPES` at the end of the "types.inc" include file and add the following three statements to the beginning of the "graphics.inc" include file:

```
      $IF !DEFINED(TYPES)
            INCLUDE "types.inc"
      $ENDIF
```

your source program file only needs to include "graphics.inc" to compile successfully.

The conditional compilation statements are:

`$DEFINE` *`variable`* `[`*`value`*`]`	define a variable [with a *`value`*]
`$UNDEFINE` *`variable`*	undefined a variable
`$IF` *`expression`*	begin an *if* clause
`$ELSE`	begin an *else* clause
`$ELIF` *`expression`*	begin an *else if* clause
`$ENDIF`	end an *if* clause

The conditional compilation operators are:

<table>
<tr><td><</td><td>less than</td></tr>
<tr><td><=</td><td>less than or equal</td></tr>
<tr><td>==</td><td>equal</td></tr>
<tr><td>!=</td><td>unequal</td></tr>
<tr><td>></td><td>greater than</td></tr>
<tr><td>>=</td><td>greater than or equal</td></tr>
<tr><td>!</td><td>logical not</td></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>||</td><td>logical or</td></tr>
</table>

TYPE AND KIND

Fortran provides several categories of data known as *type*. These are integers, floating-point numbers, complex numbers, character, and logical values. Integers are an exact representation of integral values. Floating-point numbers are approximations of real numbers. Complex numbers are composed of two floating-point numbers representing the real and imaginary portions of the value. Character values are representations of individual or strings of printable characters. Logical data items represent the Boolean values of true and false. The representation and presentation of these data types are described in detail in the following sections of this chapter.

Fortran further provides for data of the same *type* to support multiple precision, range, and size or length. The numeric model is known as *kind*. Several features are available for inquiring about the numeric model and specifying the kind of the data entity. They are:

`KIND(x)`	*x* is a numeric expression.
`SELECTED_INT_KIND(p)`	*p* is an integer expression.
`SELECTED_REAL_KIND(p, r)`	*p* and *r* are integer expressions

These functions return an integer value that in the Absoft implementation of Fortran represents the number of bytes required to store a value specified by the expression. The value of the expression *p* represents the number of required digits of precision. The value of the expression *r* represents the number of required range of the exponent. If the value cannot be represented, the functions return a value of `-1`. Examples:

Function	Value
`KIND(1.0)`	4
`KIND(32873)`	4
`SELECTED_INT_KIND(3)`	2
`SELECTED_INT_KIND(5)`	4
`SELECTED_INT_KIND(12)`	8
`SELECTED_REAL_KIND(6,30)`	4
`SELECTED_REAL_KIND(14,100)`	8
`SELECTED_REAL_KIND(30,300)`	16

DATA ITEMS

The symbolic name used to represent a constant, variable, array, substring, statement function, derived type, or external function identifies its data type, and once established, it

does not change within a particular program unit. The data type of an array element name is always the same as the type associated with the array.

Special FORTRAN statements, called type statements, may be used to specify a data type as character, logical, integer, real, double precision, or complex. When a type statement is not used to explicitly establish a type, the first letter of the name is used to determine the type. If the first letter is I, J, K, L, M, N, i, j, k, l, m, or n, the type is integer; any other letter yields an implied type of real. The `IMPLICIT` statement, described later, may be used to change the default implied types.

The `IMPLICIT NONE` statement, also described later, causes the compiler to require the declaration of all variables.

An intrinsic function, `LOG`, `EXP`, `SQRT`, `INT`, etc., may be used with either a specific name or generic name. The data types of the specific intrinsic function names are given in the **Programs, Subroutines, and Functions** chapter. A generic function assumes the data type of its arguments as discussed in that chapter.

A main program, subroutine, common block, module, and block data subprogram are all identified with symbolic names, but have no data type.

Constants

Fortran constants are identified explicitly by stating their actual value; they do not change in value during program execution. The plus (+) character is not required for positive constants. The value of zero is neither positive nor negative; a zero with a sign is just zero.

The data type of a constant is determined by the specific sequence of characters used to form the constant. A constant may be given a symbolic name with the `PARAMETER` statement.

For fixed format source programs. except within character and Hollerith constants, blanks are not significant and may be used to increase legibility. For example, the following forms are equivalent:

```
3.14159265358979        3.1415 92653 58979
2.71828182845904        2.7182 81828 45904
```

Blanks are significant in a free format source program and may not occur within a numeric or logical constant.

Character Constant

A character constant is a string of ASCII characters delimited by either apostrophes (`'`) or quotation marks (`"`). The character used to delimit the string may be part of the string itself by representing it with two successive delimiting characters. The number of charac-

ters in the string determines the length of the character constant. A character constant requires a character storage unit (one byte) for each character in the string.

```
"TEST"                      'TEST'
"EVERY GOOD BOY"            'EVERY GOOD BOY'
"Luck is everything"        'Luck is everything'
"didn't"                    'didn''t'
```

As an extension to Fortran, special escape sequences may be embedded in a character constant by using the backslash (\) followed immediately by one of the letters in the following list. The actual character value generated in place of the escape sequence is system dependent. For compatibility with FORTRAN programs which do not expect the backslash as an escape sequence, these escape sequences are not recognized by the compiler unless the **–YCSLASH=1** option is used.

Escape Sequence	Meaning
`\n`	Newline
`\t`	Tab
`\r`	Carriage return
`\f`	Form feed
`\b`	Backspace
`\\`	Backslash
`\`*nnn*	Sets character position with value *nnn*, where *nnn* are octal digits.

For example,

```
WRITE(*,*) "First line\nSecond line"
```

When compiled *without* the **- YCSLASH=1** option, it displays,

```
First line\nSecond line
```

When compiled *with* the **- YCSLASH=1** option, it displays,

```
First line
Second line
```

Logical Constant

Logical constants are formed with the strings of characters, `.TRUE.` and `.FALSE.`, representing the Boolean values true and false respectively. A false value is represented with the value zero, and a true value is represented with the value one. A default logical constant requires one numeric storage unit (four bytes).

Integer Constant

An integer constant is an exact binary representation of an integer value with negative integers maintained in two's complement form. An integer constant is a string of decimal digits that may contain a leading sign

```
15
101
-72
1126
123456789
```

The default range of an integer constant is -2147483648 to +2147483647. This is internally represented using four bytes of memory. One, two, and eight byte integer constants are available by attaching a *kind* parameter to the end of the constant:

```
22_1
367_2
54893657452854_8
-8746359852145_8
```

The range of an one-byte integer constant is -128 to +127. The range of an two-byte integer constant is -32768 to +32767. The range of an eight-byte integer constant is -9223372036854775808 to +9223372036854775807.

The *kind* parameter can be represented symbolically:

```
INTEGER, PARAMETER :: I2=SELECTED_INT_KIND(3)
J = 224_I2
```

Alternate Integer Bases

The compiler normally expects all numeric constants to be in base ten, however, three alternate unsigned integer bases are available when explicitly specified. These optional bases are binary, octal, and hexadecimal and are designated by preceding the constant with the characters `B`, `O`, and `Z` respectively and delimiting the constant itself with apostrophes. The following examples all result in the assignment of the decimal value 3994575:

```
I = B'1111001111001111001111'
J = O'17171717'
K = Z'3CF3CF'
```

The VAX FORTRAN form of hexadecimal and octal constants may also be used:

```
J = '017171717'O
K = '3CF3CF'X
```

Real Constant

A real constant consists of an optional sign and a string of digits that contains a decimal point. The decimal point separates the integer part of the constant from the fractional part and may be placed before or after the string indicating that either the integer or fractional part is zero. A real constant may have an exponent that specifies a power of ten applied to the constant. An exponent is appended to a real constant with the letter `E`, `D`, or `Q` and an integer constant. The exponent letters, `E`, `D`, or `Q`, indicate the precision of the constant: single, double, and quad precision respectively. If an exponent is given and the fractional part is zero, the decimal point may be omitted.

Constant	Value
1E2	100.0
-12.76	-12.76
1.07D-1	.107
0.4237E3	423.7
7.892Q2	789.2

Single precision values are maintained in IEEE single precision floating point representation. The exponent range is -37 to +39. The most significant bit is interpreted as the sign, the next eight bits provide a binary exponent biased by 127, and the remaining twenty-three bits form the binary mantissa with a twenty-fourth bit implied. This representation supplies seven digits of precision and a range of `±0.3402823E+39` to `±0.1175494E-37`.

Double precision values are maintained in IEEE double precision floating point representation. The exponent range is -307 to +309. The most significant bit is interpreted as the sign, the next eleven bits provide a binary exponent biased by 1023, and the remaining fifty-two bits form the binary mantissa with a fifty-third bit implied. This representation yields sixteen digits of precision and a range of `±0.1797693134862320D+309` to `±0.2225073858507202D-307`.

Quad precision values are maintained as a pair IEEE double precision floating point values, with the most significant value representing the exponent and the most significant portion of the mantissa and least significant value representing the least significant portion of the mantissa and least. The exponent range is -307 to +309. The most significant bit is interpreted as the sign, the next eleven bits provide a binary exponent biased by 1023, and the remaining fifty-two bits of both values form the binary mantissa. This representation yields thirty-one digits of precision and a range of ±0.1797693134862315708145274237317300Q+309 to ±0.222507385850720229569787989021279Q-307.

Real constants may also be qualified with a *kind* parameter:

```
1.0_4
33.7_8
-0.7193_16
```

The *kind* parameter can be represented symbolically:

```
INTEGER, PARAMETER :: DP=SELECTED_REAL_KIND(15,100)
A = 3.0_DP
```

Complex Constant

A complex constant is stated using a left parenthesis, a pair of real or integer constants separated by a comma, and a right parenthesis. The first constant is the real portion (in the mathematical sense) and the second is the imaginary portion. A complex constant requires two numeric storage units (eight bytes).

Constant	Value
(2.76,-3.81)	= 2.76 -3.81*i*
(-12,15)	= -12.0 +15.0*i*
(0.62E2,-0.22E-1)	= 62.0 -.022*i*

Hollerith Constant

The Hollerith data type is an older method of representing characters in FORTRAN. While it is not included in the current standard, this implementation of FORTRAN includes the Hollerith data type to provide compatibility for older programs. Like character constants, a Hollerith constant is formed with a string of any of the characters from the ASCII character set. Logical, integer, real, double precision, and complex variables can be defined with a Hollerith value through `DATA` statements and `READ` statements.

A Hollerith constant is stated with a nonzero, unsigned integer constant, the letter `H`, and a string of characters whose length must be the same as the integer constant.

```
4HTEST
14HEVERY GOOD BOY
```

When a Hollerith constant is assigned to a variable it is left justified and space padded if the length of the constant is less than the length of the variable. If a Hollerith constant appears anywhere in the source code except within a `DATA` statement, a `FORMAT` statement, or a `CALL` statement, an error will result. Embedded escape sequences (i.e. `\n`) are not permitted in a Hollerith constant.

Two additional forms of Hollerith constants are available that are padded with 0's (zeros) rather than spaces: `L` and `R`. Hollerith using the letter `L` are truncated from the right if they are too large. They are left justified and 0 padded if they are too short. Hollerith using the letter `R` are truncated from the left if they are too large. They are right justified and 0 padded if they are too short.

Variables

A variable is used to maintain a Fortran quantity and is associated with a single storage location through a symbolic name. Simple variables are often called scalar variables to distinguish them from arrays and array elements. Arrays are discussed in the next chapter. Unlike a constant, the value of a variable can be changed during the execution of a program with assignment statements and input and output statements.

Substrings

A substring is a contiguous segment of a character entity and is itself a character data type. It can be used as the destination of an assignment statement or as an operand in an expression. Either a character variable or character array element can be qualified with a substring name:

```
v( [e1] : [e2] )

a(s [,s]...)( [e1] : [e2] )
```

where: `e1` and `e2` are called substring expressions and must have integer values.

`v` is the symbolic name of a character variable, and `a(s [,s]…)` is the name of a character array element (see the **Arrays** chapter).

The values `e1` and `e2` specify the leftmost and rightmost positions of the substring. The substring consists of all of the characters between these two positions, inclusive. For example, if `A` is a character variable with a value of `'ABCDEF'`, then `A(3:5)` would have a value of `'CDE'`.

The value of the substring expression `e1` must be greater than or equal to one, and if omitted implies a value of one. The value of the substring expression `e2` must be greater than or equal to `e1` and less than or equal to the length of the character entity, and if omitted implies the length of the character entity.

Substring expressions may contain array or function references. The evaluation of a function within a substring expression must not alter the value of other entities also occurring within the substring expression.

STORAGE

Storage refers to the physical computer memory where variables and arrays are stored. Variables and arrays can be made to share the same storage locations through equivalences, common block declarations, and subprogram argument lists. Data items that share storage in this manner are said to be associated.

The contents of variables and arrays are either defined or undefined. All variables and arrays not initially defined through `DATA` statements are undefined.

A storage unit is primarily a FORTRAN 77 concept and refers to the amount of storage needed to record a particular class of data. A storage unit can be a numeric storage unit or a character storage unit.

Numeric Storage Unit

A numeric storage unit can be used to hold or store an integer, real, or logical datum. One numeric storage unit consists of four bytes. The amount of storage for default numeric data is as follows:

Data Type	Storage
Integer	1 storage unit
Real	1 storage unit
Double precision	2 storage units
Complex	2 storage units
Logical	1 storage unit

Character Storage Unit

A character datum is a string of characters. The string may consist of any sequence of ASCII characters. The length of a character datum is the number of characters in the string. A character storage unit differs from numeric storage units in that one character storage unit is equal to one byte and holds or stores one character.

Storage Sequence

The storage sequence refers to the sequence of storage units, whether they are held in memory or stored on external media such as a disk or a tape.

Storage Association

The storage locations of variables and arrays become associated in the following ways:

- The EQUIVALENCE statement (described in the **Specification and DATA Statements** chapter) causes the storage units of the variables and array elements listed within the enclosing parentheses to be shared. Note that the data types of the associated entities need not be the same.

- The variable and array names appearing in the COMMON statements (described in the **Specification and DATA Statements** chapter) of two different program units are associated.

- The dummy arguments of subroutine and function subprograms are associated with the actual arguments in the referencing program unit.

- An ENTRY statement (described in the **Programs, Subroutines, and Functions** chapter) in a function subprogram causes its corresponding name to be associated with the name appearing in the FUNCTION statement.

Storage Definition

Storage becomes defined through DATA statements, assignment statements, and I/O statements. READ statements cause the items in their associated I/O lists to become defined. Any I/O statement can cause items in its parameter list to become defined (the IOSTAT variable for instance). A DO variable becomes defined as part of the loop initialization process.

The fact that storage can become undefined at all should be carefully noted. Some events that cause storage to become undefined are obvious: starting execution of a program that does not initially define all of its variables (through DATA statements), attempting to read past the end of a file, and executing an INQUIRE statement on a file that does not exist. When two variables of different types are either partially or totally associated, defining one causes the other to become undefined.

Because Fortran provides for both dynamic as well as static storage allocation, certain events can cause dynamically allocated storage to become undefined. In particular, returning from subroutine and function subprograms causes all of their variables to become undefined except for those:

- in blank COMMON.

- specified in a MODULE

- specified in SAVE statements.

The **-s** compiler option has the effect of an implicit SAVE for every program unit encountered during the current compilation (see the chapter **Using the Compilers** in the *Pro Fortran User Guide*).

CHAPTER 3

Specification and DATA Statements

Specification statements are used to define the properties of the symbolic entities, variables, arrays, symbolic constants, etc. that are used within a program unit. For this reason, specification statements are also called declaration statements and are grouped together in the declaration section of a program unit: before any statement function statements, `DATA` statements, and executable statements. Specification statements themselves are classified as non-executable.

`DATA` statements are used to establish initial values for the variables and arrays used within a Fortran program. Variables *not* appearing in `DATA` statements may contain random values when a program starts executing. The use of undefined variables can cause problems that are difficult to detect when transporting a program from one environment to another, because the previous environment may have set all storage to zeros while the new environment performs no such housekeeping.

TYPE STATEMENTS

The most common of the specification statements are the type statements. They are used to give data types to symbol names and declare array names. Once a data type has been associated with a symbol name it remains the same for all occurrences of that name throughout a program unit.

Arithmetic and Logical Type Statements

The forms of the type statement for the arithmetic and logical data types are:

```
type  v [,v]...
type [*len] v[/value/] [,v[/value/]]...
type [(kind)] [[attributes] ::] v [,v]... [=value]
```

where: *type* can be any of the following specifiers: `LOGICAL`, `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `DOUBLE COMPLEX`.

v is the symbolic name of a variable, an array, a constant, a function, a dummy procedure, or an array declaration.

len is an unsigned integer constant that specifies the length, in bytes, of a variable, an array element, a symbolic constant, or a function.

`value` is an optional initial value for the preceding variable or array. When initializing an array, `value` must contain constants separated by commas for each element of the array.

`kind` is a kind expression that specifies the kind of the data entity (see below).

`attributes` is a list attributes that apply to the data entity (see below).

The following `len` specifiers are available:

- `LOGICAL*4` is the default for `LOGICAL` and occupies one numeric storage unit. The default may be changed to `LOGICAL*2` with the **-i*n*** compiler option (see the chapter **Using the Compilers** in the *Pro Fortran User Guide*).

- `LOGICAL*2` data is a representation of the logical values of true and false. This type of logical data occupies one half of one numeric storage unit. A false value is represented by the number zero and a true value is represented by the number one.

- `LOGICAL*1` data is a representation of the logical values of true and false. This type of logical data occupies one byte. A false value is represented by the number zero and a true value is represented by the number one.

- `INTEGER*8` data is an exact binary representation of an integer in the range of -9223372036854775808 to +9223372036854775807 with negative integers carried in two's complement form. This type of integer is maintained in two numeric storage units.

- `INTEGER*4` is the default for `INTEGER` and occupies one numeric storage unit. The default may be changed to `INTEGER*2` or `INTEGER*8` with the **-i*n*** compiler option (see the chapter **Using the Compilers** in the *Pro Fortran User Guide*).

- `INTEGER*2` data is an exact binary representation of an integer in the range of -32768 to +32767 with negative integers carried in two's complement form. This type of integer is maintained in one half of one numeric storage unit.

- `INTEGER*1` data is an exact binary representation of an integer in the range of -128 to +127 with negative integers carried in two's complement form. This type of integer is maintained in one byte of storage.

- `REAL*4` is the default for `REAL` and occupies one numeric storage unit. The default may be changed to `REAL*8` with the **-N113** compiler option (see the chapter **Using the Compilers** in the *Pro Fortran User Guide*).

- `REAL*8` data is identical to `DOUBLE PRECISION` and occupies two numeric storage units.

- `REAL*16` is quad precision and occupies 4 numeric storage units

- `COMPLEX*8` data is identical to `COMPLEX` and occupies two numeric storage units. The default may be changed to `COMPLEX*16` with the **-N113** compiler option (see the chapter **Using the Compilers** in the *Pro Fortran User Guide*).

- `COMPLEX*16` data is similar to `COMPLEX` except that both halves of the complex value are represented as `DOUBLE PRECISION` and it occupies four numeric storage units.

- `COMPLEX*32` data is similar to `COMPLEX` except that both halves of the complex value are represented as `REAL*16` and it occupies eight numeric storage units.

attributes consists of comma separated list of one or more of the following:

`PARAMETER`	allows a constant to be given a symbolic name. The symbol must be given a value.
`ALLOCATABLE`	specifies that the array will have a deferred shape that will be specified when it is allocated.
`ASYNCHRONOUS`	specifies that the variable may appear in an `ASYNCHRONOUS` I/O data transfer statement.
`DIMENSION`	is given with an array dimension declaration to be applied to the variable name(s) given.
`EXTERNAL`	allows symbolic names to be used as arguments in CALL statements and function references without the compiler automatically creating a variable at the point of reference.
`INTENT(`*inout*`)`	specifies the intended use of a dummy argument. *inout* may be one of `IN`, `OUT`, `INOUT`.
`INTRINSIC`	allows the specific or generic name of an intrinsic function to be used as an argument in CALL statements and function references without the compiler automatically creating a variable at the point of reference.

`OPTIONAL` specifies that the declared dummy argument is optional and may be omitted from the actual argument list of a subroutine or function reference.

`POINTER` specifies that the variable will be a pointer.

`SAVE` `prevents` variables and arrays that are declared locally in subprograms from being deallocated and losing their definition status when the program is exited.

`STDCALL` **Windows 32-bit only**. Indicates that the function in the specification is to use the Windows `STDCALL` call/return protocol.

`TARGET` allows the variable to be associated via a pointer assignment operator with a variable that has the POINTER attribute.

`AUTOMATIC` forces a variable to be allocated on the stack. This is the default in the absence of a `SAVE` statement or compiler directive.

`VALUE` specifies that the declared dummy argument is passed by value rather than the default of by reference.

`VOLATILE` informs the compiler that variable can be modified outside of Fortran and it should not be maintained in a CPU register.

`BIND(`*`spec`*`)` specifies that the entity has interoperability with C and external linkage. *`spec`* takes the form:

```
C [,NAME=character_constant]
```

where: *`character_constant`* is the external name of the entity.

kind examples:

```
REAL(4)
REAL(KIND(1.0D0))
INTEGER(SELECTED_INT_KIND(4))
```

Character Type Statement

The form of the type statement for the character data type is:

```
CHARACTER [*len [,]] v[*len] [,v[*len]]...
CHARACTER [(len)] [attributes] [::] v [,v]... [=value]
CHARACTER [*len [,]] v[*len][/value/] [,v[*len][/value/]]...
```

where: *v* is a variable name, an array name, an array declaration, the symbolic name of a constant, a function name, or a dummy procedure name

len is either an unsigned INTEGER constant, an INTEGER constant expression within parentheses, or an asterisk within parentheses and specifies the length, in bytes, of a variable, an array element, a symbolic constant, or a function.

value is an optional initial value for the preceding variable or array. When initializing an array, *value* must contain constants separated by commas for each element of the array.

If *len* directly follows the word CHARACTER, the length specification applies to all symbols not qualified by their own length specifications. When *len* is not specified directly after the keyword CHARACTER, all symbols not qualified by their own length specifications default to one byte.

The length of symbolic CHARACTER constants, dummy arguments of SUBROUTINE and FUNCTION subprograms, and CHARACTER functions may be given as an asterisk enclosed in parentheses: (*). The length of a symbolic constant declared in this manner is the number of characters appearing in the associated PARAMETER statement. Dummy arguments and functions assume the length of the actual argument declared by the referencing program unit.

```
CHARACTER TITLE*(*)
PARAMETER (TITLE = 'FORTRAN 77')
```

or:

```
CHARACTER(LEN=*), PARAMETER :: TITLE = 'Fortran 90'
```

produces a ten byte symbolic CHARACTER constant.

DERIVED TYPES

Derived types are useful for manipulating and maintaining data objects that are most suitably organized as an aggregate. Derived types are also known as *user defined data types*. A derived type must first be defined. Once defined, instances of the derived type can be declared with unique variable names.

Derived type definition:

```
TYPE name
component_declarations
END TYPE [name]
```

where: *name* is the symbolic name that will be used to reference the type.

component_declarations declarations of the variables, arrays, and derived types that comprise the new type. They may have default initializations.

Examples:

```
TYPE DATE
  INTEGER Month
  INTEGER Day
  INTEGER Year
END TYPE DATE

TYPE POINT
  INTEGER X, Y
END TYPE POINT

TYPE TRIANGLE
  TYPE(POINT) A, B, C
END TYPE TRIANGLE
```

Once a derived type has been defined, it is available to declare variables of that type and to create values of that type. A derived type declaration takes a form similar to normal type statement:

```
TYPE(type_name) [attributes [::]] v [,v]... [=value]
```

where: *type_name* is the name of a previously defined type.

v is the symbolic name of a variable, an array, or an array declaration.

value is a type constant (see below).

Examples:

```
TYPE(DATE) :: DATEA, DATEB
```

```
TYPE(TRIANGLE), DIMENSION(10) :: T1, T2, T3
```

Structure Constuctor

A structure constructor is used to create a derived type constant value and takes the form of the type name and a comma separated list of expressions that define the components of the type. Examples:

```
TYPE(DATE) :: DATEC = DATE(5, 12, 2001)
TYPE(TRIANGLE) :: T1 = TRIANGLE(POINT(6,2), POINT(6,14), POINT(18,2))

DATEA = DATE(2, 16, 1977)
```

Component keywords can be used for identification, to change the order or to omit components. An omitted component must have a default initialization established when the type was declared. Examples:

```
TYPE DATE
  INTEGER :: Month = 1
  INTEGER :: Day   = 1
  INTEGER :  Year  = 1970
END TYPE DATE

TYPE(DATE) :: TheDate

TheDate = DATE(Month=12, Day=31, Year=2013)
TheDate = DATE(Year=2013, Month=12, Day=31)
TheDate = DATE(Year=2013)
```

In the last example, the two omitted components (`Month` and `Day`) are given the default initializations from the declaration.

Derived Type Component Reference

The components of a derived type are accessed with the variable name and the component separated by a percent sign (%):

```
TYPE(DATE) :: DateA
TYPE(TRIANGLE) :: T1

DateA%Month = 7
DateA%Day   = 12
DateA%Year  = 2002

T1%A%X = -12
T1%A%Y =   4
T1%B%X =  15
T1%B%Y =  22
T1%C%X =  15
T1%C%Y =   4
```

Normal array element syntax is used to access arrays of derived types:

```
TYPE(DATE), DIMENSION(5) :: Dates

IF (Dates(i)%Month == Dates(j)%Month .AND. &
    Dates(i)%Day   == Dates(j)%Day   .AND. &
    Dates(i)%Year  == Dates(j)%Year) THEN
```

The ability to define operators for manipulating derived types and their components is provided in Fortran. This is discussed on the Chapter **Modules**.

POINTERS

A pointer is an object that can be made to point to other objects. If the object that a pointer points to is not itself a pointer, it must have the `TARGET` attribute. The pointer assignment operator, `=>`, is used to *associate* a pointer to an object. It is important to distinguish the difference between pointer assignment and normal assignment.

```
REAL, TARGET  :: A, B
REAL, POINTER :: P

IF (I > 10) THEN
  P => A
ELSE
  P => B
END IF

P = 1.0/P
```

A powerful use of pointers is to create linked lists.

```
TYPE Style
  INTEGER index
  REAL value
  TYPE(Style), POINTER :: next
END TYPE Style

TYPE(Style), POINTER :: Head, Current, Next

ALLOCATE(Head)
Head = Style(1, 0.0, NULL())
```

allocate a linked list of 10 elements:

```
Current => Head
do i=2, 10
  ALLOCATE(Current%next)
  Current => Current%next
  Current = Style(i, 0.0, NULL())
end do
```

deallocate a linked list:

```
Current => Head
do
```

```
    if (.not. ASSOCIATED(Current)) exit
    Next => Current%next
    DEALLOCATE(Current)
    Current => Next
  end do
```

Pointer Aliases

A pointer can point to a commonly used array section to simplify a program.

```
INTEGER, DIMENSION(3,4), TARGET  :: a
INTEGER, DIMENSION(:,:), POINTER :: b

a = RESHAPE((/1,2,3,4,5,6,7,8,9,10,11,12/), (/3,4/))

b => a(1:3,2:3)
print *,b

end

4  5  6  7  8  9
```

Pointer Bounds Remapping

The bounds of a pointer object can be remapped with a bounds remapping list. A bounds remapping list takes the same form as a normal array specification. If a bounds remapping list it present it must be the same rank as the pointer target. A bounds remapping list may not specify an array larger than the pointer target.

```
REAL, DIMENSION(50,50), TARGET :: T
REAL, DIMENSION(:, :), POINTER :: P
P(0,0) => T
```

In above example, the dimensions of the pointer object, `P`, are `0:49`, `0:49`.

```
P(0:24,0:24) => T
```

In above example, the dimensions of the pointer object, `P`, are `0:24`, `0:24` and are the section of the pointer target, `T`, `1:25`, `1:25`.

Arrays of Pointers

Arrays of pointers are not allowed in Fortran, but the equivalent effect can be easily attained with a type containing a pointer element.

```
TYPE POINTER_ARRAY
  REAL, POINTER, DIMENSION(:) :: A
END TYPE POINTER_ARRAY

TYPE(POINTER_ARRAY) :: DIMENSION(10) :: S

DO I=1,10
  ALLOCATE(S(I)%A(100))
END DO
```

ASSOCIATED() Function

The logical function `ASSOCIATED` is used to determine if a pointer is associated with a target. It is also used to determine if a pointer is associated with a specific target.

```
ASSOCIATED(pointer[, target])
```

NULL() Function

The `NULL` function is used to give a pointer a disassociated status.

```
REAL, POINTER :: P => NULL()

P => NULL()
```

NULLIFY Statement

The `NULLIFY` statement is used to disassociate a pointer.

```
REAL, POINTER :: P
…
NULLIFY(P)
```

is equivalent to:

```
P => NULL()
```

ALLOCATABLE STATEMENT

The `ALLOCATABLE` statement specifies the `ALLOCATABLE` attribute for a list of variable names:

```
ALLOCATABLE [::] var[(def-spec)] [,var[(def-spec)]]…
```

where *var* is the symbolic name of a variable

def-spec is a deferred shape specification if the `DIMENSION` attribute had been previously specified for the variable

ASYNCHRONOUS STATEMENT

The `ASYNCHRONOUS` statement specifies the `ASYNCHRONOUS` attribute for a list of variable names:

```
ASYNCHRONOUS [::] var [,var]…
```

where *var* is the symbolic name of a variable

AUTOMATIC STATEMENT

The `AUTOMATIC` statement specifies the `AUTOMATIC` attribute for a list of variable names:

```
AUTOMATIC [::] var [,var]…
```

where *var* is the symbolic name of a variable

DIMENSION STATEMENT

The `DIMENSION` statement declares the names and supplies the dimension information for arrays to be used within a program unit.

```
DIMENSION [::] a(d) [,a(d)]...
```

where *a(d)* is an array declarator as described in the chapter **Arrays**.

Arrays may be declared with either `DIMENSION` statements, `COMMON` statements, or type statements, but multiple declarations are not allowed. That is, once a symbolic name has been declared to be an array it may not appear in any other declaration statement with an array declarator in the same program unit. The following three statements declare the same array:

```
DIMENSION A(5,5,5)
REAL A(5,5,5)
```

```
REAL(KIND(1.0e0), DIMENSION(5,5,5) :: A
REAL(KIND(1.0e0) :: A(5,5,5)
COMMON A(5,5,5)
```

COMMON STATEMENT

The COMMON statement is used to declare the storage order of variables and arrays in a consistent and predictable manner. This is done through a FORTRAN data structure called a common block, which is a contiguous block of storage. A common block may be identified by a symbolic name but does not have a data type. Once the order of storage in a common block has been established, any program unit that declares the same common block can reference the data stored there without having to pass symbol names through argument lists.

The GLOBAL statement may be used to make the common block accessible to other tasks on systems which support shared data.

Common blocks are specified in the following manner:

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

where: *cb* is the symbolic name of the common block. If *cb* is omitted, the first pair of slashes may also be omitted.

nlist contains the symbolic names of variables, arrays, and array declarators.

When the COMMON block name is omitted, the COMMON block is called blank COMMON. The symbolic name "BLANK" is reserved by the compiler for blank COMMON and if used explicitly as a COMMON block name will result in all entities in the *nlist* being placed in blank COMMON.

Any COMMON block name or an omitted name (blank COMMON) can occur more than once in the COMMON statements in a program unit. The list of variables and arrays following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

A COMMON block name can be the same as that of a variable, array, or program unit.

CONTAINS STATEMENT

The CONTAINS statement marks the section of program unit (MAIN, SUBROUTINE, FUNCTION, or MODULE) that contains internal procedures. See the Chapter **PROGRAMS, SUBROUTINES, and FUNCTIONS**. Example

```
CONTAINS
```

EQUIVALENCE STATEMENT

The `EQUIVALENCE` statement provides a means for one or more variables to share the same storage location. Variables that share storage in this manner are said to be associated. The association may be total if both variables are the same size or partial if they are of different sizes. The `EQUIVALENCE` statement is used in the following manner:

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

The symbolic names of at least two variables, arrays, array elements, or character substrings must be specified in each *nlist*. Only integer constant expressions may be used in subscript and substring expressions. An array name unqualified by a subscript implies the first element of the array.

An `EQUIVALENCE` statement causes storage for all items in an individual *nlist* to be allocated at the same starting location:

```
REAL A,B
INTEGER I,J
EQUIVALENCE (A,B), (I,J)
```

The variables `A` and `B` share the same storage location and are totally associated. The variables `I` and `J` share the same storage location and are totally associated.

Items that are equivalenced can be of different data types and have different lengths. When a storage association is established in this manner several elements of one data type may occupy the same storage as one element of a different data type:

```
DOUBLE PRECISION D
INTEGER I(2)
EQUIVALENCE (D,I)
```

The array element `I(1)` shares the same storage location as the upper (most significant) thirty-two bits of `D`, and the array element `I(2)` shares the same storage location as the lower (least significant) thirty-two bits of `D`. Because only a portion of `D` is stored in the same location as `I(1)`, these entities are only partially associated.

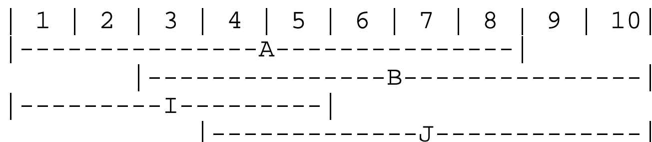
The `EQUIVALENCE` may not specify that an item occupy more than one storage location or that a gap occur between consecutive array elements.

Equivalence of Arrays

The EQUIVALENCE statement can be used to cause the storage locations of arrays to become either partially or totally associated.

```
REAL A(8),B(8)
INTEGER I(5),J(7)
EQUIVALENCE (A(3),B(1)), (A(1),I(1)), (I(4),J(1))
```

Storage would be allocated as follows:

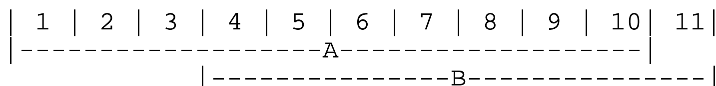


Equivalence of Substrings

The EQUIVALENCE statement can be used to cause the storage locations of substrings to become either partially or totally associated.

```
CHARACTER A(2)*5
CHARACTER B*8
EQUIVALENCE (A(2)(2:4),B(4:7))
```

Byte storage would be allocated as follows:



Notice that the lengths of the equivalenced substrings need not be the same, as in the above example.

COMMON and EQUIVALENCE Restrictions

The EQUIVALENCE statement can be used to increase the size of a common block by adding storage to the end, but it cannot increase the size by adding storage units prior to the first item in a COMMON block.

The EQUIVALENCE statement must not cause two different COMMON blocks to have their storage associated.

ENUMERATIONS

An enumeration is a construct for creating and generating named integer constants. The constants are identical to named integer constants created with the PARAMETER attribute or statement.

```
ENUM, BIND(C)
ENUMERATOR [::] enum-spec[, enum-spec]…
END ENUM
```

where: *enum-spec* is the symbolic name used to reference the constant.

enum-spec may be given a value with an equals sign and a scalar integer constant. If any *enum-spec* is given a value, the double colon separator must appear. The default initial value is 0. A subsequent *enum-spec* without a value is incremented by 1 from the previous *enum-spec*.

example:

```
ENUM, BIND(C)
ENUMERATOR :: STEP1=1, STEP2, STEP3
ENUMERATOR STEP4
END ENUM
```

Equivalent INTEGER declaration with a PARAMETER attribute:

```
INTEGER, PARAMETER :: STEP1=1, STEP2=2, STEP3=3, STEP4=4
```

EXTERNAL STATEMENT

The EXTERNAL statement allows symbolic names to be used as arguments in CALL statements and function references without the compiler automatically creating a variable at the point of reference. Symbolic names so declared may or may not have an associated data type. The EXTERNAL statement is given with a list of external or dummy procedure names or intrinsic function names:

```
EXTERNAL proc [,proc]...
```

where: *proc* is a symbolic name of a procedure or intrinsic function.

An intrinsic function name appearing in an EXTERNAL statement specifies that the particular intrinsic function has been replaced by a user supplied routine.

IMPLICIT STATEMENT

The IMPLICIT statement is used to establish implicit data typing that differs from the default INTEGER and REAL typing described in chapter **The Fortran Program**. The

IMPLICIT statement can also be used to remove implied typing altogether. The IMPLICIT statement takes the following form:

```
IMPLICIT type [*len] (a [,a]...) [,type [*len] (a [,a]...)]...
```

where: *type* is a type chosen from the set CHARACTER, LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, or NONE.

len is an unsigned integer constant specifying the length, in bytes, of LOGICAL, INTEGER, REAL, COMPLEX or CHARACTER variables.

a is an alphabetic specifier which is either a single letter or a range of letters. A range of letters is specified with a character representing the lower bound of the range, a minus, and a character representing the upper bound of the range. The range A-z specifies all the letters of the alphabet.

If *len* is not specified, the defaults are:

CHARACTER	1 byte
LOGICAL	4 bytes
INTEGER	4 bytes
REAL	4 bytes
COMPLEX	8 bytes

The IMPLICIT statement must appear before all other declaration statements except PARAMETER statements and specifies the data type of all symbolic names that can take a data type that are not given one explicitly with a type statement. The data type will be the data type that corresponds to the first character of their names.

When NONE appears in place of a type specifier, all variables used within the program unit must appear in an explicit type statement.

INTENT STATEMENT

The INTENT statement specifies the INTENT attribute for a list of dummy arguments. It is used to inform the compiler how dummy arguments are intended to be used. INTENT(IN), INTENT(OUT), and INTENT(INOUT) are available.

```
INTENT(spec) [::] var [,var…]
```

where *spec* is IN, OUT, or INOUT.

var is the symbolic name of a function or subroutine dummy argument.

```
SUBROUTINE SOLVER(TD)
  IMPLICIT NONE
```

```
    REAL, INTENT(IN) :: TD
    ...
  END SUBROUTINE SOLVER
```

INTRINSIC STATEMENT

The `INTRINSIC` statement designates symbolic names as intrinsic functions (see the chapter **Programs, Subroutines, and Functions**). Similar to the `EXTERNAL` statement, this allows intrinsic functions to be used as arguments in `CALL` statements and function references without the compiler automatically creating a variable at the point of reference. The intrinsic function name specified in an `INTRINSIC` statement retains its associated data type. The `INTRINSIC` statement is given in the following manner:

```
INTRINSIC func [,func]...
```

where: *func* is the name of an intrinsic function.

The following intrinsic functions do not follow conventional FORTRAN calling conventions and may not appear in an `INTRINSIC` statement:

```
AMAX0        LEN
AMAX1        LGE
AMIN0        LGT
AMIN1        LLE
CHAR         LLT
CMPLX        MAX
DBLE         MAX0
DCMPLX       MAX1
DMAX1        MIN
FLOAT        MIN0
ICHAR        MIN1
IDINT        REAL
IFIX         SNGL
INT

DFLOATI      IIFIX
DFLOATJ      IINT
FLOATI       JIDINT
FLOATJ       JIFIX
HFIX         JINT
IIDINT
```

This restriction also applies to the Absoft Fortran 77 intrinsic functions below:

```
ADJUSTL      VAL2
ADJUSTR      VAL4
TRIM         [%]LOC
REPEAT       [%]VAL
```

The `INTRINSIC` statement is used to pass intrinsic functions to external procedures:

```
INTRINSIC SIN,COS
DIMENSION A(100),B(100)
CALL TRIG(SIN,A)
CALL TRIG(COS,B)
```

```
      END

      SUBROUTINE TRIG(FUNC,ARRAY)
      DIMENSION ARRAY(100)
      DO 10 I=1,100
10    ARRAY(I) = FUNC(FLOAT(I))
      END
```

NAMELIST STATEMENT

The NAMELIST statement associates a unique group name with a list of variable or array names. The group-name is used in namelist directed input and output operations. Namelists are specified in the following manner:

NAMELIST /*group_name*/*nlist*[[,]/*group_name*/*nlist*]...

where: *group_name* is a symbolic name.

nlist is a list of variables or array names that is to be associated with *group_name*

The *group_name* must be a unique identifier and cannot be used for any purpose other than namelist directed I/O in the program unit in which it appears.

Variables in a namelist can be of any data type and can be explicitly or implicitly typed. Subscripted array names and CHARACTER substrings cannot appear in the NAMELIST, however NAMELIST directed I/O can be used to assign values to array elements and CHARACTER substrings.

The order of appearance of variables in a NAMELIST controls the order in which the values are written during NAMELIST directed output. The order of appearance has no affect on NAMELIST directed input.

Adjustable arrays are not permitted in NAMELIST statements.

OPTIONAL STATEMENT

The OPTIONAL statement specifies the OPTIONAL attribute for a list of dummy arguments:

OPTIONAL [::] *var* [,*var*…]

where *var* is the symbolic name of a function or subroutine dummy argument.

```
INTERFACE
  REAL FUNCTION Accumulate(A, B, C)
    REAL :: A
    REAL, OPTIONAL :: B, C
  END FUNCTION Accumulate
END INTERFACE
```

```
D = Accumulate(X,Y,Z)
D = Accumulate(X,B=Y)
D = Accumulate(X,C=Z)
```

The `PRESENT` function is used to determine if an *optional* dummy argument is present.

```
REAL FUNCTION Accumulate(A, B, C)
  REAL :: A
  REAL, OPTIONAL :: B, C
  Accumulate = A
  IF (PRESENT(B)) Accumulate = Accumulate + B
  IF (PRESENT(C)) Accumulate = Accumulate + C
END FUNCTION Accumulate
```

PARAMETER STATEMENT

The `PARAMETER` statement allows a constant to be given a symbolic name in the following manner:

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

where *p* is the symbolic name that is used to reference the constant and *c* is a constant expression.

If the data type and length attributes of the symbolic name are to be other than the implied default for that name, then the type (and size) must be previously defined in an explicit type statement or through the typing supplied by an `IMPLICIT` statement. A character parameter may have its length declared as an asterisk in parentheses, in which case the actual size will be the number of characters in the expression.

The type of the constant expression must match the type of the symbolic name.

```
INTEGER EOF
CHARACTER TITLE*(*)
PARAMETER (PI=3.1415926, THIRD=1.0/3.0)
PARAMETER (EOF=-1)
PARAMETER (TITLE='FORTRAN 77')
```

POINTER STATEMENT

The `POINTER` statement specifies the `POINTER` attribute for a list of variable names:

```
POINTER [::] var[(def-spec)] [,var[(def-spec)]]…
```

where *var* is the symbolic name of a variable

def-spec is a deferred shape specification if the `DIMENSION` attribute had been previously specified for the variable

POINTER STATEMENT (CRAY-STYLE)

The `POINTER` statement is an extension to standard Fortran and is provided for compatibility with older implementations of Fortran. This type of pointer is commonly known as a “Cray pointer”. It is recommended that new programs use the standard pointer mechanism described earlier in this chapter.

The `POINTER` statement is used to establish a means for directly manipulating the address of a variable. Normally, when a FORTRAN variable is created, either explicitly with a declaration statement of implicitly by reference in the program, its location or address in memory is fixed by the compiler or the linker. However, there are situations where it is useful for the location of a variable to be dynamic. The `POINTER` statement provides this mechanism by associating a pointer to a variable as follows:

```
POINTER (ptr,pbv) [, (ptr, pbv),...]
```

where: *ptr* is the symbolic name that is used to manipulate the address of the associated variable and *pbv* is the pointer-based variable.

Before the pointer-based variable can be used, the pointer must be defined with the initial address of the variable. The `LOC` function is useful for this purpose:

```
INTEGER m(100), a
POINTER (pa, a)

pa = LOC(m)
DO i=1, 100
      a = i
      pa = pa + 4
END DO
```

The array will contain the integers from 1 to 100 after the execution of the `DO` loop.

When a pointer-based variable is referenced, its address is established from the current value of the associated pointer variable. Pointers can be used just as if they were `INTEGER` variables except that they cannot appear as dummy arguments. Pointer-based variables can be of any FORTRAN data type. Pointer-based variables cannot be dummy arguments or appear in `COMMON`, `GLOBAL`, `EQUIVALENCE`, `NAMELIST`, `SAVE`, `VALUE` or `PARAMETER` statements. Further information and examples are presented in the appendices.

PRIVATE AND PUBLIC STATEMENTS

Modules allow specifications, data, and procedures to be encapsulated. The default is that all data entities are visible to procedures that use the module; they are `PUBLIC`. Often it is useful to limit the accessibility of symbolic entities to the module itself. For example, only the generic interface to a generic function is typically visible outside of the module. The specific interfaces and function implementations are *private* to the module. A `PRIVATE` statement can be to used limit the scope to only the module.

```
REAL, PARAMETER, PRIVATE :: PI=3.1415926535

PRIVATE :: SP_LOGB, DP_LOGB

TYPE SEMAPHORE
  PRIVATE
  REAL, DIMENSION(), POINTER :: ACCESS
  LOGICAL STATE
END TYPE SEMAPHORE
```

PROCEDURE STATEMENT

The `PROCEDURE` statement declares external procedures, dummy procedures, and procedure pointers:

```
PROCEDURE([proc-inter]) [[,proc-attr] ::] proc-list
```

where *proc-inter* is the name of an interface or a type name.

proc-attr is comma separated list of any of the following:

```
PUBLIC
PRIVATE
INTENT(INOUT)
OPTIONAL
POINTER
SAVE
```

proc-list is a list of procedure names. The procedure pointer name may be initialized with the `NULL()` function.

The following two statements are equivalent:

```
PROCEDURE(INTEGER) IFUN
INTEGER, EXTERNAL :: IFUN
```

A typical use of the `PROCEDURE` statement is shown below:

```
INTERFACE
  FUNCTION CONSTRUCT(X,Y)
  REAL, INTENT(IN) :: X, Y
  REAL :: CONSTRUCT
  END FUNCTION CONSTRUCT
END INTERFACE

PROCEDURE(CONSTRUCT), POINTER :: SYSCON => NULL()
```

RECORD STATEMENT

RECORD statements are used to define variables that are instances of structures defined in STRUCTURE declarations. Variables declared in RECORD statements are composite or aggregate data items. RECORD statements are defined as follows:

```
RECORD /structure-name/rlist[,/structure-name/rlist...]
```

where: *structure-name* is a name given in a STRUCTURE definition.

rlist is one or more variable names, array names or array declarators separated by commas.

Record names cannot appear in NAMELIST statements and can only be read from and written to UNFORMATTED files. For more information on the use of records, see the appendices.

SAVE STATEMENT

Fortran permits dynamic as well as static storage allocation. Variables, arrays, and COMMON blocks declared in a main program are allocated statically and always retain their definition status. Variables and arrays that are declared locally in subprograms are allocated dynamically when the subprogram is invoked. When the subprogram executes a RETURN or END statement, these items are deallocated and lose their definition status. The SAVE statement is used to retain the definition status of these items by specifying them in the following manner:

```
SAVE [[::] a [,a]...]
```

where: *a* is either a common block name delimited with slashes, a variable name, or an array name.

If *a* is not specified, the effect is as though all items in the program unit were presented in the list.

In order to maintain portability in FORTRAN programs, it is recommended that the SAVE statement be used on COMMON blocks declared only in subprograms. Although it has no effect in this implementation, other FORTRAN compilers may cause deallocation of common blocks upon returning from a subprogram if they are not SAVEd.

SEQUENCE STATEMENT

The storage sequence of members of a derived type is not specified. For optimization purposes, a compiler may rearrange them. The SEQUENCE statement is used specify the order of the derived type members and to assure adjacency.

```
TYPE PIXEL
  SEQUENCE
  INTEGER(SELECTED_INT_KIND(4)) :: XPos, YPos
  INTEGER(SELECTED_INT_KIND(9)) :: RVal, GVal, BVal
END TYPE PIXEL
```

STDCALL STATEMENT

The STDCALL statement indicates that a function or subroutine is to use the Windows 32-bit STDCALL call/return protocol

STDCALL *proc* [,*proc*]…

where *proc* is the symbolic name of a subroutine or function

NOTE: This statement is applicible only to programs intended for execution on a Windows 32-bit system only. The statement may appear in Windows 64-bit programs, but has no effect.

STRUCTURE DECLARATION

The STRUCTURE statement is an extension to standard Fortran and is provided for compatibility with older implementations of Fortran. It is recommended that new programs use the standard derived type mechanism described earlier in this chapter.

The STRUCTURE declaration provides a mechanism for organizing various data items into meaningful groups forming an aggregate data type called a structure. The STRUCTURE declaration establishes the data types, ordering, and alignment of a structure, but may not actually define storage for a structure. Storage for the actual structure may be defined in the STRUCTURE declaration or with a RECORD statement (see above section **RECORD Statement**). A STRUCTURE declaration takes the following form:

```
STRUCTURE [/structure-name/][rlist [,rlist...]]
      field-declaration
      [field-declaration]
      .
      .
      .
      [field-declaration]
END STRUCTURE
```

where: *structure-name* is the name used to identify the structure declaration.

rlist is a list of symbolic names or array declarators allocating records having the form defined in the STRUCTURE declaration.

field-declaration defines a data item in the structure.

The *structure-name* must be unique among STRUCTURE declarations, but can be used for variable and global names.

A *field-declaration* can be any FORTRAN type statement, a POINTER declaration, a UNION declaration, a RECORD statement, or another STRUCTURE declaration. Arithmetic and logical type statements may take an optional /value/ specifier to provide initialization for the data item in each instance of the structure as described earlier in this chapter.

The name %FILL can be used in place of the symbol name in a *field-declaration*. (e.g INTEGER*2 %FILL). In this case, no field name is defined, but empty space is added to the structure for the purpose of alignment. This extension is provided for compatibility with other FORTRAN environments.

If the *field-declaration* is another STRUCTURE declaration, the *structure-name* may be omitted, but the *rlist* must be given. In this case the symbolic names in *rlist* become fields of the structure which contains it. If the *structure-name* is given, it can be used in RECORD statements to define instances of the substructure.

The structure fields established with field-declarations are accessed by appending a period and the field name to the record name:

```
STRUCTURE /date/ day
      INTEGER mm
      INTEGER dd
      INTEGER yy
END STRUCUTRE

day.mm = 9
day.dd = 12
day.yy = 90
```

See the appendix **Using Structures and Pointers** for examples using STRUCTURE declarations.

UNION DECLARATION

A `UNION` declaration defines a data area which is shared by one or more fields or groups of fields. It begins with a `UNION` statement and ends with an `ENDUNION` statement. In between are `MAP` and `ENDMAP` statements which define the field or groups of fields which will share the storage area. A `UNION` declaration is as follows:

```
UNION
      map-declaration
      map-declaration
      [map-declaration]
      .
      .
      .
      [map-declaration]
END UNION
```

where: *map-declaration* takes the following form:

```
MAP
      field-declaration
      [field-declaration]
      .
      .
      .
      [field-declaration]
END MAP
```

A *field-declaration* contains one or more of the following, a `STRUCTURE` declaration, a `POINTER` declaration, another `UNION` declaration, a `RECORD` statement or a standard FORTRAN type declaration. Field-declarations cannot have been previously declared or be dummy arguments.

The size of the shared data area for the union is the size of the largest map area contained within the union. The fields of only one of the map areas are defined at any given time during program execution.

Example:

```
UNION
      MAP
            INTEGER*4 long
      END MAP
      MAP
            INTEGER*2 short1, short2
      END MAP
END UNION
```

In the above example, the storage for the first half of the field `long` is shared by the field `short1`, and the storage for the second half of the field `long` is shared by the field `short2`.

TARGET STATEMENT

The TARGET statement specifies the TARGET attribute for a list of variable names:

```
TARGET [::] var[(array-decl)] [,var[(array-decl)]]…
```

where *var* is the symbolic name of a variable

array-decl is an array declaration.

VALUE STATEMENT

The VALUE statement informs the compiler that certain dummy arguments are going to be passed by value to a subroutine or function. When a value parameter is passed, the contents of a variable rather than its address is passed. The result is that the actual argument cannot be changed by the program unit. Pass by value is the default method for C and Pascal programs and is used when the VAL intrinsic function is used in Absoft Fortran 77.

```
VALUE [::] a [[,a]...]
```

where: *a* is the name of a dummy argument.

Value arguments can be of any data type except CHARACTER. Value arguments cannot be arrays, but they can be of type RECORD. VALUE statements cannot appear in program units which contain ENTRY statements.

VOLATILE STATEMENT

The VOLATILE statement disables optimization for any symbol, array, or COMMON block. It is useful when a variable is actually an absolute address, when two dummy arguments can represent the same location, or when a POINTER variable points to defined storage.

```
VOLATILE [::] a [[,a]...]
```

where: *a* is either a common block name delimited with slashes, a variable name, or an array name.

DATA STATEMENT

Variables, substrings, arrays, and array elements are given initial values with DATA statements. DATA statements may appear only after the declaration statements in the program unit in which they appear. DATA statements take the following form:

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

where: *vlist* contains the symbolic names of variables, arrays, array elements, substrings, and implied DO lists

clist contains the constants which will be used to provide the initial values for the items in *vlist*

A constant may be specified in *clist* with an optional repeat specifier: a positive integer constant (or symbolic name of a constant) followed by an asterisk. The repeat specifier is used to indicate one or more occurrences of the same constant:

```
DATA A,B,C,D,E/1.0,1.0,1.0,1.0,1.0/
```

can be written as:

```
DATA A,B,C,D,E/5*1.0/
```

An array name unqualified by a subscript implies every element in the array:

```
INTEGER M(5)
DATA M/5*0/
```

means:

```
INTEGER M(5)
DATA M(1),M(2),M(3),M(4),M(5)/0,0,0,0,0/
```

Type conversion is automatically performed for arithmetic constants (INTEGER, REAL, DOUBLE PRECISION, and COMPLEX) when the data type of the corresponding item in *vlist* is different. CHARACTER constants are either truncated or padded with spaces when the length of the corresponding character item in *vlist* is either shorter or longer than the constant respectively.

The items specified in *vlist* may not be dummy arguments, functions, or items in blank COMMON. Items in a named common block can be initialized only within a BLOCK DATA subprogram (see the chapter **Programs, Subroutines, and Functions**).

Implied DO List In A DATA Statement

An implied `DO` list is used to initialize array elements as though the assignments were within a `DO` loop. The implied `DO` list is of the form:

```
(dlist, i = m1, m2 [,m3])
```

where: *`dlist`* contains array elements and implied `DO` lists

`i` is the `DO` variable and must be an integer

`m1`, *`m2`*, and *`m3`* are integer constant expressions which establish the initial value, limit value, and increment value respectively (see the **Control Statements** chapter)

```
INTEGER M(10,10),N(10),L(4)
CHARACTER*3 S(5)

DATA (N(I),I=1,10),((M(I,J),J=3,8),I=3,8)/5*1,5*2,36*99/
DATA (L(I),I=1,4)/'ABCD','EFGH','IJKL','MNOP'/
DATA (S(I),I=1,5)/'ABC','DEF','GHI','JKL','MNO'/
```

CHAPTER 4

Arrays

ARRAYS

An array is compound data object that consists of a sequence of data elements all of the same type and referenced by one symbolic name. When an array name is used alone it refers to the entire sequence starting with the first element. When an array name is qualified by a subscript it refers to an individual element of the sequence. Arrays can statically declared or dynamically declared (allocatable).

ARRAY DECLARATOR

An array declarator is used to assign a symbolic name to an array, define its data type (either implicitly or explicitly), and declare its dimension information:

```
a(d [,d]...)
```

where *a* is the symbolic name that will be used to reference the array and the elements of the array, and *d* is called a dimension declarator. An array declarator must contain at least one and no more than seven dimension declarators. A dimension declarator is given with either one or two arguments:

```
[d1:] d2
```

where *d1* and *d2* are called the lower and upper dimension bounds respectively. The lower and upper dimension bounds must be expressions containing only constants or integer variables. Integer variables are used only to define adjustable and automatic arrays (described below) in subroutine and function subprograms. If the lower dimension bound is not specified, it has a default value of one.

An array declarator specifies the *rank* and *shape* of an array. The number of dimensions is determined by the number of dimension declarators and called its *rank*. Dimension bounds specify the size or extent of an individual dimension. The sequence of dimension extents of an array is called its *shape*.

```
REAL A(10,10,10)              ! rank 3, shape (10,10,10)
REAL B(2,12,8,8)              ! rank 4, shape (2,12,8,8)
REAL C(-10:-1,2:11,-2:7)      ! rank 3, shape (10,10,10)
```

While the value of a dimension bound may be positive, negative, or even zero, the value of the lower dimension bound must always be less than or equal to the value of the upper

dimension bound. The extent of each dimension is defined as `d2-d1+1`. The number of elements in an array is equal to the product of all of its dimension extents.

Array declarators are called constant, adjustable, automatic, deferred shape, or assumed shape depending on the form of the dimension bounds. A constant array declarator must have integer constant expressions for all dimension bounds. An adjustable array declarator contains one or more integer variables in the expressions used for its bounds. An automatic array is an explicit shape array in a subprogram that takes it dimension bounds and hence its shape in non-constant expressions.

An array declarator in which the upper bound of the last dimension is an asterisk (*) is an assumed size array declarator. Adjustable and assumed size array declarators may appear only in subroutine and function subprograms.

All array declarators are permitted in `DIMENSION` and type statements, however only constant array declarators are allowed in `COMMON` or `SAVE` statements.

An array can be either an actual array or a dummy array. An actual array uses constant array declarators and has storage established for it in the program unit in which it is declared. A dummy array may use constant, adjustable, or assumed size array declarators and declares an array that is associated through a subroutine or function subprogram dummy argument list with an actual array.

The number of dimensions and the dimension extents of arrays associated with one another either through common blocks, equivalences, or dummy argument lists need not match.

ARRAY SUBSCRIPT

The individual elements of an array are referenced by qualifying the array name with a subscript:

```
a(s [,s]...)
```

where each *s* in the subscript is called a subscript expression and *a* is the symbolic name of the array. The subscript expressions are integer scalar expressions whose values fall between the lower and upper bounds of the corresponding dimension. There must be a subscript expression for each declared dimension.

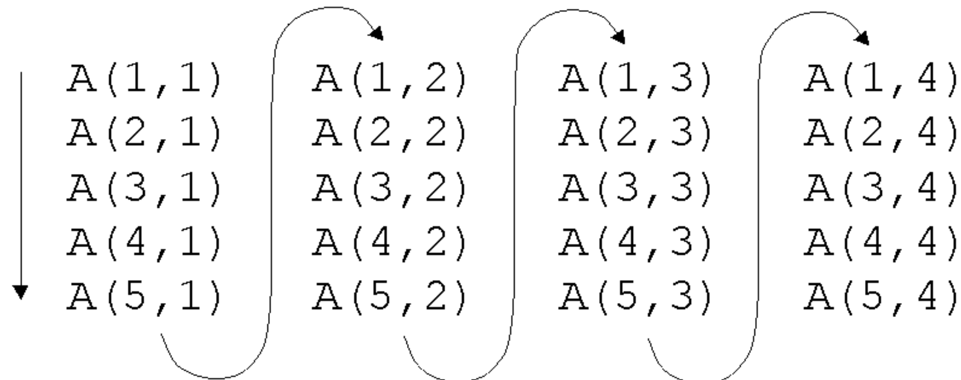
Subscript expressions may contain array element and function references. The evaluation of a subscript expression must not affect the value of any other expression in the subscript. This means that functions should not have side effects altering the values of the other subscript expressions.

Note the following example where `A` is a two-dimensional array and `F` is an external function.

```
Y = A(X,F(X))
```

The function `F(X)` will be evaluated before the value of `X` is fetched from memory. Therefore, if `F(X)` alters the value of `X`, the altered value will be used as the first subscript expression.

Arrays are stored in contiguous array element order. Fortran arrays are stored in *column major* order. The ordering of elements in the array A(5,4) is:



The order of an array element within the column major storage sequence of the array in memory is called the <u>subscript value</u>. This is calculated according to the following table:

Number of Dimensions	Dimension Declarator	Subscript	Subscript Value
1	(j1:k1)	(s1)	1+(s1-j1)
2	(j1:k1,j2:k2)	(s1,s2)	1+(s1-j1)+(s2-j2)*d1
3	(j1:k1,j2:k2,j3:k3)	(s1,s2,s3)	1+(s1-j1)+(s2-j2)*d1 +(s3-j3)*d2*d1
.	.	.	.
.	.	.	.
.	.	.	.
n	(j1:k1,…,jn:kn)	(s1,...,sn)	1+(s1-j1)+(s2-j2)*d1 +(s3-j3)*d2*d1+… +(sn-jn)*dn-1*dn-2 *…*d1 where: di = ki-ji+1

Subscript Value

Note that subscript values always range from 1 to the size of the array:

```
DIMENSION X(-4:4),Y(5,5)

X(3) = Y(2,4)
```

For the array element name `X(3)`, the subscript is `(3)`, the subscript expression is `3` with a value of three, and the subscript value is eight. For the array element name `Y(2,4)`, the subscript is `(2,4)`, the subscript expressions are `2` and `4` with values two and four, respectively, and the subscript value is seventeen. The effect of the assignment statement is to replace the eighth element of `X` with the seventeenth element of `Y`.

ARRAY VALUED EXPRESSIONS

Some FORTRAN constructs accept array names unqualified by a subscript. This means that every element in the array is selected. The elements are processed in column major order. The first element is specified with subscript expressions all equal to their lower dimension bounds. The next element will have the leftmost subscript expression increased by one. After an array subscript expression has been increased through its entire extent it is returned to the lower bound and the next subscript expression to the right is increased by one.

When an array name is used unqualified by a subscript, it implies that every element in the array is to be selected as described above. Generally speaking, any Fortran 90 expression may have array operands and the result is array valued. For example:

```
REAL, DIMENSION(200,300) :: A, B, C
A = 0.0
C = A+B
```

Array names unqualified by a subscript may also be used in `COMMON` statements for data alignment and sharing purposes, in actual and dummy argument lists to pass entire arrays to other procedures, in `EQUIVALENCE` statements where it implies the first element of the array, and in `DATA` statements for giving every element an initial value. Array names may also be used in the input and output statements to specify internal files, format specifications and elements of input and output lists.

ASSUMED SHAPE ARRAY

An assumed shape array is a subprogram dummy argument declaration where the dimension bounds are omitted. The array *assumes* its shape from the actual argument. Note that only the shape is assumed, not the lower and upper bounds.

```
REAL, DIMENSION(0:10,10:10) :: A          ! actual argument

REAL, DIMENSION(:,:) :: D                 ! dummy argument
```

The equivalent explicit shape declaration is:

```
REAL, DIMENSION(1:11,1:11) :: A           ! dummy argument
```

AUTOMATIC ARRAY

An automatic shape array is an explicit shape array in a subprogram that takes its dimension bounds and hence its shape from non-constant expressions. Such an array is local to the subprogram, allocated on the stack on entry, and deallocated on exit.

```
SUBROUTINE SWAP(A,B)
REAL, DIMENSION(:)       :: A,B           ! assumed shape arrays
REAL, DIMENSION(SIZE(A)) :: TEMP          ! automatic array
TEMP = A
A    = B
B    = TEMP
END SUBROUTINE SWAP
```

DEFERRED SHAPE ARRAY

A deferred shape or allocatable array is an array that has an `ALLOCATABLE` attribute and a specified rank. However, the bounds are determined when storage is created for the array in an `ALLOCATE` statement. Allocatable arrays are freed with the `DEALLOCATE` statement,

```
REAL, DIMENSION(:,:), ALLOCATABLE :: A
…
ALLOCATE(A(2:12),(4:14))
…
DEALLOCATE(A)
```

The `ALLOCATE` and `DEALLOCATE` statements may take the optional specifer `STAT=`*`ierr`*. *`ierr`* is an `INTEGER` variable that is defined with a 0 if the allocation succeeds. If the allocation fails, *`ierr`* is defined with one of the following integer values:

10045	attempt to `DEALLOCATE` unallocated array
10046	attempt to `DEALLOCATE` item with modified size
10047	attempt to `ALLOCATE` array which has already been allocated
10070	attempt to `ALLOCATE` of requested size failed

```
ALLOCATE(A(2:12),(4:14), STAT=ierr)
```

`The syntax of the ALLOCATE statement is:`

`ALLOCATE (`*`object_list`*` [STAT=`*`ierr`*`])`

where *`object_list`* is a comma separated list of an array declarators.

`ierr` is a scalar integer variable.

`The syntax of the DEALLOCATE statement is:`

`DEALLOCATE (`*`object_list`*` [STAT=`*`ierr`*`])`

where *`object_list`* is a comma separated list of allocatable array names.

ierr is a scalar integer variable.

POINTER ARRAYS

Pointer arrays are similar to allocatable arrays in that they are explicitly allocated with the ALLOCATE statement to have computed sizes and are explicitly deallocated with the DEALLOCATE statement. Simple examples of pointer arrays result by replacing ALLOCATABLE with POINTER in the preceding examples of allocatable arrays.

```
REAL, DIMENSION(:,:), POINTER :: A
...
IF (.NOT. ASSOCIATED(A)) ALLOCATE(A(2:12,4:14), STAT=IERR)
...
DEALLOCATE(A, STAT=IERR)
```

In addition, pointer arrays can be used as *aliases* for (may *point to*) other arrays and array sections; the pointer assignment statement is used to establish such aliases. The target for pointer associations (as such aliasing is called) may be other explicitly allocated arrays, or static or automatic arrays that have been explicitly identified as allowable targets for pointers. The association status of a pointer array may be tested with the ASSOCIATED intrinsic function. Pointer arrays may be dummy arguments and structure components, neither of which are allowed for allocatable arrays.

Given this apparent similarity between allocatable arrays and pointer arrays, what is the fundamental distinction between these two forms of dynamic arrays, and when should allocatable arrays be used rather than pointer arrays? Pointer arrays subsume all of the functionality of allocatable arrays, and in this sense allocatable arrays are never needed - pointer arrays could always suffice. The problem with pointer arrays is efficiency. Though pointer arrays must always point to explicit *targets*, which makes some optimization practical that would otherwise be infeasible, pointer assignment makes optimization of pointer arrays much more difficult than for allocatable arrays. Because of their more limited nature and functionality, allocatable arrays are just “simpler” and can be expected to be more efficient than pointer arrays.

Therefore, when all that is needed is simple dynamic allocation and deallocation of arrays, and automatic arrays are not sufficient, use allocatable arrays. A common example of this is if a “work array” is needed of a size dependent upon the results of a local computation. If, on the other hand, the algorithm calls for a dynamic alias, of for example a “moving” section of a host array, then a pointer array is probably indicated.

ARRAY SECTIONS

Array section subscript syntax, called a *subscript triplet*, provides a means to access a subarray:

[*lower-bound*]:[*upper-bound*]:[*stride*]

Any element of the triplet can be omitted. If *lower-bound* is omitted, the default is the actual lower bound. If *upper-bound* is omitted, the default is the actual upper bound. If *stride* is omitted, the default is 1. Examples:

```
A(I, J:N)              ! elements J-N of row I
A(I, :)                ! all elements of row I
A(I, : 1:N:3)          ! elements 1, 4, …, through N of row I
```

Array Sections

`INTEGER, DIMENSION(5,5) :: m`

m(3,4)

1	6	11	16	21
2	7	12	17	22
3	8	13	**18**	23
4	9	14	19	24
5	10	15	20	25

m(3,3:5)

1	6	11	16	21
2	7	12	17	22
3	8	**13**	**18**	**23**
4	9	14	19	24
5	10	15	20	25

m(:,3)

1	6	**11**	16	21
2	7	**12**	17	22
3	8	**13**	18	23
4	9	**14**	19	24
5	10	**15**	20	25

m(1::2,2:4)

1	**6**	**11**	**16**	21
2	7	12	17	22
3	**8**	**13**	**18**	23
4	9	14	19	24
5	**10**	**15**	**20**	25

ARRAY CONSTRUCTOR

Array constructors are used to create general rank-one arrays (vectors). Two equivalent syntaxes are available for creating a vector. The first consists of `(/`, the list of array elements, and `/)`. The second form is `[`, the list of array elements, and `]`. Either form may be used, but the opening and closing characters must match. The array elements can be constants, expressions, or implied DO lists.

```
(/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

or:

```
(/ (i, i=1,10) /)
 [ (i, i=1,10) ]
```

```
(/2, 4, 6, 8, 10, 12, 14, 16, 18, 20/)
 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

or:

```
(/ (i, i=2,20,2) /)
 [ (i, i=2,20,2) ]
```

```
(/1, 2, 3, 1, 2, 3, 1, 2, 3/)
 [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

or:

```
(/ ((i, i=1,3), j=1,3) /)
 [ ((i, i=1,3), j=1,3) ]
```

```
(/0, 1, 0, 1, 0, 1, 0, 1, 0, 1/)
 [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

or:

```
(/ (0,1, i=1,5) /)
 [ (0,1, i=1,5) ]
```

```
INTEGER, DIMENSION(10) :: M, N
M = (/ (0,1 i=1,5) /)
N =  [ (M+1) ]
```

VECTOR SUBSCRIPT

A form of section subscript syntax, call a vector subscript, provides access to individual array elements:

```
A( (/1, 8, 6, 5/) )
```

or:

```
A(  [1, 8, 6, 5]  )
```

Produces a section with the elements `A(1)`, `A(8)`, `A(6)`, and `A(5)`.

```
A( (/1, 8, 6, 5/) ) = (/-1, -1, -1, -1/)
```

Example:

```
REAL, DIMENSION(5,2) :: A
A(:,1) = (/1.,2.,3.,4.,5./)
A(:,2) =  [6.,7.,8.,9.,10.]
PRINT *, A((/1,3,5/),1)
PRINT *, A( [1,3,5], 2)
END
```

```
1.00000  3.00000  5.00000
6.00000  8.00000  10.0000
```

ARRAY CONFORMANCE

Two arrays of the same shape are said to be conformable. Array assignments can only performed when both the left and right hand sides are conformable. The `RESHAPE` function is available to change the shape of an array.

```
REAL, DIMENSION(2,3) :: A
REAL, DIMENSION(3,2) :: B

A = [1.,2.,3.,4.,5.,6]          ! produces an error
B = A                           ! produces an error

A = RESHAPE([1.,2.,3.,4.,5.,6.], [2,3])
B = RESHAPE(A, (/3,2/))
```

ARRAY OPERATIONS

Most operations that can be performed on scalar variables can be performed on arrays.

```
REAL, DIMENSION(2,3) :: A,B
```

$$A = \begin{bmatrix} 2 & 3 & 5 \\ 1 & 7 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 4 & 1 \\ 2 & 2 & 3 \end{bmatrix} \quad 2 = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

$$A+B = \begin{bmatrix} 7 & 7 & 6 \\ 3 & 9 & 7 \end{bmatrix} \quad A*B = \begin{bmatrix} 10 & 12 & 5 \\ 2 & 14 & 12 \end{bmatrix}$$

$$A+2 = \begin{bmatrix} 4 & 5 & 7 \\ 3 & 9 & 6 \end{bmatrix} \quad A*2 = \begin{bmatrix} 4 & 6 & 10 \\ 2 & 14 & 8 \end{bmatrix}$$

```
REAL, DIMENSION(2,3) :: A, B, C

A = 0.0
B = (B+1.0)/2.0
C = SQRT(B)
```

ARRAY VALUED FUNCTIONS

A function may return an array. However, whenever such a function is referenced it must always have an explicit interface.

```
FUNCTION VADD(a, r)
REAL, DIMENSION(:) :: a
REAL r
REAL, DIMENSION(SIZE(a)) :: VADD
VADD = a+COS(r)
RETURN
END
```

CHAPTER 5

Modules and Interfaces

A module is a Fortran program unit that encapsulates global declarations, global data, derived types and procedures. A module is not executable. It contains definitions to be used by other programs unit and these definitions may include procedures, but the module itself is not executable.

An interface provides an explicit prototype of a procedure.

MODULES

Module entities are made available to other program units by the `USE` statement:

```
program Seismic_Processing
use Seismic_Trace_Definitions
...
end program
```

Available modules entities are said to be "use associated" within the using program. A module may contain *private* entities; such entities are not available to using program units (private entities are available only within the module itself, including any procedure definitions within the module). In addition, the *only* form of the use statement limits the module entities that are available in the using program unit.

The principal contents of a module include: type definitions, interface definitions, procedure definitions, and shared data objects (including global constants). Typical uses of modules include: procedure libraries (with explicit interfaces), encapsulated data abstractions, and shared-data units (alternative to COMMON)

Module Structure

The general structure of a module is as follows; additional syntax rules are listed for specific items in the following description:

```
module module-name
  use-statements
  constant-definitions
  variable-declarations
  interface-blocks
  type-definitions
contains
  module-subprograms
end [module [module-name]]
```

Module Use

Other program units (main programs, functions, subroutines, and other modules) may use the definitions provided in a module by including a `USE` statement immediately after the program unit heading, as in the Seismic_Processing example above:

```
use module-name [rename-list]
use module-name, only: only-list
```

In the first example, all public entities of the module are imported. In the second example, only the specified public entities in the *only-list* are imported.

Module entities are imported into the using program with the same name as they have in the module, unless they are renamed in the `USE` statement; this may be necessary to avoid name conflicts, as an imported ("use associated") name does not "mask" a local entity with the same name. A *rename* has the form:

```
local-name => use-name
```

where *local-name* is the new name (in the using program) and *use-name* is the name of the entity in the module (the new name "points to" the module entity). An *only* can be either the name of the module entity being imported, or a rename:

```
[ local-name => ] use-name
```

PRIVATE And PUBLIC Statements

Modules allow specifications, data, and procedures to be encapsulated. The default is that all data entities are visible to procedures that use the module; they are `PUBLIC`. Often it is useful to limit the accessibility of symbolic entities to the module itself. For example, only the generic interface to a generic function is typically visible outside of the module. The specific interfaces and function implementations are *private* to the module. A `PRIVATE` statement can be to used limit the scope to only the module.

```
REAL, PARAMETER, PRIVATE :: PI=3.1415926535

PRIVATE :: SP_LOGB, DP_LOGB

TYPE SEMAPHORE
  PRIVATE
  REAL, DIMENSION(), POINTER :: ACCESS
  LOGICAL STATE
END TYPE SEMAPHORE
```

INTERFACES

A procedure interface comprises the information needed to use that procedure correctly; explicit interfaces make this information available to the calling environment. *Interface blocks are used to provide various explicit interfaces.* Explicit interfaces include dummy argument list characteristics, alternate names for a procedure (primarily used to define procedure overloads -

that is, generic procedures), and new operator and assignment definitions. Interface blocks are not needed to make module and internal procedure interfaces explicit, as these interfaces are automatically explicit wherever such procedures are accessible. However, external procedure interfaces are not automatically explicit; interface blocks with one or more interface bodies may be used to make them explicit:

```
interface
  interface-body
  [interface-body] ...
end interface
```

Each `interface-body` specifies an external (or dummy) procedure name, its type (if it is a function), and the order names, and types (and kinds) of all dummy arguments. If a function has certain properties it may be given an *operator interface*, thereby creating a *defined operator*, and called using operator notation; it must have one (unary operator) or two (binary operator) intent(in) arguments. Such a function may be called with either the normal function syntax or infix operation format; in the latter form the first actual argument appears as the first operation operand and the second actual argument is the second operand (for unary operators the operand follows the operator). The operator form of the interface block is used to define a new operator and associate it with one (or more) functions:

```
interface operator (defined-operator)
  [interface-body] ...
  [module-procedure-stmt] ...
end interface
```

Examples of using an operator interface to define an overloaded operators are:

```
interface operator (+)
  integer function sum_char_int(c, i)
    character :: c
    integer :: i
  end function
end interface
```

and:

```
MODULE DATES

TYPE DATE
  INTEGER Month, Day, Year
END TYPE DATE

INTERFACE OPERATOR(==)
  MODULE PROCEDURE DatesEqual
END INTERFACE

PRIVATE DatesEqual

CONTAINS

  LOGICAL FUNCTION DatesEqual(A, B)
    TYPE(DATE), INTENT(IN) :: A, B
    IF ((A%Month == B%Month) .AND. &
        (A%Day   == B%Day)   .AND. &
```

```
        (A%Year  == B%Year)) THEN
      DatesEqual = .TRUE.
    ELSE
      DatesEqual = .FALSE.
    ENDIF
  END FUNCTION DatesEqual

END MODULE DATES

PROGRAM Main

USE DATES

TYPE(DATE) :: A = DATE(2,10,2002)
TYPE(DATE) :: B = DATE(2,11,2002)

IF (A == B) PRINT *,"Equal"

END PROGRAM Main
```

A defined operator can be either a user-defined dot operator or (an overload of) an intrinsic operator, as in the preceding examples. If it is an intrinsic operator it must not redefine an intrinsic operation - for example, the + operator must not be given an operator interface for a function with two integer arguments, as that would be an attempt to redefine addition of two integer values. All operator definitions are considered to be generic procedure definitions and must be consistent with the generic reference resolution rules (see below). The function(s) associated with a defined operator may be either external functions (in which case the interface contains the corresponding interface bodies, as in the above example) or accessible module functions (in which case the interface contains the corresponding *module-procedure* statements).

An example of a user-defined dot operator:

```
MODULE strings

INTERFACE OPERATOR (.CONCAT.)
  MODULE PROCEDURE concatentation
END INTERFACE
PRIVATE concatenation

CONTAINS
  FUNCTION concatentation(A, B)
    CHARACTER(LEN=*), INTENT(IN) :: A, B
    CHARACTER(LEN=LEN_TRIM(A)+LEN_TRIM(B)) :: concatentation
    concatentation = TRIM(A)//TRIM(B)
  END FUNCTION concatentation
END MODULE strings
```

If a subroutine has two arguments, the first being intent(out) or intent(inout) and the second being intent(in), then it may be given an *assignment overload*:

```
interface assignment (=)
  [interface-body] ...
  [module-procedure-stmt] ...
end interface
```

An example is:

```
interface assignment (=)
  subroutine to_char_from_int(c, i)
    character :: c
    integer :: i
  end subroutine
end interface
```

The purpose of such a subroutine is to convert the value of the second argument to an appropriate value for the type of the first argument, and to assign this converted value to the first actual argument; the subroutine defines the conversion that takes place in such an assignment. The *assignment interface* makes it possible to use assignment syntax for this operation, as an alternative to using normal subroutine calls. In analogy with operator interfaces, assignment interfaces define assignment overloads and thus must be consistent with the generic reference resolution rules. Intrinsic assignments cannot be redefined except for intrinsic assignment of structures (that is, derived type intrinsic assignment). As with operator functions, assignment subroutines may be either external subroutines or module subroutines.

Interface blocks may be used to define overloaded (generic) procedure names. Any procedure name may be (further) overloaded, including an intrinsic procedure name:

```
interface generic-name
  [interface-body] ...
  [module-procedure-stmt] ...
end interface
```

A *generic name* may be associated with any number of external procedures and module procedures. Such a procedure may be called using either its original (specific) name or the generic name. A call to a procedure using the generic name is considered to be a generic reference; any generic reference must be resolvable to a specific procedure, in accordance with the generic reference resolution rules. Generally speaking, this means that each procedure sharing the same generic name must have a different argument "signature" (type pattern). External and module procedures may be given generic interfaces. For example:

```
INTERFACE LOGB
  MODULE PROCEDURE SP_LOGB, DP_LOGB
END INTERFACE LOGB

CONTAINS
  FUNCTION SP_LOGB(X)
    REAL(KIND(1.0E0)) :: SP_LOGB, X
    SP_LOGB = EXPONENT(X) - 1.0E0
  END FUNCTION SP_LOGB

  FUNCTION DP_LOGB(X)
    REAL(KIND(1.0D0)) :: DP_LOGB, X
    DP_LOGB = EXPONENT(X) - 1.0D0
  END FUNCTION DP_LOGB
```

A *module-procedure-stmt* takes the form of:

```
module procedure
```

Interface bodies are not necessary as the compiler will determine the interface from the actual implementation of the module procedure. An example would be:

```
interface c_associated
  module procedure c_associated_c_ptr, c_associated_c_funptr
end interface c_associated
```

Abstract Interfaces

An abstract interface describes the characteristics of a procedure without defining the interface of specific subroutine or function. It is useful for defining an interface for a procedure pointer that may point to one of several different procedures, all with the same characteristics. The syntax is the same as an explicit interface with the addition of the keyword ABSTRACT:

```
abstract interface
  interface-body
  [interface-body] ...
end interface
```

For example:

```
ABSTRACT INTERFACE
  FUNCTION REAL_FUN(X)
    REAL :: REAL_FUN, X
  END FUNCTION REAL_FUN
END INTERFACE

PROCEDURE(REAL_FUN), POINTER :: FUNC
```

CHAPTER 6

Expressions and Assignment Statements

Being primarily a computational language, a large number of FORTRAN statements employ expressions. The evaluation of an expression results in a single value which may be used to define a variable, take part in a logical decision, be written to a file, etc. The simplest form of an expression is a scalar value: a constant or single variable. More complicated expressions can be formed by specifying operations to be performed on one or more operands.

There are four types of expressions available in Fortran: arithmetic, character, relational, and logical. This chapter describes the rules for the formation and evaluation of these expressions.

Assignment statements, together with expressions, are the fundamental working tools of FORTRAN. Assignment statements are used to establish a value for variables and array elements. Assignment statements assign a value to a storage location.

ARITHMETIC EXPRESSIONS

An arithmetic expression produces a numeric result and is formed with integer, real, double precision, and complex operands and arithmetic operators. An arithmetic operand may be one of the following:

- an arithmetic scalar value
- an arithmetic array element
- an arithmetic expression enclosed in parentheses
- the result of an arithmetic function

The arithmetic operators are:

Operator	Purpose
**	exponentiation
*	multiplication
/	division
+	addition or identity
-	subtraction or negation

The operators **, *, and / operate only on pairs of operands, while + and - may operate on either pairs of operands or on single operands. Pairs of operators in succession are not allowed: A+-B must be stated as A+(-B). In addition, there is precedence among the arithmetic operators which establishes the order of evaluation:

Operator	Precedence
**	highest
* and /	intermediate
+ and -	lowest

Except for the exponentiation operator, when two or more operators of equal precedence occur consecutively within an arithmetic expression they may be evaluated in any order if the result of the expression is mathematically equivalent to the stated form. However, exponentiation is always evaluated from right to left:

Expression	Evaluation
A+B-C	(A+B)-C or A+(B-C)
A**B**C	A**(B**C)
A+B/C	A+(B/C)

However, the result of an arithmetic expression involving integer operands and the division operator is the quotient; the remainder is discarded: 10/3 produces an integer result of 3. Consequently, expressions such as I*J/K may have different values depending on the order of evaluation:

(4*5)/2 = 10, but 4*(5/2) = 8

Data Type of Arithmetic Expressions

When all of the operands of an arithmetic expression are of the same data type, the data type of the result is the same as that of the operands. When expressions involving operands of different types are evaluated, automatic conversions between types occur. These conversions are always performed in the direction of the highest ordered data type presented and the data type of the result is that of the highest ordered operand encountered. `INTEGER` is the lowest ordered data type and `COMPLEX` is the highest.

Data Type Conversion Order

```
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
COMPLEX*16
```

Consider the expression `I/R*D+C`, where `I` is `INTEGER`, `R` is `REAL`, `D` is `DOUBLE PRECISION`, and `C` is `COMPLEX`. The evaluation proceeds as follows:

- the value of I is converted to REAL and then divided by the value of R
- the result of the division is implicitly converted to DOUBLE PRECISION and multiplied by the value of D
- the result of the multiplication is then added to the real portion of the of the value C giving DOUBLE PRECISION
- the imaginary portion of the value C is implicitly converted to DOUBLE PRECISION in the final result
- the data type of the result of the expression is COMPLEX*16

Parentheses are used to force a specific order of evaluation that the compiler may not override.

When exponentiation of REAL, DOUBLE PRECISION, and COMPLEX operands involves integer powers, the integer power is not converted to the data type of the other operand. Exponentiation by an integer power is a special operation which allows expressions such as -2.1**3 to be evaluated correctly.

Conversion from a lower to a higher precision does not increase the accuracy of the converted value. For example, converting the result of the real expression 1.0/3.0 to DOUBLE PRECISION yields:

```
0.333333343267441D+00
```

not:

```
0.333333300000000D+00 or 0.333333333333333D+00
```

Arithmetic Constant Expression

Arithmetic expressions in which all of the operands are constants or the symbolic names of constants are called arithmetic constant expressions. INTEGER, REAL, DOUBLE PRECISION, and COMPLEX constant expressions may be used in PARAMETER statements. Integer constant expressions may also be used in specification and declaration statements (see the **Specifications and DATA Statements** chapter).

CHARACTER EXPRESSIONS

A CHARACTER expression produces a character result and is formed using character operands and character operators. A CHARACTER operand may be one of the following:

- a CHARACTER scalar value
- a CHARACTER array element
- a CHARACTER substring
- a CHARACTER expression enclosed in parentheses
- the result of a CHARACTER function

The only CHARACTER operator is //, meaning concatenation. Although parentheses are allowed in character expressions, they do not alter the value of the result. The following character expressions all produce the value 'CHARACTER':

```
'CHA'//'RAC'//'TER'
('CHA'//'RAC')//'TER'
'CHA'//('RAC'//'TER')
```

RELATIONAL EXPRESSIONS

A relational expression produces a logical result (.TRUE. or .FALSE.) and is formed using arithmetic expressions or CHARACTER expressions and relational operators. The relational operators perform comparisons; they are:

Operator	Comparison
< or .LT.	less than
<= or .LE.	less than or equal to
== or .EQ.	equal to
/= or .NE.	not equal to
<> or .NE.	not equal to
> or .GT.	greater than
>= or .GE.	greater than or equal to

Only the .EQ. and .NE. relational operators can be applied to complex operands.

All of the relational operators have the same precedence which is lower than the arithmetic operators and the character operator.

If the data types of two arithmetic operands are different, the operand with the lowest order is converted to the type of the other operand before the relational comparison is performed. The same type coercion rules apply to relational operators as arithmetic operators when comparisons are made, but results are always returned as LOGICAL.

Character comparison proceeds on a character by character basis using the ASCII collating sequence to establish comparison relationships. Since the letter 'A' precedes the letter 'B' in the ASCII code, 'A' is less than 'B'. Also, all of the upper case characters have lower "values" than the lower case characters. A complete chart of the ASCII character set is provided in the appendices.

When the length of one of the `CHARACTER` operands used in a relational expression is shorter than the other operand, the comparison proceeds as though the shorter operand were extended with blank characters to the length of the longer operand.

When an integer value is compared with a `CHARACTER` constant, one to four bytes of the character string are extracted as an integer and a comparison is made between the two integer values. This is useful if the integer has been defined with a Hollerith data type. This type of comparison is only defined for character constants with a length less than or equal to four.

LOGICAL EXPRESSIONS

A `LOGICAL` expression is formed with `LOGICAL` or `INTEGER` operands and logical operators. A `LOGICAL` operand may be one of the following:

- a `LOGICAL` or `INTEGER` scalar value
- a `LOGICAL` or `INTEGER` array element
- a `LOGICAL` or `INTEGER` expression enclosed in parentheses
- a relational expression
- the result of a `LOGICAL` or `INTEGER` function

A `LOGICAL` expression involving `LOGICAL` operands and relational expressions produces a `LOGICAL` result (`.TRUE.` or `.FALSE.`). When applied to `LOGICAL` operands the logical operators, their meanings, and order of precedence are:

Operator	Purpose	Precedence
`.NOT.`	negation	highest
`.AND.`	conjunction	
`.OR.`	inclusive disjunction	
`.EQV.`	equivalence	lowest
`.NEQV.`	nonequivalence	
`.XOR.`	nonequivalence	same as `.NEQV.`

A LOGICAL expression involving INTEGER operands produces an INTEGER result. The operation is performed on a bit-wise basis. When applied to integer operands the logical operators have the following meanings:

Operator	Purpose
.NOT.	one's complement
.AND.	Boolean and
.OR.	Boolean or
.EQV.	integer compare
.NEQV.	Boolean exclusive or
.XOR.	Boolean exclusive or

Note that expressions involving INTEGER and LOGICAL operands are ambiguous.

Consider the following example:

```
LOGICAL l,m
INTEGER i

i = 2
l = .TRUE.
IF (l .AND. i) WRITE(*,*) "first if clause executed"
m = i
IF (l .AND. m) WRITE(*,*) "second if clause executed"
END
```

Since the LOGICAL constant .TRUE. has a value of one when converted to INTEGER, the first IF clause would not normally be executed since a Boolean .AND. between the values one and two produces a zero which is logically false. For this reason, integers are converted to LOGICAL when the .AND. operator is used with a LOGICAL. The second IF clause will execute since the conversion of a non-zero INTEGER to LOGICAL (in the assignment statement m = i) gives the value .TRUE. (stored as the value one). Note that the above code is not guaranteed to be portable to other FORTRAN environments.

OPERATOR PRECEDENCE

As described above, a precedence exists among the operators used with the various types of expressions. Because more than one type of operator may be used in an expression, a precedence also exists among the operators taken as a whole: arithmetic is the highest, followed by character, then relational, and finally logical which is the lowest.

```
A+B .GT. C .AND. D-E .LE. F
```

is evaluated as:

```
((A+B) .GT. C) .AND. ((D-E) .LE. F)
```

ARITHMETIC ASSIGNMENT STATEMENT

Arithmetic assignment statements are used to store a value in arithmetic variables. Arithmetic assignment statements take the following form:

```
v = e
```

where: *v* is the symbolic name of an integer, real, double precision, or complex variable or array element whose contents are to be replaced by *e*.

e is a character constant or arithmetic expression.

If the data type of *e* is arithmetic and different than the type of *v* , then the value of *e* is converted to the type of *v* before storage occurs. This may cause truncation.

If *e* is a CHARACTER constant, bytes of data will be moved directly to the storage location with no type conversion. If *e* is a CHARACTER expression, a type mismatch error will occur.

LOGICAL ASSIGNMENT STATEMENT

LOGICAL assignment statements are used to store a value in LOGICAL variables. LOGICAL assignment statements are formed exactly like arithmetic assignment statements:

```
v = e
```

where: *v* is the symbolic name of a logical variable or logical array element.

e is an arithmetic or logical expression.

If the data type of *e* is not LOGICAL, the value assigned to *v* is the LOGICAL value .FALSE. if the value of the expression *e* is zero. For non-zero values of *e*, the value assigned to *v* is the LOGICAL value .TRUE.. This rule for the conversion of an arithmetic expression to a LOGICAL value applies wherever a LOGICAL expression is expected (i.e. an IF statement).

CHARACTER ASSIGNMENT STATEMENT

`CHARACTER` assignment statements are used to store a value in `CHARACTER` variables:

```
v = e
```

where: *v* is the symbolic name of a character variable, character array element, or character substring. *e* is an expression whose type is character.

If the length of *e* is greater than the length of *v* , the leftmost characters of *e* are used.

If the length of *e* is less than the length of *v* , blank characters are added to the right of *e* until it is the same length as *v* .

ASSIGN STATEMENT

The `ASSIGN` statement is used to store the address of a labeled statement in an integer variable. Once defined with a statement label, the integer variable may be used as the destination of an assigned `GOTO` statement (**Control Statements** chapter) or as a format descriptor in an I/O statement (**Input/Output and Format Specification** chapter). The `ASSIGN` statement is given in the following manner.

```
ASSIGN s TO i
```

where: *s* is the label of a statement appearing in the same program unit that the `ASSIGN` statement does.

i is an `INTEGER` variable name.

Caution: No protection is provided against attempting to use a variable that does not contain a valid address as established with the `ASSIGN` statement.

CHAPTER 7

Control Statements

Control statements direct the flow of execution in a Fortran program. Included in the control statements are constructs for looping, conditional and unconditional branching, making multiple choice decisions, and halting program execution.

GOTO STATEMENTS

Unconditional GOTO

The unconditional `GOTO` statement causes immediate transfer of control to a labeled statement:

```
GOTO s
```

The statement label *s* must be in the same program unit as the `GOTO` statement.

Computed GOTO

The computed `GOTO` statement provides a means for transferring control to one of several different destinations depending on a specific condition:

```
GOTO (s [,s]...) [,] e
```

e is an expression which is converted as necessary to integer and is used to select a destination from one of the statements in the list of labels (*s* [,*s*]...). The selection is made such that if the value of *e* is one, the first label is used, if the value of *e* is two, the second label is used, and so on. The same label may appear more than once in the label list. If the value of *e* is less than 1 or greater than the number of labels in the list no transfer is made. All of the statement labels in the list must be in the same program unit as the computed `GOTO` statement.

Assigned GOTO

The assigned `GOTO` statement is used with an integer variable that contains the address of a labeled statement as established with an `ASSIGN` statement:

```
GOTO i [[,] (s [,s]...)]
```

The address of the labeled statement contained in the integer variable *i* is used as the destination. If the optional list of statement labels, (*s* [,*s*]...), appears then *i* must be defined with the address of one of them or no transfer is made.

IF STATEMENTS

Arithmetic IF

The arithmetic `IF` statement is used to transfer control based on the sign of the value of an expression:

```
IF (e) s1 , s2 , s3
```

e can be an `INTEGER`, `REAL`, or `DOUBLE PRECISION` expression which if negative, transfers control to the statement labeled *s1*; if zero, transfers control to the statement labeled *s2*; and if positive, transfers control to the statement labeled *s3*. The statements labeled *s1*, *s2*, and *s3* must be in the same program unit as the arithmetic `IF` statement.

Logical IF

The logical `IF` statement is used to execute another statement based on the value of a logical expression:

```
IF (e) st
```

The statement *st* is executed only if the value of the logical expression *e* is `.TRUE.`. The statement *st* cannot be any of the following: `DO`, `IF`, `ELSE IF`, `ELSE`, `END IF`, `END`, `END DO`, `REPEAT`, `SELECT CASE`, `CASE`, or `END SELECT`.

Block IF

A block `IF` consists of `IF` (*e*) `THEN`, `ELSE`, and `END IF` statements. Each `IF` (*e*) `THEN` statement must be balanced by an `END IF` statement. A block `IF` provides for the selective execution of a particular block of statements depending on the result of the `LOGICAL` expression *e*.

```
[if-construct-name:] IF (e) THEN
      block of statements
ELSE [if-construct-name]
      block of statements
END IF [if-construct-name]
```

The `ELSE` statement and the second block of statements are optional. If the value of the `LOGICAL` expression *e* is `.TRUE.`, the first block of statements is executed and then control of execution is transferred to the statement immediately following the `END IF` statement. If *e* has a `.FALSE.` value, then, if a second block of statements exists

(constructed by `ELSE` or `ELSE IF` statements) it is executed, and control of execution is transferred to the statement immediately following the `END IF` statement.

A block `IF` construct may be labeled with an *if-construct-name* for clarity.

Each block of statements may contain more block `IF` constructs. Since each block `IF` must be terminated by an `END IF` statement there is no ambiguity in the execution path.

A more complicated block `IF` can be constructed using the alternate form of the `ELSE` statement: the `ELSE IF (e) THEN` statement. Multiple `ELSE IF (e) THEN` statements can appear within a block `IF`, each one being evaluated if the previous logical expression *e* has a `.FALSE.` value:

```
IF (I.GT.0 .AND. I.LE.10) THEN
     block of statements
ELSE IF (I.GT.10 .AND. I.LE.100) THEN
     block of statements
ELSE IF (I.GT.100 .AND. I.LE.1000) THEN
     block of statements
ELSE
     block of statements
END IF
```

LOOP STATEMENTS

The `DO` statements provide the fundamental structure for constructing loops in Fortran. The standard `DO` loop is discussed in this section.

Basic DO loop

The basic `DO` statement takes the following form:

```
DO [s [,]] i = e1, e2 [,e3]
```

where: *s* is the label of the statement that defines the range of the `DO` loop and must follow the `DO` statement in the same program unit. If *s* is omitted, the loop must be terminated with an END DO statement.

i is called the `DO` variable and must be either an `INTEGER`, `REAL`, or `DOUBLE PRECISION` scalar variable

e1, *e2*, and *e3* may be integer, real, or double precision expressions whose values are called the initial value, the limit value, and the increment value, respectively

The loop termination statement, labeled *s*, if present, must not be a `DO`, arithmetic `IF`, block `IF`, `ELSE`, `END IF`, unconditional `GOTO`, assigned `GOTO`, `RETURN`, `STOP`, `END`, `SELECT CASE`, `CASE`, or `END SELECT` statement.

DO loops may be nested to any level, but each nested loop must be entirely contained within the range of the outer loop. The termination statements of nested DO loops may be the same. If the termination statement of nested DO loops is the same, no more than 16 DO loops may have the same termination statement.

DO loops may appear within IF blocks and IF blocks may appear within DO loops, but each structure must be entirely contained within the enclosing structure.

DO Loop Execution

The following steps govern the execution of a DO loop:

1. The expression *e1*, the initial value, is evaluated and assigned to the DO variable *i*, with appropriate type conversion as necessary.

2. The expressions *e2* and *e3*, the limit value and increment value respectively, are evaluated. If *e3* is omitted, it is given the default value of one.

3. The iteration count is calculated from the following expression:

   ```
   MAX( INT( (e2 - e1 + e3)/e3), 0)
   ```

   and determines how many times the statements within the loop will be executed.

4. The iteration count is tested, and if it is zero, control of execution is transferred to the statement immediately following the loop termination statement.

5. The statements within the range of the loop are executed.

6. The DO variable is increased by the increment value, the iteration count is decreased by one, and control branches to step four.

Variables that appear in the expressions *e1*, *e2*, and *e3* may be modified within the loop, without affecting the number of times the loop is iterated.

```
      K = 0
      L = 10
      DO 10 I=1, L
      DO 10 J=1, I
      L = J
10    K = K+1
```

When the execution of both the inner and outer loops is finished, the values of both I and J are 11, the value of K is 55, and the value of L is 10.

DO WHILE

The `DO WHILE` statement provides a method of looping not necessarily governed by an iteration count. The form of the `DO WHILE` statement is:

```
DO WHILE (e)
```

where: *e* is a `LOGICAL` expression.

The `DO WHILE` statement tests the `LOGICAL` expression at the top of the loop. If the expression evaluates to a `.TRUE.` value, the statements within the body of the loop are executed. If the expression evaluates to a `.FALSE.` value, execution proceeds with the statement following the loop:

```
INTEGER            :: status
INTEGER, PARAMETER :: eof-1

DATA a,b,c /3*0.0/

status = 0
DO WHILE (status<>eof)
     c = c + a*b
     READ (*,*,IOSTAT=status) a,b
END DO
```

Infinite DO

```
DO
      block
END DO
```

This is essentially a `DO` forever construct for use in situations where the number of loop iterations is unknown and must be determined from some external condition (i.e. processing text files).

END DO statement

The `END DO` statement is used to terminate `DO` loops. After execution of an `END DO` statement, the next statement executed depends on the result of the `DO` loop incrementation processing. The form of the `END DO` statement is:

```
END DO
```

EXIT statement

The EXIT statement provides a convenient means for abnormal termination of a DO loop. This statement cause control of execution to be transferred to the statement following the terminal statement of a DO loop or block DO.

```
DO
     READ (*,*,IOSTAT=ios) v1,v2; IF (ios==-1) EXIT
     CALL process(v1,v2)
END DO
```

CYCLE statement

The CYCLE statement causes immediate loop index and iteration count processing to be performed for the DO loop or block DO structure to which the CYCLE statement belongs.

```
READ (*,*) n
z = 0.0
DO (n TIMES)
     READ (*,*) x,y; IF (y==0.0) CYCLE
     z = z + x/y
END DO
```

CONTINUE STATEMENT

The CONTINUE statement is used to provide a reference point. It is usually used as the terminating statement of a basic DO loop, but it can appear anywhere in the executable section of a program unit. Executing the CONTINUE statement itself has no effect. The form of the CONTINUE statement is:

```
CONTINUE
```

Labeled DO loops

DO loops may use labels for clarity and allow iterating or exiting a specific loop. The label is called a *do-construct-name*. If a DO loop is labeled with a *do-construct-name*, the corresponding END DO statement must have the same *do-construct-name*. CYCLE and EXIT statements may mention a *do-construct-name* to indicate the specific loop they refer to. Examples:

```
outer_loop: do
              . . .
inner_loop:   do
                . . .
                if ( . . . ) cycle inner_loop
                . . .
                if ( . . . ) exit outer_loop
                . . .
              end do inner_loop
              . . .
            end do outer_loop
```

BLOCK CASE

The block CASE structure is an extension to the FORTRAN standard for constructing blocks which are executed based on comparison and range selection. The SELECT CASE statement is used with an END SELECT statement, at least one CASE statement and, optionally, a CASE DEFAULT statement to control the execution sequence. The SELECT CASE statement is used to form a block CASE.

The form of a block CASE is:

```
[case-contruct-name] SELECT CASE (e)
      CASE (case_selector) [case-contruct-name]
            [block]
      [CASE (case_selector) [case-contruct-name]
            [block]
      ...]
      [CASE DEFAULT [case-contruct-name]]
            [block]
END SELECT [case-contruct-name]
```

where: *e* is an expression formed from one of the enumerative data types: CHARACTER, INTEGER, REAL, or DOUBLE PRECISION. For the purposes of the block CASE construct, the value of CHARACTER expression is its position in the ASCII collating sequence.

case-construct-name is an optional label than may be used for identification. If a *case-construct-name* is used is must be the same for each element of the CASE structure.

A CASE block must contain at least one CASE statement and must be terminated by an END SELECT statement. Control of execution must not be transferred into a block CASE.

CASE blocks are delimited by a CASE statement and the next CASE, CASE DEFAULT, or END SELECT statement. A CASE block may be empty. After execution of a CASE block, control of execution is transferred to the statement following the END SELECT statement with the same CASE level. Block CASE structures may be nested. Since each block CASE must be terminated by an END SELECT statement there is no ambiguity in the execution sequence.

A *case_selector* takes the form of either of the following:

```
CASE (con[,con,...,con])

CASE DEFAULT
```

con may be either a value selector or a range selector. A value selector is a constant. A range selector takes one of the following three forms:

con1:*con2*	where	(*con1* .LE. *e*).AND.(*e* .LE. *con2*)
con:	where	*con* .LE. *e*
:*con*	where	*e* .LE. *con*

All constants must be of the same type as the expression *e* in the SELECT CASE statement. A block CASE may have only one CASE DEFAULT statement where control of execution is transferred if no match is found in any other CASE statement. If a CASE DEFAULT statement is not present and no other match is found, there is no match.

Execution of a block CASE statement

Execution of block CASE statement causes evaluation of the expression *e* in the SELECT CASE statement. The value of the expression is then compared sequentially with the parameters of the case selectors. If a match is made, transfer of control is passed to that case block. If the comparison fails, the next case selector is checked.

Block CASE Example

```
!
!    routine to count the number and types of characters
!    in a text file
!
USE ISO_FORTRAN_ENV
IMPLICIT INTEGER(a-z)
CHARACTER line*80

lines=0; alf=0; num=0; blk=0; trm=0; spl=0

  DO
    READ (*,'(a)',IOSTAT=ios) line
    IF (ios==IOSTAT_END) EXIT
    chars = LEN(TRIM(line))
    lines = lines+1
      DO (i=1, chars)
        SELECT CASE (line(i:i))
          CASE ("A":"Z","a":"z")
            alf = alf+1
          CASE ("0":"9")
            num = num+1
          CASE (" ")
            blk = blk+1
          CASE (".","!","?")
            trm = trm+1
          CASE DEFAULT
            spl = spl+1
        END SELECT
      END DO
  END DO

END
```

STOP STATEMENT

The `STOP` statement terminates execution of a program:

```
STOP [s]
```

The optional string *s* may be a `CHARACTER` constant or string of five or fewer digits and is output to standard out with end of record characters.

PAUSE STATEMENT

The `PAUSE` statement suspends execution until a carriage return character is read from standard input (usually from the keyboard).

```
PAUSE [s]
```

The optional string *s* may be a `CHARACTER` constant or string of five or fewer digits and is output to unit * without end of record characters.

END STATEMENT

Every program unit and module must have an END statement which terminates the range of the program unit or module within a source file. A source file itself may contain more than one program unit; the entry points of the individual program units in the compiled object file are available to the linker.

An END statement in a program unit is executable and if encountered in a main program has the effect of a STOP statement and if encountered in a subroutine or function subprogram has the effect of a RETURN statement. An END statement is given on a statement line by itself:

```
END [type [name]]
```

where: *type* is PROGRAM, SUBROUTINE, FUNCTION, or MODULE.

name is the name of the program unit or module.

WHERE

The WHERE keyword can be used both as a statement and a construct, similar to the IF keyword. WHERE is used to perform masked array assignments, applying a logical test to each element of an array. The syntax of the WHERE statement is:

```
WHERE ( mask_expr )assign_stmt
```

where: *mask_expr* is a logical array expression

assign_stmt is an array assignment statement. The shape of the array must be the same as the shape of the array used in the *mask_expr*

In the following example, the arcsine function will only be evaluated if the absolute value of the element of the array a is less than or equal to 1.0.

```
REAL a(100), b(100)
.
.
.
WHERE (ABS(a) <= 1.0) b = ASIN(a)
```

The syntax of the WHERE construct is:

```
[name:] WHERE ( mask_expr )
      [where_body_construct]
[ELSEWHERE ( mask_expr ) [name]
      where_body_construct]
[ELSEWHERE [name]
      where_body_construct]
END WHERE [name]
```

where: `mask_expr` is a logical array expression

`where_body_construct` is an array assignment statement or a `WHERE` statement or construct. The shape of all arrays must be the same as the shape of the array used in the `mask_expr`

Example:

```
REAL, DIMENSION(10,10) :: A

      WHERE (A <= 10.0)
        A = A -  0.5
      ELSEWHERE (A <= 100.0)
        A = A -  5.0
      ELSEWHERE
        A = A - 50.0
      END WHERE
```

FORALL

The `FORALL` keyword can be used both as a statement and a structure. It is similar to the masked array assignment `WHERE`, but is more general, allowing more array shapes to be assigned. It is used to perform array assignments, possibly masked, on an element by element basis. The syntax of the `FORALL` statement is:

```
FORALL (triplet_spec [,triplet_spec]… [,mask_expr] )assign_stmt
```

where: *triplet_spec* is a triplet specification of an index variable normally used as an array element index. It has the following form:

```
index = subscript : subscript [: stride]
```

where: *index* is a scalar integer variable. It is valid only with the scope of the `FORALL` statement

subscript is a scalar integer expression and may not contain a reference to any *index* in the *triplet_spec* in which it appears

stride is a scalar integer expression and may not be zero. If omitted, a default value of 1 is supplied. It. may not contain a reference to any *index* in the *triplet_spec* in which it appears

`mask_expr` is any logical scalar expression, including one which references an *index* of a *triplet_spec*.

`assign_stmt` is an assignment statement or a pointer assignment statement.

In the following example, every element of the array `a` is assigned the value 1.0.

```
REAL a(100, 100)
.
.
.
FORALL (i=1:100, j=1,100) a(i,j) = 1.0
```

The syntax of the `FORALL` construct is:

```
[name:] FORALL (triplet_spec [,triplet_spec]… [,mask_expr] )
     forall_body_construct
END FORALL [name]
```

where: *triplet_spec* is a triplet specification of an index variable normally used as an array element index. It has the following form:

```
index = subscript : subscript [: stride]
```

where: *index* is a scalar integer variable. It is valid only with the scope of the `FORALL` statement

subscript is a scalar integer expression and may not contain a reference to any *index* in the *triplet_spec* in which it appears

stride is a scalar integer expression and may not be zero. If omitted, a default value of 1 is supplied. It. may not contain a reference to any *index* in the *triplet_spec* in which it appears

`mask_expr` is any logical scalar expression, including one which references an *index* of a *triplet_spec*.

`forall_body_construct` is an assignment statement, pointer assignment statement, `WHERE` statement or construct, or `FORALL` statement or construct.

CHAPTER 8

Input/Output and FORMAT Specification

Input and output statements provide a channel through which Fortran programs can communicate with the outside world. Facilities are available for accessing disk and tape files, communicating with terminals and printers, and controlling external devices. Fortran input and output statements are designed to allow access to the wide variety of features implemented on various computer systems in the most portable manner possible.

A format specification is used with formatted input and output statements to control the appearance of data on output and provide information regarding the type and size of data on input. Converting the internal binary representation of a floating point number into a string of digits requires a format specification and is called editing. A format specification divides a record into fields, each field representing a value. An explicitly stated format specification designates the exact size and appearance of values within fields.

When an asterisk (*) is used as a format specification it means "list directed" editing. Instead of performing editing based on explicitly stated formatting information, data will be transferred in a manner which is "reasonable" for its type and size.

Throughout the remainder of this chapter, input and output will be referred to in the conventional abbreviated form: I/O.

RECORDS

All FORTRAN I/O takes place through a data structure called a record. A record can be a single character or sequence of characters or values. A record might be a line of text, the data received from a bar code reader, the coordinates to move a plotter pen, or a punched card. FORTRAN uses three types of records:

- Formatted
- Unformatted
- Endfile

Formatted Record

A formatted record is a sequence of ASCII characters. It may or may not be terminated depending on the operating system. If it is terminated, the usual terminating characters are a carriage return, a line feed, or both. A single line of text on this page is a formatted record. The minimum length of a formatted record is zero. The maximum record length is limited only by available memory.

Unformatted Record

An unformatted record is a sequence of values. Its interpretation is dependent on the data type of the value. For example, the binary pattern 01010111 can be interpreted as the integer value 87 or the character value "W" depending on its data type. The minimum length of an unformatted record is zero. Records in unformatted sequential access files which contain no record length information (see below) have unlimited length. Records in unformatted sequential access files that contain imbedded record length information have a maximum size of 2,147,483,647 bytes. The maximum length of direct access unformatted records is limited only by available memory.

Endfile Record

The endfile record is the last record of a file and has no length. An endfile record may or may not be an actual record depending on the file system of a particular operating system.

FILES

A file is composed of zero or more records and can be created and accessed by means other than Fortran programs. For example, a text processor might be used to create and edit a document file and a Fortran program used to manipulate the information in the file.

Files that are usually stored on disks or tapes are called external files. Files can also be maintained in main memory. These are called internal files.

File Name

Most external files are accessed explicitly by their names. While the file naming conventions of operating systems vary greatly, Fortran can accommodate most of the differences. The circumstances where a name is not required to access a file are discussed later in this chapter.

File Position

The position within a file refers to the next record that will be read or written. When a file is opened it is usually positioned to just before the first record. The end of the file is just after the last record. Some of the I/O statements allow the current position within a file to be changed.

File Access

The method used to transfer records to and from files is called the access mode. External files may contain either formatted or unformatted records. When the records in a file can be read or written in an arbitrary manner, randomly, the access mode is direct. Individual records are accessed through a record number, a positive integer. All of the records in a direct access file have the same length and contain only the data actually written to them;

there are no record termination characters. Records may be rewritten, but not deleted. Generally, only disk files can use the direct access mode of record transfer.

When the records are transferred in order, one after another, the access mode is sequential. The records in a sequential access file may be of different lengths. Some files, like terminals, printers, and tape drives, can only use the sequential access mode.

Formatted sequential access files usually contain textual information and each record has a terminating character(s) as described above.

Unformatted sequential access is generally used for two conflicting, but equally common purposes:

- For controlling external devices such as plotters, graphics terminals, and machinery as well as processing unencoded binary information such as object files. In this case it is important that the data transferred to and from the external media be a true byte stream containing no record length information.

- For its data compression and speed of access characteristics. In this case it must be possible to determine the length of a record for partial record reads and backspacing purposes.

This implementation of Fortran contains provisions for both of these requirements. The default manner of unformatted processing of a sequential access is to imbed record length information for a maximum record size of 2,147,483,647 bytes. The `OPEN` statement `FORM` and `ACCESS` specifiers provide mechanisms for disabling record length information (see below).

In addition to sequential and direct access, data may also be transferred to a file using stream access. This method considers a file to a sequence of storage units of length one which can be indexed by a positive integer. The first storage position in a stream access file is 1. Stream access files can be either unformatted or formatted.

Internal Files

Internal files are comprised of `CHARACTER` variables, `CHARACTER` array elements, `CHARACTER` substrings, or `CHARACTER` arrays. An internal file which is a `CHARACTER` variable, `CHARACTER` array element, or character substring has one record whose length is the length of the character entity. An internal file that is a CHARACTER array has as many records as there are array elements. The length of an individual record is the length of a `CHARACTER` array element. Data may only be transferred through the formatted sequential access mode. Internal files are usually used to convert variables between numeric and `CHARACTER` data types.

Input/Output Editing

Editing provides the means for converting between the internal format of a value and the external format:

internal 01000100010000110100001001000001

external	interpretation
ABCD	character
1145258561	integer
781.035	floating point

See **Giving A Format Specification** later in this chapter

I/O SPECIFIERS

Fortran I/O statements are formed with lists of specifiers that are used to identify the parameters of the operation and direct the control of execution when exceptions occur.

Unit Specifier

The mechanism through which a channel of communication with a file is established and maintained is called a unit. A unit may be either explicitly or implicitly identified, and may refer to an external or internal file. When the channel is established, the unit is said to be connected to the file. The relationship is symmetric; that is, you can also say that the file is connected to the unit.

A connection to an external file is established and maintained with an external unit identifier that is an integer expression whose value is an arbitrary positive integer. An external unit identifier is global to the program; a file opened in one program unit may be referenced with the same unit number in other program units. There is no relationship between a FORTRAN unit specifier and the numbers used by various operating systems to identify files.

A connection to an internal file is made with an internal file identifier which is the name of the `CHARACTER` variable, `CHARACTER` array element, `CHARACTER` substring, or `CHARACTER` array that comprises the file.

Unit numbers that are "preconnected" to system devices and default files are:

1. Unit 0 is preconnected to "standard error", usually the screen.

2. Unit 5 is preconnected to "standard input", usually the keyboard, for input operations. Unit 6 is preconnected to "standard output", usually the monitor, for output operations.

3. An asterisk as a unit identifier refers to "standard input" for input operations and "standard output" for output operations.

4. All other unit numbers are preconnected to default files for sequential input and output operations. If a sequential input or output operation references a unit which has not been connected with a FORTRAN `OPEN` statement, the effect is as if an `OPEN` statement with only the `UNIT=` specifier present had been executed to connect the unit. Execution of direct access input and output operations is not permitted on preconnected units.

With the exception of the asterisk, the preconnection of a unit number may be defeated by explicitly connecting the unit number to a file with the FORTRAN `OPEN` statement.

A unit specifier is given as:

```
[UNIT=] u
```

where: *u* is either a positive `INTEGER` expression representing an external unit identifier, or a `CHARACTER` entity representing an internal file identifier.

The characters `UNIT=` may be omitted if the unit identifier occurs first in the list of identifiers.

Implicit File Connections

If an input/output statement specifies a unit that was not connected to a file with an `OPEN` statement, a file is implicitly connected to the unit. The file is given the name:

```
fort.n
```

where *n* is the unit number used in the connection.

Format Specifier

The format specifier establishes the method of converting between internal and external representations. It can be given in one of two ways:

```
[FMT=] f
```

or

```
[FMT=] *
```

where: *f* is the statement label of a `FORMAT` statement, an integer variable that has been assigned a `FORMAT` statement label with an `ASSIGN` statement, a `CHARACTER` array name, or any `CHARACTER` expression

`*` indicates "list directed" editing

The characters `FMT=` may be omitted if the format specifier occurs second in the list of identifiers and the first item is the unit specifier with the characters `UNIT=` also omitted. The following are equivalent:

```
WRITE (UNIT=9, FMT=1000)
WRITE (9,1000)
```

Namelist Specifier

The namelist specifier establishes that conversion from internal and external representations is to be accomplished through namelist directed I/O and is given as:

```
[NML=] n
```

where *n* is the name of a previously defined namelist identifier.

The characters `NML=` may be omitted if the namelist specifier occurs second in the list of identifiers and the first item is the unit specifier with the characters `UNIT=` also omitted.

Record Specifier

The record specifier establishes which direct access record is to be accessed and is given as:

```
REC = rn
```

where *rn* is a positive integer expression.

Error Specifier

The error specifier provides a method to transfer control of execution to a different section of the program unit if an error condition occurs during an I/O statement. It takes as an argument the label of the statement where control is to be transferred:

```
ERR = s
```

where: *s* is the statement label.

End of File Specifier

The end of file specifier provides a method to transfer control of execution to a different section of the program unit if an end of file condition occurs during an I/O statement. It also takes as an argument the label of the statement where control is to be transferred:

```
END = s
```

where: *s* is the statement label.

Asynchronous Specifier

The asynchronous specifier determines whether an I/O statement is synchronous or asynchronous. If NO is specified, the statement may be performed asynchronously. If whether is specified, the statement is synchronous. In the current implementation, all statements are performed synchronously regardless of value this specifier.

```
ASYNCHRONOUS = constant character value
```

where: *constant character value* must be "YES" or "NO".

Blank Specifier

The blank specifier temporarily changes the interpretation of blanks for the duration of the I/O operation.

```
BLANK = s
```

where: *s* is a CHARACTER expression that evaluates to NULL or ZERO.

Decimal Specifier

The decimal specifier temporarily changes the character used to separate the whole and fractional portions in the character strings representing real numbers for the duration of the I/O operation.

```
DECIMAL = s
```

where: *s* is a CHARACTER expression that evaluates to COMMA or POINT.

Delim Specifier

The delim specifier temporarily changes the character used the delimiter for writing list-directed CHARACTER data for the duration of the I/O operation.

```
DELIM = s
```

where: *s* is a CHARACTER expression that evaluates to APOSTROPHE, QUOTE, or NONE.

ID Specifier

The id specifier gives a variable that is associated with an asynchronous I/O operation.

```
ID = id
```

where: *id* is an `INTEGER` variable that receives an identifier for an asynchronous I/O operation. The value zero is assigned for operations that have been completed.

Pad Specifier

The pad specifier temporarily changes the pad mode for the duration of the I/O operation.

```
PAD = s
```

where: *s* is a `CHARACTER` expression that evaluates to `YES` or `NO`.

Pos Specifier

The pos specifier gives the file position for the start of data transfer when a file has been connected for stream access. If the file is open for formatted access, the value given should be 1 for the beginning of the file or a value previously returned by a `POS` specifier in an `INQUIRE` statement for the file.

```
POS = i
```

where: *i* is a `INTEGER` expression.

Sign Specifier

The sign specifier temporarily changes the sign mode for the duration of the I/O operation.

```
SIGN = s
```

where: *s* is a `CHARACTER` expression that evaluates to `PLUS`, `SUPPRESS`, or `PROCESSOR_DEFINED`.

I/O Status Specifier

The I/O status specifier is used to monitor error and end of file conditions after the completion of an I/O statement. Its associated integer variable becomes defined with a -1 if end of file has occurred, a positive integer if an error occurred, and zero if there is neither an error nor end of file condition:

```
IOSTAT = ios
```

where: `ios` is the symbolic name of an `INTEGER` variable or array element.

I/O Message Specifier

If an error, end of file, or end of record condition occurs during an I/O statement, the `CHARACTER` variable associated with the `IOMSG=` specifier is defined with a string describing the error. If no error, end of file, or end of record occurs, the variable is unchanged.

```
IOMSG = msg
```

where: `msg` is the symbolic name of an `CHARACTER` variable or array element.

Partial Record Specifier

Fortran I/O is fundamentally record oriented, and explicit specification is needed for a read (write) statement to consume (produce) only part of a record. In (the default) whole-record I/O, the position of the file is said to *advance* to the next record after a read or a write statement. Thus partial-record I/O is called *nonadvancing* I/O, as the file position is "left where it is" rather than advancing to the beginning of the next record. In nonadvancing input the position of the file is left at the beginning of the next datum within the record that has not yet been read, and the next read statement continues reading from that point; in nonadvancing output an end-of-record is not written by the write statement, and the next write statement continues the same output record.

Nonadvancing I/O is specified with the `ADVANCE="NO"` in the read or write statement; nonadvancing can be specified only for sequential, formatted I/O:

```
ADVANCE = as
```

where: `as` is a `CHARACTER` expression that evaluates to `YES` or `NO`.The default is `YES`.

I/O LIST

The I/O list, iolist, contains the names of variables, arrays, array elements, and expressions (only in output statements) whose values are to be transferred with an I/O statement. The following items may appear in an iolist:

- A variable name
- An array element name
- A CHARACTER substring name
- An array name which is interpreted as every element in the array
- Any expression (only in an output statement)

Implied DO List In An I/O List

The elements of an iolist in an implied DO list are transferred as though the I/O statement was within a DO loop. An implied DO list is stated in the following manner:

```
(dlist, i = e1, e2 [,e3])
```

where: *i* is the DO variable

e1, *e2*, and *e3* establish the initial value, the limit value, and increment value respectively (see the **Control Statements** chapter).

dlist is an iolist and may consist of other implied DO lists

In a READ statement (see below), the DO variable, *i*, must not occur within *dlist* except as an element of subscript, but may occur in the iolist prior to the implied DO list.

DATA TRANSFER STATEMENTS

Data transfer statements transfer one or more records of data.

READ, WRITE and PRINT

The READ statements transfer input data from files into storage and the WRITE and PRINT statements transfer output data from storage to files.

```
READ (cilist) [iolist]

READ f [,iolist]

WRITE (cilist) [iolist]
```

```
PRINT f [,iolist]
PRINT n
```

where: *f* is a format identifier

iolist is an I/O list

n is a list name previously defined in a NAMELIST statement

cilist is a parameter control list that may contain:

1. A unit specifier identifying the file connection.
2. An optional format specifier for formatted data transfers or an optional namelist specifier for namelist directed data transfers, but not both.
3. An optional record specifier for direct access connections.
4. An optional error specifier directing the execution path in the event of error occurring during the data transfer operation.
5. An optional end of file specifier directing the execution path in the event of end of file occurring during the data transfer operation.
6. An optional I/O status specifier to monitor the error or end of file status.
7. An optional I/O message specifier

The PRINT statements, as well as the READ statements which do not contain a *cilist*, implicitly use the asterisk as a unit identifier.

ACCEPT and TYPE

The `ACCEPT` statements transfer input data from records accessed in sequential mode and the `TYPE` statements transfer output data to records accessed in sequential mode.

```
ACCEPT f [,iolist]

ACCEPT n

TYPE f [,iolist]

TYPE n
```

where: *f* is a format identifier

iolist is an I/O list

n is a list name previously defined in a `NAMELIST` statement

The `ACCEPT` and `TYPE` statements implicitly use the asterisk as a unit identifier.

Unformatted Data Transfer

Unformatted data transfer is permitted only to external files. One unedited record is transferred per data transfer statement.

Formatted Data Transfer

Formatted data transfer requires a format specifier which directs the interpretation applied to items in the *iolist*. Formatted data transfer causes one or more records to be transferred.

Printing

`WRITE` statements which specify a unit connected with `ACTION='PRINT'` in the `OPEN` statement (see below) use the first character of each record to control vertical spacing.

This character, called the carriage control character, is not printed and causes the following vertical spacing to be performed before the record is output:

Character	Vertical Spacing
blank	one line
0	two lines
1	top of page
+	no advance (over print)

Any other character appearing in the first position of record or a record containing no characters causes vertical spacing of one line.

OPEN STATEMENT

The OPEN statement connects a unit to an existing file, creates a file and connects a unit to it or modifies an existing connection. The OPEN statement has the following form:

```
OPEN ([UNIT=] u [,olist])
```

where: *u* is the external unit specifier

olist is optional and consists of zero or more of the following specifiers, each of which must have a variable or constant following the equal sign:

FILE	=	a CHARACTER expression which represents the name of the file to be connected to the unit. If this specifier is omitted and the specified unit is not currently connected, a file name will be created.
ACCESS	=	a CHARACTER expression which must be SEQUENTIAL, DIRECT, APPEND, STREAM or TRANSPARENT and specifies the access mode or position. The default value is SEQUENTIAL. TRANSPARENT disables the embedding of record length information.
ACTION	=	a CHARACTER expression which must be READ, WRITE, READWRITE, or PRINT. If READ is specified, only READ statements and file positioning statements are allowed to refer to the connection. If WRITE is specified, only WRITE, PRINT, and file positioning statements are allowed to refer to the connection. If BOTH is specified, any input/output statement may be used to refer to the connection. If PRINT is specified, the first character in each record is interpreted for carriage control (see the previous section on printing) and only WRITE and PRINT statements are allowed to refer to the connection. The default for ACTION is BOTH.
ASYNCHRONOUS	=	a CHARACTER expression which must be YES or NO. The default value is NO. Regardless of the value of the ASYNCHRONOUS specifier, all I/O is synchronous.
BLANK	=	a CHARACTER expression which must be NULL or ZERO specifying how blank characters in formatted numeric input fields are to be handled. A value of ZERO causes blanks in the input field (leading, embedded, and trailing) to be replaced with zeros. The default value is NULL and causes blanks to be ignored.
DECIMAL	=	a CHARACTER expression which must be COMMA or POINT and specifies the character that separates the whole and fractional parts of a real number when represented as a character string.

When POINT is specified, the character is a decimal point. When COMMA is specified, the character is a comma.

DELIM = a CHARACTER expression which must be APOSTROPHE, QUOTE, or NONE and specifies the delimiter for writing list-directed CHARACTER data.

ENCODING = a CHARACTER expression which must be DEFAULT.

ERR = an error specifier as described above.

FORM = a CHARACTER expression which must be FORMATTED, UNFORMATTED, or BINARY specifying the type of records in the file. The default value is UNFORMATTED for direct access files and FORMATTED for sequential access files. BINARY is UNFORMATTED without record length information.

IOSTAT = an I/O status specifier as described above.

IOMSG = an I/O message specifier as described above.

NEWUNIT = an INTEGER variable which receives a negative integer unit number which can be used to identify the connected file in other I/O statements. When this specifier is present, the UNIT specifier must not appear.

PAD = a CHARACTER expression which must be YES or NO specifying whether blank padding is used for input. The default is NO.

POSITION = a CHARACTER expression which must be REWIND, APPEND, or ASIS. If REWIND is specified the file is opened at its beginning position for input or output. If APPEND is specified, the file is opened at its end position for output. The default is ASIS and has the same effect as REWIND.

RECL = a positive INTEGER expression which must be given for direct access file connections and specifies, in bytes, the length of each direct access record.

SIGN = a CHARACTER expression which must be PLUS, SUPPRESS, or PROCESSOR_DEFINED. This specified the initial sign mode which controls the use of optional plus characters in numeric output. When SUPPRESS is specified, any optional plush signs will be omitted in numeric output. When PLUS is specified, optional plus characters will always be produced. The default value is PROCESSOR_DEFINED.

STATUS =	a CHARACTER expression which must be OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. The file must already exist when OLD is specified. The file must *not* exist when NEW is specified. If SCRATCH is specified a file will be created which will exist only during the execution of the program and FILE= must not specified. If UNKNOWN is specified, a file will be created if one does not already exist. The default value is UNKNOWN.
READONLY	a specifier **without** an equal sign equivalent to ACTION=READ.
CARRIAGECONTROL=	a CHARACTER expression which must be FORTRAN, LIST or NONE. Setting the value to FORTRAN is equivalent to ACTION='PRINT'. Setting the value to LIST or NONE has no effect and is only supported for compatibility.
CONVERT =	a CHARACTER expression which must evaluate to BIG_ENDIAN or LITTLE_ENDIAN. This specifier controls the byte ordering of binary data in unformatted files. The default is the ordering appropriate for the type of processor the compiler is installed on.

If a unit is already connected to a file, execution of an OPEN statement for that unit is allowed. If the file to be connected is not the same as the file which is connected, the current connection is terminated before the new connection is established. If the file to be connected is the same as the file which is connected, only the BLANK= and ACTION= specifiers may have a different value from the ones currently in effect. Execution of the OPEN statement causes the new values of BLANK= and ACTION= to be in effect.

CLOSE STATEMENT

The `CLOSE` statement flushes a file's buffers and disconnects a file from a unit. The `CLOSE` statement has the following form:

```
CLOSE ([UNIT=] u [,clist])
```

where: *u* is the external unit specifier.

clist is optional and consists of zero or more of the following specifiers, each of which must have a variable or constant following the equals sign:

`IOSTAT` = an I/O status specifier as described above.

`IOMSG` = an I/O status message as described above.

`ERR` = an error specifier as described above.

`STATUS` = a character expression which must be `KEEP` or `DELETE` which determines whether a file will continue to exist after it has been closed. `STATUS` has no effect if the value of the `STATUS` specifier in the `OPEN` statement was `SCRATCH`. The default value is `KEEP`.

Normal termination of execution of a Fortran program causes all units that are connected to be closed.

BACKSPACE STATEMENT

The `BACKSPACE` statement causes the file pointer to be positioned to a point just before the previous record. The forms of the `BACKSPACE` statement are:

```
BACKSPACE u
BACKSPACE ([UNIT=] u [,alist])
```

where: *u* is the external unit specifier.

alist is optional and consists of zero or more of the following specifiers:

`IOSTAT` = an I/O status specifier as described above.

`IOMSG` = an I/O status message as described above.

`ERR` = an error specifier as described above.

REWIND STATEMENT

The `REWIND` statement causes the file pointer to be positioned to a point just before the first record. The forms of the `REWIND` statement are:

```
REWIND u
REWIND ([UNIT=] u [,alist])
```

where: *u* is the external unit specifier.

alist is optional and consists of zero or more of the following specifiers:

`IOSTAT` = an I/O status specifier as described above.

`IOMSG` = an I/O status message as described above.

`ERR` = an error specifier as described above.

ENDFILE STATEMENT

The `ENDFILE` statement does nothing to disk files. The forms of the `ENDFILE` statement are:

```
ENDFILE u
ENDFILE ([UNIT=] u [,alist])
```

where: *u* is the external unit specifier.

alist is optional and consists of zero or more of the following specifiers:

`IOSTAT` = an I/O status specifier as described above.

`IOMSG` = an I/O status message as described above.

`ERR` = an error specifier as described above.

FLUSH STATEMENT

The `FLUSH` statement causes any pending internal buffers associated with a data transfer statements to be completed. The forms of the `FLUSH` statement are:

```
FLUSH u
FLUSH ([UNIT=] u [,alist])
```

where: *u* is the external unit specifier.

alist is optional and consists of zero or more of the following specifiers:

IOSTAT	=	an I/O status specifier as described above.
IOMSG	=	an I/O status message as described above.
ERR	=	an error specifier as described above

WAIT STATEMENT

The `WAIT` statement causes any pending asynchronous I/O requests associated with a data transfer statements to be completed. The forms of the `WAIT` statement are:

```
WAIT ([UNIT=] u [,alist])
```

where: *u* is the external unit specifier.

alist is optional and consists of zero or more of the following specifiers:

ID	=	an integer variable which contains the identifier of a pending data transfer operation on the specified unit. If the ID specifier appears, the WAIT is performed for the specified operation. If ID is omitted, the WAIT is performed for all pending operations on the specified unit.
IOSTAT	=	an I/O status specifier as described above.
IOMSG	=	an I/O status message as described above.
END	=	an error specifier as described above
EOR	=	an error specifier as described above
ERR	=	an error specifier as described above

INQUIRE STATEMENT

The INQUIRE statement is used to obtain information regarding the properties of files and units. The forms of the INQUIRE statement are:

```
INQUIRE ([UNIT=] u, ilist)
INQUIRE (FILE= fin, ilist)
INQUIRE (IOLENGTH= n) olist
```

The first form, inquiry by unit, takes a unit number as the principal argument and is used for making inquiries about specific units. The unit number, *u*, is a positive integer expression. The second form, inquiry by file, takes a file name as the principal argument and is used for making inquiries about specific named files. The file name, *fin*, is a character expression. The third form is used to determine the length in bytes of *olist*. *olist* is a list of variable names. Only one of UNIT=, FILE=, or IOLENGTH= may be specified. One or more of the following *ilist* specifiers are also used with the INQUIRE statement:

NUMBER	=	an INTEGER variable or array element which is defined with the number of the unit that is connected to the file.
NAMED	=	a LOGICAL variable or array element which is defined with a true value if the file has a name.
NAME	=	a CHARACTER variable or array element which is defined with the name of the file.
EXIST	=	a LOGICAL variable or array element which is defined with a true value if the unit or file exists.

`OPENED` = a `LOGICAL` variable or array element which is defined with a true value if the unit or file is connected.

`ACCESS` = a `CHARACTER` variable or array element which is defined with either the value `SEQUENTIAL` or `DIRECT` depending on the access mode.

`SEQUENTIAL` = a `CHARACTER` variable or array element which is defined with the value `YES` or `NO` indicating whether the file can be connected for sequential access.

`DIRECT` = a `CHARACTER` variable or array element which is defined with the value `YES` or `NO` indicating whether the file can be connected for direct access.

`ACTION` = a `CHARACTER` variable or array element which is defined with either the value `READ`, `WRITE`, or `READWRITE` depending on the action mode.

`READ` = a `CHARACTER` variable or array element which is defined with the value `YES`, `NO`, or `UNKNOWN` indicating whether the file can be read from.

`WRITE` = a `CHARACTER` variable or array element which is defined with the value `YES`, `NO`, or `UNKNOWN` indicating whether the file can be written to.

`READWRITE` = a `CHARACTER` variable or array element which is defined with the value `YES`, `NO`, or `UNKNOWN` indicating whether the file can be read from *and* written to.

`FORM` = a `CHARACTER` variable or array element which is defined with either the value `FORMATTED` or `UNFORMATTED` depending on whether the file is connected for formatted or unformatted I/O.

`FORMATTED` = a `CHARACTER` variable or array element which is defined with the value `YES` or `NO` indicating whether the file can be connected for formatted I/O.

`UNFORMATTED`= a `CHARACTER` variable or array element which is defined with the value `YES` or `NO` indicating whether the file can be connected for unformatted I/O.

`ASYNCHRONOUS` = a `CHARACTER` variable or array element which is defined with the value `YES`, `NO`, or `UNDEFINED` indicating whether the connection allows asynchronous input/output.

BLANK	=	a CHARACTER variable or array element which is defined with either the value NULL or ZERO depending on how blanks are handled.
DECIMAL	=	a CHARACTER variable or array element which is defined with the value COMMA, POINT, or UNDEFINED.
DELIM	=	a CHARACTER variable or array element which is defined with the value APOSTROPHE, QUOTE, or NONE.
ENCODING	=	a CHARACTER variable or array element which is defined with the value UNKNOWN or UNDEFINED.
ERR	=	an error specifier as described above.
ID	=	an INTEGER expression which identifies a pending data transfer operation on the specified unit. This specifier is using along with the PENDING specifier.
IOLENGTH	=	IOLENGTH= value for an output item list.
IOSTAT	=	an I/O status specifier as described above.
IOMSG	=	an I/O message specifier as described above.
PAD	=	a CHARACTER variable or array element which is defined with the value YES or NO.
PENDING	=	a LOGICAL variable or array element which is used to determine the status of an asynchronous data transfer. If an ID specifier also appears, the PENDING variable is defined with the value false if the operation identified by ID has completed. If no ID specifier is present and all previously pending data transfer operations are complete, the PENDING variable is defined with the value false. Otherwise, the PENDING variable is defined with the value true.
POS	=	a INTEGER variable or array element which is defined with the number of the file storage unit immediately following the current position of a file connected for stream access.
POSITION	=	a CHARACTER variable or array element which is defined with the value ASIS, REWIND or APPEND.
RECL	=	an INTEGER variable or array element which is defined with the record length if the file is connected for direct access.

NEXTREC	=	an INTEGER variable or array element which is defined with the value of the next record number to be read or written.
SIGN	=	a CHARACTER variable or array element which is defined with the value PLUS, SUPPRESS or PROCESSOR_DEFINED.
SIZE	=	an INTEGER variable or array element which is defined with the size of the file in bytes.
STREAM	=	a CHARACTER variable or array element which is defined with the value YES, NO or UNKNOWN and indication whether the file can be connected for stream access.

Some of the specifiers may not be defined if a unit is not connected or a file does not exist. For example:

```
CHARACTER*20 FN,AM
LOGICAL OS
INTEGER RL
INQUIRE (UNIT=18, OPENED=OS, NAME=FN, ACCESS=AM, RECL=RL)
```

If unit 18 is not connected to a file, OS will be defined with a false value, but FN, AM, and RL will be undefined. If unit 18 is connected for sequential access, OS, FN, and AM will be defined appropriately, but record length is meaningless in this context, and RL will be undefined.

ENCODE AND DECODE STATEMENTS

The ENCODE and DECODE statements use internal files to effectively transfer data in internal form to character form, and vise versa. ENCODE can be thought of as writing the *list* of variables to the CHARACTER variable *char_form* with space padding if necessary, while DECODE reads the values of the variables from *char_form*. The forms of the ENCODE and DECODE statements are:

```
ENCODE (count,fmt,char_form[,IOSTAT=ios][,ERR=label]) [list]
DECODE (count,fmt,char_form[,IOSTAT=ios][,ERR=label]) [list]
```

where: *count* is the number of characters to convert to character form in the ENCODE statement. It is the number of characters to convert to internal form in the DECODE statement.

fmt is a format specifier described in the **Format Specifier** section near the beginning of this chapter.

char_form is a scalar variable or array which will hold the converted character form for the ENCODE statement. It holds the character form to be converted for the DECODE statement.

ios is an INTEGER*4 variable used to monitor error and end of file conditions. It is described in the **I/O Status Specifier** section near the beginning of this chapter.

label is a statement label at which execution will be continued in the event of an error during an ENCODE or DECODE conversion.

list is a list of variables separated by commas. These are in internal form.

The following example assigns the ASCII representation of the variables I and J to the character variable C. After the ENCODE statement, C equals " 123 456   ".

```
      CHARACTER*20 C
      I = 123
      J = 456
      ENCODE (20,100,C) I,J
100   FORMAT (2I4)
      END
```

GIVING A FORMAT SPECIFICATION

An explicit format specification may be given in either a FORMAT statement or in a character array or character expression. A FORMAT statement must be labeled so that it can be referenced by the data transfer statements (READ, WRITE, PRINT, etc.). The form of the FORMAT statement is:

```
FORMAT format_specification
```

When a format specification is given with a CHARACTER array or CHARACTER expression (CHARACTER variables, array elements, and substrings are simple CHARACTER expressions) it appears as a format specifier in the cilist of data transfer statements as described later in this chapter. An array name not qualified by subscripts produces a format specification which is the concatenation of all of the elements of the array. Leading and trailing blanks within the CHARACTER item are not significant.

A format specification is given with an opening parenthesis, an optional list of edit descriptors, and a closing parenthesis. A format specification may be given within a format specification; that is, it may be nested. When a format specification is given in this manner it is called a group specifier and can be given a repeat count, called the group repeat count, which is a positive INTEGER constant immediately preceding the opening parenthesis. The maximum level of nesting is 20.

The edit descriptors define the fields of a record and are separated by commas except between a P edit descriptor and an F, E, D, or G edit descriptor and before or after slash and colon edit descriptors (see below). The fields defined by edit descriptors have an associated width, called the field width.

An edit descriptor is either repeatable or nonrepeatable. Repeatable means that the edit descriptor is to be used more than once before going on to the next edit descriptor in the list. The repeat factor is given immediately before the edit descriptor as a positive integer constant.

The repeatable edit descriptors and their meanings are:

`Iw` and `Iw.m`	integer editing
`Bw` and `Bw.m`	binary editing
`Ow` and `Ow.m`	octal editing
`Zw` and `Zw.m`	hexadecimal editing
`Fw.d`	floating point editing
`Ew.d` and `Ew.dEe`	single precision scientific editing
`Dw.d`	double precision scientific editing
`ENw.d` and `Ew.dEe`	single precision engineering editing
`ESw.d` and `Ew.dEe`	single precision scientific editing
`Gw.d` and `Gw.dEe`	general floating point editing
`Lw`	logical editing
`A[w]`	character editing

w and *e* are nonzero, unsigned, integer constants and *d* and *m* are unsigned integer constants.

The nonrepeatable edit descriptors and their meanings are:

`'h1 h2 … hn'`	character string
`nHh1 h2 … hn`	Hollerith string
`nX`	skip positions
`Tc`, `TLc`, and `TRC`	tab to column
`kP`	set scale factor
`/`	start a new record
`:`	conditionally terminate I/O
`S`, `SP`, and `SS`	set sign control
`BZ` and `BN`	set blank control
`$` or `\`	suppress end of record
`Q`	return count of characters remaining in current record
`"h1 h2 … hn"`	character string

h is an ASCII character; *n* and *c* are nonzero, unsigned, integer constants; and *k* is an optionally signed integer constant.

FORMAT AND I/O LIST INTERACTION

During formatted data transfers, the I/O list items and the edit descriptors in the format specification are processed in parallel, from left to right. The I/O list specifies the variables that are transferred between memory and the fields of a record, while the edit descriptors dictate the conversions between internal and external representations.

The repeatable edit descriptors control the transfer and conversion of I/O list items. A repeatable edit descriptor or format specification preceded by a repeat count, *r*, is treated

as *r* occurrences of that edit descriptor or format specification. Each repeatable edit descriptor controls the transfer of one item in the I/O list except for complex items which require two `F`, `E`, `D`, or `G` edit descriptors. A complex I/O list item is considered to be two real items.

The nonrepeatable edit descriptors are used to manipulate the record. They can be used to change the position within the record, skip one or more records, and output literal strings. The processing of I/O list items is suspended while nonrepeatable edit descriptors are processed.

If the end of the format specification is reached before exhausting all of the items in the I/O list, processing starts over at the beginning of the last format specification encountered and the file is positioned to the beginning of the next record. The last format specification encountered may be a group specifier, if one exists, or it may be the entire format specification. If there is a repeat count in front of a group specifier it is also reused.

INPUT VALIDATION

Before numeric conversion from external to internal values using a format specification, input characters will be checked to assure that they are valid for the specified edit descriptor.

Valid input under the `I` edit descriptor:

digits: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
characters: `+, -`

Valid input under the `B` edit descriptor:

digits: `0, 1`

Valid input under the `O` edit descriptor:

digits: `0, 1, 2, 3, 4, 5, 6, 7`

Valid input under the `Z` edit descriptor:

digits: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
characters: `A, B, C, D, E, F, a, b, c, d, e, f`

Valid input under the `F`, `E`, `D`, and `G` edit descriptors:

digits: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
characters: `E, D, e, d, +, -, .`

The appearance of any character not considered valid for a particular edit descriptor will generate a runtime error. However, the appearance of a valid character in an invalid position will not result in an error. If the `ERR=` I/O specifier was present in the input statement generating the error, control will be transferred to the specified line number. If the `IOSTAT=` I/O specifier was present in the input statement generating the error, the specified variable will be defined with the error code.

INTEGER EDITING

The `B`, `O`, `Z` and `I` edit descriptors control the translation of character strings representing integer values to and from the appropriate internal formats.

I Editing

The `I`*w* and `I`*w*`.`*m* edit descriptors must correspond to an integer I/O list item. The field width in the record consists of *w* characters.

On input, the I/O list item will be defined with the value of the integer constant in the input field which may have an optional leading sign.

The output field consists of a string of digits representing the integer value which is right justified and may have a leading minus sign if the value is negative. If *m* is specified, the string will consist of at least *m* digits with leading zeros as required. The output field will always contain at least one digit unless an *m* of zero is specified in which case only blank characters will be output. If the specified field width is too small to represent the integer value, the field is completely filled with the asterisk character.

```
      WRITE (*,10) 12, -12, 12
10    FORMAT (2I4,I6.4)

  12 -12  0012
```

B, O, and Z Editing

The `B`, `O`, and `Z` edit descriptors are specified in the same manner as the `I` edit descriptor and perform bit editing on binary, octal, and hexadecimal fields respectively. The field width in the record consists of *w* characters. An input list item can be up to thirty-two bits in length and may have a `LOGICAL`, `INTEGER`, `REAL`, or `COMPLEX` data type. An output list value can be no longer than thirty-two bits in length and may have a `LOGICAL`, `INTEGER`, `REAL`, or `COMPLEX` data type. (Note that `COMPLEX` data requires two edit descriptors per data item).

On input, the I/O list item will be defined with the binary representation of the external value.

The output field consists of a string of characters representing the value and is right justified. If *m* is specified, the string will consist of at least *m* digits with leading zeros as

required. The output field will always contain at least one digit unless an *m* of zero is specified in which case only blank characters will be output.

```
     WRITE (*,10) 199, 199, 199
10   FORMAT (Z4,O7.6,B9)

  C7 000307 11000111
```

FLOATING POINT EDITING

The `F`, `E`, `D`, and `G` edit descriptors control the translation of character strings representing floating point values (`REAL`, `DOUBLE PRECISION`, and `COMPLEX`) to and from the appropriate internal formats. The edit descriptor must correspond to a floating point I/O list item. On input, the I/O list item will be defined with the value of the floating point constant in the input field.

A complex value consists of a pair of real values and consequently requires two real edit descriptors.

F Editing

The field width of the `F`*w*.*d* edit descriptor consists of *w* characters. The fractional portion, if any, consists of *d* characters. If the specified field width is too small to represent the value, the field is completely filled with the asterisk character.

The input field consists of an optional sign and a string of digits which can contain a decimal point. This may be followed by an exponent which takes the form of either a signed integer constant or the letter `E` or `D` followed by an optionally signed integer constant.

The output field consists of a minus sign if the value is negative and a string of digits containing a decimal point with *d* fractional digits. The value is rounded to *d* fractional digits and the string is right justified in the field. The position of the decimal point may be modified by the scale factor as described under the *k*`P` edit descriptor.

```
     WRITE (*,10) 1.23, -1.23, 123.0, -123.0
10   FORMAT (2F6.2,F6.1,F6.0)

  1.23 -1.23 123.0 -123.
```

E and D Editing

The field width of the `E`*w*.*d*, `E`*w*.*d*`E`*e*, and `D`*w*.*d* edit descriptors consists of *w* characters in scientific notation. *d* specifies the number of significant digits. If *e* is specified, the exponent contains *e* digits, otherwise, the exponent contains two digits for `E` editing and three digits for `D` editing.

The input field is identical to that specified for `F` editing.

The output field consists of a minus sign if the value is negative, a zero, a decimal point, a string of `d` digits, and an exponent whose form is specified in the table below. The value is rounded to `d` fractional digits and the string is right justified in the field. The position of the decimal point may be modified by the scale factor as described under the `kP` edit descriptor.

Edit Descriptor	Absolute value of Exponent	Form of Exponent
Ew.d	≤ 99	E±nn
Ew.d	100 - 999	±nnn
Ew.dEe	≤ $(10^e)-1$	E±n1n2…ne
Dw.d	≤ 99	D±nn
Dw.d	100 - 999	±nnn

```
     WRITE (*,10) 1.23, -1.23, -123.0E-6, .123D3
10   FORMAT (2E12.4,E12.3E3,D12.4)

  0.1230E+01 -0.1230E+01 -0.123E-003  0.1230D+03
```

EN Editing

The `ENw.d` and `ENw.dEe` edit descriptors produce engineering notation where the decimal exponent is divisible by 3 and the integer portion is greater than or equal to 1 and less than 1000. The input field is identical to that specified for `F` editing.

```
PRINT "(EN15.3)",5.79642, -100221.0, 0.008554

      5.796E+00
   -100.221E+03
      8.554E-03
```

ES Editing

The `ESw.d` and `ESw.dEe` edit descriptors produce engineering notation where the decimal integer portion is greater than or equal to 1 and less than 10. The input field is identical to that specified for `F` editing.

```
PRINT "(ES15.3)",5.79642, -100221.0, 0.008554

      5.796E+00
     -1.002E+05
      8.554E-03
```

G Editing

The G*w*.*d* and G*w*.*d*E*e* edit descriptors are similar to the F and E edit descriptors and provide a flexible method of accomplishing output editing.

The input field is identical to that specified for F editing.

The form of the output field depends on the magnitude of the value in the I/O list. F editing will be used unless the value of the item would cause the field width to be exceeded in which case E editing is used. In both cases, the field consists of *w* right justified characters.

Magnitude of N	Equivalent Conversion
N < 0.1	E*w*.*d*
0.1 < N < 1.0	F(*w*-4).*d*, 4X
1.0 < N < 10.0	F(*w*-4).(*d*-1), 4X
.	.
.	.
.	.
$10^{d-2} < N < 10^{d-1}$	F(*w*-4).1, 4X
$10^{d-1} < N < 10^{d}$	F(*w*-4).0, 4X
$N > 10^{d}$	E*w*.*d*[E*e*]

```
      WRITE (*,10) 1.0, 10.0, 100.0, 1000.0, 10000.0
10    FORMAT (5G10.4)

 1.000     10.00     100.0     1000.    0.1000E+05
```

P Editing

The *k*P edit descriptor is used to scale floating point values edited with the F, E, D, and G edit descriptors. k is called the <u>scale factor</u> and is given as an integer constant which may negative, positive, or zero. The scale factor starts at zero for each formatted I/O statement. If there is an exponent in the input field, the scale factor has no effect, otherwise the external value is equal to the internal value multiplied by 10^k.

For output with F editing, the effect of the scale factor is the same as described for input. For E and D editing, the scale factor is used to control decimal normalization of the output value. If *k* is negative, leading zeros are inserted after the decimal point, the exponent is reduced by *k* , and |*k*| significant digits are lost. If *k* is positive, the decimal point is moved to the right within the *d* significant digits, the exponent is reduced by *k* , and no significant digits are lost. The field width remains constant in all cases, meaning that $-d < k < d + 2$.

For output with G editing, the P edit descriptor has no effect unless the value cannot be represented with F editing. In that case the effect is the same as the E edit descriptor.

```
     WRITE (*,10) 1.23, 1.23, 1.23
10   FORMAT (1PF8.4,-1PF8.4,1PE12.4)

 12.3000   .1230  1.2300E+00
```

CHARACTER AND LOGICAL EDITING

The `A` and `L` edit descriptors control the translation of character strings representing `CHARACTER` and `LOGICAL` values to and from the appropriate internal formats.

A Editing

The `A[w]` edit descriptor is used to copy characters (bytes) to and from I/O list items. If present, *w* specifies the field width; otherwise the field width is the same as the length of the I/O list item. The only editing performed is to space fill or truncate for input and output respectively.

For input, when *w* is less than the length of the I/O list item, the characters from the field are left justified and space filled to the length of the item. When *w* is equal to or greater than the length of the I/O list item, the rightmost characters in the field are used to define the item.

For output, when *w* is less than or equal to the length of the I/O list item, the field will contain the leftmost *w* characters of the item. When *w* is greater than the length of the I/O list item, the item is right justified in the field with leading spaces added as necessary.

```
     WRITE (*,10) 'HELLO, WORLD ', ',', 'WORLD'
10   FORMAT (A5,A,A6)

HELLO, WORLD
```

L Editing

The `Lw` edit descriptor must correspond to a logical I/O list item. The field width in the record consists of *w* characters.

The input field consists of an optional decimal point and either the letter `T` (`.TRUE.`) or `F` (`.FALSE.`). Other characters may follow, but they do not take part in determining the `LOGICAL` value. The field may contain leading spaces.

The output field is right justified and contains either the letter `T` or `F` representing the values `.TRUE.` and `.FALSE.`, respectively.

```
      WRITE (*,10) .TRUE., .FALSE.
10    FORMAT (2L2)

 T F
```

SIGN CONTROL EDITING

The S, SP, and SS edit descriptors control the output of optional plus signs. Normally, a leading plus sign is not output for positive numeric values. The SP edit descriptor forces a plus sign to appear in the output field. The S and SS edit descriptors return the processing of plus signs to the default state of not being output.

```
      WRITE (*,10) 123, -123, 123.0, -123.0, 123.0
10    FORMAT (SP,2I5,2F7.1,SS,F7.1)

 +123 -123 +123.0 -123.0  123.0
```

BLANK CONTROL EDITING

The BN and BZ edit descriptors control the processing of blanks in numeric input fields which can be interpreted either as nulls or zeros. The default for an individual file connection is established with the "BLANK=" specifier. If the specifier does not appear in an OPEN statement blanks are treated as nulls. The BN edit descriptor causes blanks to be treated as nulls and the BZ edit descriptor causes blanks to be treated as zeros.

DECIMAL SYMBOL EDITING

The DC and DP edit descriptors control which character is used to separate the whole and fractional portions in the character strings representing real numbers. This character can be either a comma (DC) or a decimal point (DP). The default for an individual file connection is established with the "DECIMAL=" specifier in the OPEN statement used to establish the connection and confirmed or modified by the "DECIMAL=" specifier in a particular I/O statement.

POSITIONAL EDITING

The X, T, and / edit descriptors are used to control the position within the record and the position within the file.

X Editing

The *n*X edit descriptor moves the position within the record *n* characters forward. On input *n* characters are bypassed in the record. On output *n* blanks are output to the record.

```
      WRITE (*,10) -123, -123.0
10    FORMAT (I4,1X,F6.1)

-123 -123.0
```

T, TL, and TR Editing

On output, the entire record is first filled with spaces. The `T`*c*, `TL`*c*, and `TR`*c* edit descriptors are also used to move the position within the record, but in a non-destructive manner. This is called tabbing. Position means character position with the first character in the record being at position one. Changing the position within the record does change the length of the record.

The `T`*c* edit descriptor moves to absolute position *c* within the record. The `TL`*c* and `TR`*c* edit descriptors move to positions relative to the current position. `TR`*c* moves the position *c* characters to the right and `TL`*c* moves the position *c* characters to the left. *c* is a positive integer constant.

```
      WRITE (*,10) 89, 567,  23, 1, 4
10    FORMAT (T8,I2,TL5,I3,T2,I2,TL3,I1,TR2,I1)

123456789
```

Slash Editing

The / edit descriptor positions the file at the beginning of the next record. On input it skips the rest of the current record. On output it creates a new record at the end of the file.

The / edit descriptor can be used to skip entire records on input or to write empty records on output. Empty records in internal or direct access files are filled with blanks.

When the / edit descriptor is used with files connected for direct access it causes the record number to be increased and data transfer will be performed with that record.

```
      WRITE (*,10) (A, A=1.0,10.0)
10    FORMAT (5F5.1,/,5F5.1)

  1.0  2.0  3.0  4.0  5.0
  6.0  7.0  8.0  9.0 10.0
```

Dollar Sign and Backslash Editing

The `$` and `\` edit descriptors are interchangeable and are used to suppress the normal output of end of record characters in formatted records. When one of these edit descriptors appears in a format list, the output of end of record characters will be suppressed for the remainder of the I/O statement.

COLON EDITING

The : edit descriptor is used to terminate a formatted I/O statement if there are no more data items to process. For example, the : edit descriptor could be used to stop positional editing when there are no more items in the I/O list.

APOSTROPHE AND HOLLERITH EDITING

Apostrophe and Hollerith edit descriptors are used to copy strings of characters to the output record. These edit descriptors may only be used with the `WRITE`, `PRINT` and `TYPE` statements.

Apostrophe Editing

An apostrophe edit descriptor takes exactly the same form as a character constant as described in **The Fortran Program** chapter. The field width is equal to the length of the string.

```
      WRITE (*,10)
10    FORMAT ('APOSTROPHE',1X,'EDIT FIELDS')

APOSTROPHE EDIT FIELDS
```

H Editing

The *n*`H` edit descriptor takes exactly the same form as a Hollerith constant as described in the chapter **The Fortran Program**. The field width is equal to the positive integer constant, *n*, which defines the length of the Hollerith constant.

```
      WRITE (*,10)
10    FORMAT (15HHOLLERITH EDIT ,6HFIELDS)

HOLLERITH EDIT FIELDS
```

Q EDITING

The `Q` edit descriptor obtains the number of characters remaining in the current input record and assigns it the corresponding I/O list element. The I/O list element must be four byte integer variable. The `Q` edit descriptor has no effect on output except that the corresponding I/O list item is skipped.

```
      READ (*,10) I,(CHRS(J),J=1,I)
10    FORMAT (Q,80A1)
```

This example uses the `Q` edit descriptor to determine the number of characters in a record and then reads that many characters into the array `CHRS`.

VARIABLE FORMAT EXPRESSIONS

Variable format expressions allow you to use an arithmetic expression enclosed in angle brackets wherever an integer would normally be used in a FORMAT statement, with exception of the number of characters in a H (Hollerith constant) field. For example:

```
1000 FORMAT (I<N+4>)
```

The variable format expression, N+4, is scanned at runtime each time the FORMAT is used. If necessary, the expression is first converted to integer. The expression may contain function references. If side effects of the I/O statement cause the value of the expression to change, the value before the execution of the I/O statement is used.

LIST DIRECTED EDITING

List directed editing is indicated with an asterisk (*) as a format specifier. List directed editing selects editing for I/O list items appropriate to their data type and value. List directed editing treats one or more records in a file as a sequence of values delimited by value separators. A value separator is one or more blanks, a comma, a slash, or an end of record. Blanks can precede and follow the comma and slash separators. Except within a quoted character constant, multiple blanks and end of record characters are treated as a single blank character. An end of record occurring within a quoted character constant is treated as a null.

Tabs are expanded modulo eight by default; other tab sizes can be used by setting an environment variable. Refer to your system documentation for instructions on modifying this system dependent variable.

The values are either constants, nulls, or one of the forms:

*r*c*

*r**

where *r* is an unsigned, nonzero, integer constant. The first form is equivalent to *r* occurrences of the constant *c*, and the second is equivalent to *r* nulls. Null items are defined by having no characters where a value would be expected, that is, between successive separators or before the first separator in a record.

List Directed Input

A character value is a string of characters between value separators. If the string is quoted embedded blanks are significant and the value can span more than one record. The corresponding I/O list item is defined with the value as though a character assignment statement was performed; left justified and truncated or blank filled as necessary.

Any form suitable for an I edit descriptor can be used for list directed input of an INTEGER item.

Any form suitable for an `L` edit descriptor can be used for list directed input of a `LOGICAL` item. In particular, `.TRUE.` and `.FALSE.` are acceptable.

`DOUBLE PRECISION` and `REAL` input is performed with the effect of a `Fw.0` edit descriptor where *w* is the number of characters in the constant. The value can be in any form acceptable to the `F` edit descriptor.

A `COMPLEX` constant must have an opening parenthesis, a floating point constant as described above, a comma, another floating-point constant, and a closing parenthesis. Leading and trailing spaces are permitted around the comma. The first constant represents the real portion of the value and the second constant represents the imaginary portion.

Null values have no effect on the corresponding I/O list items; their definition status will not change.

A slash in the input record terminates a list directed input statement. Any unprocessed I/O list items will be left unchanged.

List Directed Output

With the exception of `CHARACTER` constants, all output items are separated by a single blank that is generated as part of the string.

`CHARACTER` output is performed using an `A` edit descriptor. There is no leading blank.

`LOGICAL` output is performed using an `L2` edit descriptor.

`INTEGER` output is performed using an `Iw` edit descriptor where *w* is one digit greater than the number of digits required to represent the value.

`DOUBLE PRECISION` and `REAL` output is performed using `1PG15.6E2` and `1PG24.15E3` edit descriptors respectively.

`COMPLEX` output consists of an opening parenthesis, the real portion of the value, a comma, the imaginary portion of the value, and a closing parenthesis. The numeric portions are formatted with descriptors that match the precision of the data as above.

NAMELIST DIRECTED EDITING

Namelist directed editing, an extension to standard Fortran, allows a number of variables to be treated as a group for the purpose of data transfer. Its use is restricted to formatted external files that have been connected for sequential access. Namelist directed editing selects editing for a namelist group member based on its type and value. Namelist directed editing treats one or more records as a group, where each group contains a series of group-member/value(s) combinations.

Namelist Directed Input

Namelist directed input reads external records until it finds the specified namelist group. It then assigns data to the specified group members in the order they are encountered. Group members which are not specified retain their previous values.

Namelist directed input has the following form:

```
$group member=value,[member=value, ...] $END
```

where: `$` is used to delimit the start and end of a particular group. The ampersand (`&`) can also be used for this purpose. The slash (`/`) can also be used to delimit the end of input for a given namelist group.

group is the symbolic name of a namelist previously defined in the program unit. The name cannot contain spaces or tabs.

member is a namelist defined variable. It may be a scalar, an array name, an array element name, a substring, or an array name with a substring. The member name cannot contain spaces or tabs. Subscript and substring specifiers must be integer constants. Use of symbolic (`PARAMETER`) constants is not allowed.

value is a constant, a list of constants, or a repetition of constants of the form *r*c*. Valid separators for value constants are spaces, tabs, and commas. A null value is specified by two consecutive commas, a leading comma, or a trailing comma. The form *r** indicates *r* null values. Character constants must be delimited by apostrophes or quotation marks. Occurrences of a character delimiter within the delimited string are represented by two consecutive occurrences of the delimiter. The end of record character is equivalent to a single space unless it occurs in a character constant, in which case it is ignored and the character constant is assumed to continue on the next record. Hollerith, binary, octal, and hexadecimal constants are not permitted.

`END` is an optional part of the terminating delimiter.

Group and member names are not case sensitive and are folded to upper case before use. Consider the following example:

```
INTEGER*4 INT,int
NAMELIST /NLIST/INT,int
...
READ (*,NML=NLIST)
```

where the input looks like:

```
$NLIST
INT = 12,
```

```
int = 15,
$END
```

Because namelist input is not case sensitive, execution of the read statement will cause `INT` to take on the value `15` and the value of `int` will be unchanged.

Conversion of external to internal representations is performed using the same editing as list directed input.

It is not necessary to assign values to all members of a namelist group. Group members not specified in the input retain their previous values. For namelist input of subscripted arrays and substring, only the values of the specified array elements and substrings are changed. Input containing group-members which are not actually members of the group is not permitted.

When namelist input is performed using an asterisk for the unit specifier, the group-name is written to standard out and the program waits for input from standard in.

An example of namelist directed input follows:

```
NAMELIST /WHO/NAME,CODE,NEW,RATIO,UNCHANGED
CHARACTER*8 NAME
INTEGER*4 CODE(4)
LOGICAL*4 NEW
REAL*4 RATIO,UNCHANGED
OPEN(10,FILE='INFO',FORM='FORMATTED',ACCESS='SEQUENTIAL')
READ(UNIT=10,NML=WHO)
```

where the input file test contains:

```
$WHO
NAME    = 'John Doe',
CODE(3) = 12,13,
NEW     = .TRUE.,
RATIO   = 1.5,
$END
```

The `NAMELIST` statement in this example creates a group named `WHO` with the members `NAME`, `CODE`, `NEW`, `RATIO`, and `UNCHANGED`. The `READ` statement then assigns values to the group members which are present in the input file. After execution of the `READ` statement, the variables `NAME`, `NEW`, and `RATIO` will have the values specified in the input. Because the array `CODE` has been subscripted, value assignment will begin with element three and continue until a new group-member name is encountered. As a result, elements `3` and `4` will be assigned the values `12` and `13` respectively. Elements `1` and `2` retain their previous values. Since the variable `UNCHANGED` does not appear in the input, it will retain whatever value it had before execution of the `READ` statement.

Namelist Directed Output

Namelist directed output transfers the current values of all members of a namelist group. The values are written in a form acceptable for namelist input. The group and group member names will be converted to upper case before being output. The order in which the values are written is determined by the order in which the group members appear in the `NAMELIST` statement. An example of namelist output follows:

```
INTEGER ONE,TWO
CHARACTER*10 ALPHA
NAMELIST /NLIST/ONE,TWO,ALPHA
ONE = 10
TWO = 20
ALPHA = 'ABCDEFGHIJ'
OPEN(10,FILE='TEST',ACCCESS='SEQUENTIAL',FORM='FORMATTED')
WRITE(UNIT=10,NML=NLIST)
...
```

The `WRITE` statement produces the following output:

```
$NLIST
ONE     = 10,
TWO     = 20,
ALPHA   = 'ABCDEFGHIJ',
$END
```

CHAPTER 9

Programs, Subroutines, and Functions

There are seven types of procedures available in Absoft Fortran: main programs, subroutines, external functions, internal procedures, statement functions, intrinsic functions, BLOCK DATA subprograms.

The main program is the entry point of a Fortran program. The compiler does not require that the main program occurs first in the source file, however, every Fortran program must have exactly one main program.

Subroutines and external functions are procedures that are defined outside of the program unit that references them. They may be specified either in separate Fortran subprograms or by means other than Fortran such as assembly language or the C programming language.

An internal procedure is a subroutine or function that is defined inside an external procedure. An internal procedure may only appear after a CONTAINS statement and must end with either END SUBROUTINE, END FUNCTION , or END. An internal procedure automatically **has access to all of the host's entities**, including variables, dummy arguments, and other internal procedures.

BLOCK DATA subprograms are nonexecutable procedures that are used to initialize variables and array elements in named COMMON blocks. There may be several block data subprograms in a Fortran program.

PROGRAMS

A main program is defined in the following manner:

```
[PROGRAM program-name]
  [specification-statements]
  [executable-statements]
[CONTAINS
  internal-procedures]
END [PROGRAM [program-name]]
```

The PROGRAM statement is not required to be present in a Fortran program. If it is present it must be the first line of the main program unit.

SUBROUTINES

A subroutine is an external procedure that is defined external to the program unit that references it and is specified in a subroutine subprogram. A subroutine may be referenced within any other procedure of the executable program.
A subroutine subprogram is defined in the following manner:

```
SUBROUTINE name[([dummy-arguments])]
  [specification-statements]
  [executable-statements]
[CONTAINS
  internal-procedures]
END [SUBROUTINE [name]]
```

where: *name* is a unique symbolic name that is used to reference the subroutine.

(*dummy-arguments*) is an optional comma separated list of variable names, array names, dummy procedure names, or asterisks that identifies the dummy arguments that are associated with the actual arguments in the referencing statement.

A subroutine is referenced with a CALL statement that has the form:

```
CALL sub [([actual-arguments)]
```

where: *sub* is the symbolic name of a subroutine or dummy procedure.

([*actual-arguments*) is the comma separated list of actual arguments that are associated with the arguments in the SUBROUTINE statement.

Subroutine Arguments

The argument lists of CALL and SUBROUTINE statements have a one to one correspondence; the first actual argument is associated with the first dummy argument and so on. The actual arguments in a CALL statement are assumed to agree in number and type with the dummy arguments declared in the SUBROUTINE statement. No type checking is performed by the compiler or the run time system to insure that this assumption is followed.

The addresses of labeled statements may be passed to subroutines by specifying the label preceded by an asterisk in the actual argument list and specifying an asterisk only in the corresponding position in the dummy argument list of the SUBROUTINE statement. This allows you to return to a location in the calling procedure other than the statement that immediately follows the CALL statement (see RETURN below).

Dummy procedure names allow you pass the names of procedures to other subprograms. The dummy procedure name can then be referenced as though it were the actual name of an external procedure.

FUNCTIONS

A function returns a value to the point within an expression that references it. An external function is specified in a separate procedure called a function subprogram. A statement function is defined in a single statement within a program unit and is local to that program unit. Intrinsic functions are library procedures provided with the Fortran environment and are available to any program unit in an executable program. A function name may not be used on the left side of an equals sign except for an external function name and then only within the program unit that defines it.

A function reference is made in the form of an operand in an expression. The function name is given with an argument list enclosed in parentheses. The parentheses must be used even if there are no arguments to the function so that the compiler can determine that a function reference is indeed being made and not simply a reference to a variable.

External Functions

An external function may be referenced within any other procedure in an executable program. Character functions must be declared with integer constant lengths so that the compiler can determine the size of the character value that will be returned.

The form of a function subprogram declaration is:

```
[type [*len]] FUNCTION func ([arg] [,arg]...) [RESULT(r)]
```

where: `func` is a unique symbolic name that is used to reference the function.

`([arg] [,arg]...)` is an optional list of variable names, array names, or dummy procedure names that identifies the dummy arguments that are associated with the actual arguments in the referencing statement.

`RESULT(r)` is an optional symbolic name that is used to record the function result. If this argument is present, the symbolic name `func` must not appear in any declaration statement.

As indicated, the function can be given an optional type and length attribute. This can be done either explicitly in the `FUNCTION` statement or in a subsequent type statement, or implicitly following the data typing rules described in **The Fortran Program** chapter. Note that an `IMPLICIT` statement may change the data type and size.

When a `CHARACTER` function is given a length attribute of `*(*)` it assumes the size established in the corresponding character declaration in the referencing program unit.

The symbolic name used to define the function or the symbolic name specified in the `RESULT` argument must be assigned a value during the execution of the function subprogram. It is the value of this variable that is returned when a `RETURN` or `END` statement is executed.

A function (external or internal) can return a pointer:

```
integer, dimension(10) :: m = (/123,  87,  55, 203,  88, &
                                908,  13, 792,  66, 118/)
integer, dimension(:), pointer :: p

p => pick(m, 100)
print *,p

contains

function pick(m, limit)
  integer, dimension(:), pointer :: pick
  integer, dimension(:) :: m
  j = 0
  do i=1,size(m)
    if (m(i) >= limit) j = j+1
  end do
  allocate (pick(j))
  j = 0
  do i=1,size(m)
    if (m(i) >= limit) then
      j = j+1
      pick(j) = m(i)
    end if
  end do
end function pick

end
```

```
 123  203  908  792  118
```

Statement Functions

A statement function is specified with a single statement that may appear only after the declaration section and before the executable section of the program unit in which it is to be used. A statement function is defined in the following manner:

> *`func ([arg[,arg]...]) = e`*

where: *`func`* is the name that is used to reference the function.

> `([`*`arg`*`[,`*`arg`*`]...])` is the dummy argument list, and *`e`* is an expression using the arguments from the dummy argument list.

The dummy argument names used in the statement function argument list are local to the statement function and may be used elsewhere in the program unit without conflict.

A statement function statement must not contain a forward reference to another statement function. The compilation of a statement function removes the symbolic name of the function from the list of available names for variables and arrays within the program unit in which it is defined. Any variable or array that is defined in a program unit may not be redefined as a statement function.

`CHARACTER` statement functions may not use the `*(*)` length specifier.

Intrinsic Functions

Intrinsic functions contained in the math library do not follow the typing rules for user defined functions and cannot be altered with an `IMPLICIT` statement. The types of these functions and their argument list definitions appear in the next chapter, **Intrinsic Procedures.**

RECURSION

Normally, a function or subroutine may not reference itself, either directly or indirectly. Such a reference would be recursive. The `RECURSIVE` keyword allows such direct or indirect self referencing statements.

```
RECURSIVE FUNCTION
RECURSIVE SUBROUTINE
```

The function N! = Nx(N-1)…2x1 lends itself to recursive programming.

```
INTEGER RECURSIVE FUNCTION Factorial(n) RESULT(r)
  IF (n == 1) THEN
    r = 1
  ELSE
    r = n*Factorial(n-1)
  END IF
END FUNCTION Factorial
```

PURE PROCEDURES

The `PURE` keyword is used with functions and subroutines to indicate that the procedure has no side effects. A pure procedure cannot perform I/O operations, contains no variable with a `SAVE` attribute, does not alter variables accessed by host or use association, does not contain a `STOP` statement, and, if a function, does not alter any dummy argument that does not have a pointer attribute.

```
PURE FUNCTION AREA(T)
```

ELEMENTAL PROCEDURES

Elemental intrinsic procedures are those with scalar dummy arguments that return scalar results. They can be called with array arguments to return an array result of the same shape. The ELEMENTAL keyword can used to produce non-intrinsic procedures with the same characteristics.

```
ELEMENTAL FUNCTION AREA(T)
```

An elemental procedure is automatically a pure procedure. All dummy arguments must be scalar without a POINTER attribute.

ENTRY STATEMENT

The ENTRY statement may only be used within subroutine and function subprograms and provides for multiple entry points into these procedures. The form of an ENTRY statement is the same as that for a SUBROUTINE statement except that the keyword ENTRY is used. An ENTRY statement appearing within a FUNCTION subprogram may appear in a type statement. An ENTRY statement may not occur within any block structure (DO, IF, or CASE).

In a function subprogram, a variable name that is used as the entry name must not appear in any statement that precedes the appearance of the entry name except in a type statement. All function and entry names in a function subprogram share an equivalence association.

Entry names used in character functions must have a character data type and the same size as the name of the function itself.

RETURN STATEMENT

The RETURN statement ends execution in the current subroutine or function subprogram and returns control of execution to the referencing program unit. The RETURN statement may only be used in function and subroutine subprograms. Execution of a RETURN statement in a function returns the current value of the function name variable to the referencing program unit. The RETURN statement is given in the following manner:

```
RETURN [e]
```

where: *e* is an INTEGER expression allowed only in subroutine RETURN statements and causes control to be returned to a labeled statement in the calling procedure associated with an asterisk in the dummy argument list. The first alternate return address corresponds to the first asterisk, the second return address to the second asterisk, etc. If the value of *e* is less than one or greater than the number of asterisks, control is returned to the statement immediately following the CALL statement.

PASSING PROCEDURES IN DUMMY ARGUMENTS

When a dummy argument is used to reference an external function, the associated actual argument must be either an external function or an intrinsic function. When a dummy argument is associated with an intrinsic function there is no automatic typing property. If a dummy argument name is also the name of an intrinsic function then the intrinsic function corresponding to the dummy argument name is removed from the list of available intrinsic functions for the subprogram.

If the dummy argument is used as the subroutine name of a `CALL` statement then the name cannot be used as a variable or a function within the same program unit.

PASSING RETURN ADDRESSES IN DUMMY ARGUMENTS

If a dummy argument is an asterisk, the compiler will assume that the actual argument is an alternate return address passed as a statement label preceded by an asterisk. No check is made by the compiler or by the run time system to insure that the passed parameter is in fact a valid alternate return address.

COMMON BLOCKS

A `COMMON` block is used to provide an area of memory whose scoping rules are greater than the current program unit. Because association is by storage offset within a known memory area, rather than by name, the types and names of the data elements do not have to be consistent between different procedures. A reference to a memory location is considered legal if the type of data stored there is the same as the type of the name used to access it. However, the compiler does not check for consistency between different program units and `COMMON` blocks.

The total amount of memory required by an executable program can be reduced by using `COMMON` blocks as a sharable storage pool for two or more subprograms. Because references to data items in common blocks are through offsets and because types do not conflict across program units, the same memory may be remapped to contain different variables.

BLOCK DATA

A `BLOCK DATA` statement takes the following form:

```
BLOCK DATA [sub]
```

where: *sub* is the unique symbolic name of the block data subprogram.

There may be more than one named `BLOCK DATA` subprogram in a Fortran program, but only one unnamed block data subprogram.

Only `COMMON`, `SAVE`, `DATA`, `DIMENSION`, `END`, `EQUIVALENCE`, `IMPLICIT`, `PARAMETER`, and type declaration statements may be used in a `BLOCK DATA` subprogram.

CHAPTER 10

INTRINSIC PROCEDURES

This chapter is devoted to summarizing and categorizing Fortran's intrinsic procedures: intrinsic functions and intrinsic subroutines. This summary has ten categories of procedure, each with certain similar characteristics, and ends with a concise alphabetical listing of all intrinsic procedures and their arguments. Each intrinsic procedure is described more fully, in alphabetical order, in the last section of this chapter.

INTRINSIC PROCEDURE SUMMARY

numeric inquiry functions

digits	significant digits (e.g., bits) for a given integer or real kind
epsilon	a small value (small compared to 1) for a given real kind
exponent	the exponent value for a given real value
fraction	the fractional part of a given real value
huge	the largest value representable for a given real or integer kind
minexponent	the minimum exponent value for a given real kind
maxexponent	the maximum exponent value for a given real kind
nearest	the processor value nearest to a given real value, in a given direction
precision	the decimal precision of a given real or complex kind
radix	numeric base (typically binary) for a given real or integer kind
rrspacing	reciprocal of the relative spacing near a given real value
range	the decimal exponent range of a given numeric kind
scale	change the exponent of a given real value by a specified amount
set_exponent	set the exponent of a given real value to the specified amount
spacing	the absolute spacing near a given real value
tiny	the smallest positive value representable for a given real kind

array inquiry functions

allocated	true if the given array is currently allocated (false otherwise)
lbound	lower bound(s) of a given array or a given dimension of an array
shape	the number of elements in each dimension of a given array
size	the size (total number of elements) of a given array
ubound	upper bound(s) of a given array or a given dimension of an array

miscellaneous inquiry functions

associated	true if the given pointer is currently allocated (false otherwise)
bit_size	the number of bits (for bit computations) in a given integer kind
command_argument_count	the number command line arguments
int_ptr_kind	the INTEGER KIND that will hold an address
is_iostat_end	Test IOSTAT value for end-of-file
is_iostat_eor	Test IOSTAT value for end-of-record
kind	the value of the kind type parameter of a given data entity
len	the number of characters in a given string value
present	true if there is an actual argument for a given optional dummy argument
selected_char_kind	the character kind for a given character type
selected_int_kind	the integer kind for a given integer decimal range
selected_real_kind	the real kind for a given decimal precision and range
storage_size	the storage size in bits for a data object

conversion functions

achar	the character in the specified position of the ASCII character set
aimag	the imaginary part of a given complex value
aint	a given real value truncated to an integer (result is still real)
anint	a given real value rounded to the nearest integer (result is still real)
char	the character in the specified position of the processor character set
cmplx	the complex value of a given single or pair of integer or real values
conjg	the complex conjugate of a given complex value
dble	the double precision value of a given numeric value of any type
iachar	position of the specified character in the ASCII character set
ibits	the specified substring of bits of a given integer value
ichar	position of the specified character in the processor character set
int	the (truncated) integer value of a given numeric value of any type
logical	the logical value of specified kind for a given logical value
nint	the (rounded) integer value of a given real value
real	the real value of a given numeric value of any type and kind
transfer	conversion to a specified type without change in the “bit pattern”

numeric computation functions

abs	the absolute value of a given numeric value of any type
acos	the arc cosine (radians) of a given real value
acosh	the inverse hyperbolic cosine
asin	the arc sine (radians) of a given real value
asinh	the inverse hyperbolic sine
atan	the arc tangent (radians) of a given real value
atanh	the inverse hyperbolic tangent
atan2	the angle (radians) of given real and imaginary components

bessel_j0	Bessel function of the 1st kind, order 0
bessel_j1	Bessel function of the 1st kind, order 1
bessel_jn	Bessel functions of the 1st kind
bessel_y0	Bessel function of the 2st kind, order 0
bessel_y1	Bessel function of the 2st kind, order 1
bessel_yn	Bessel functions of the 2st kind
ceiling	the smallest integer not less than a given real value
cos	the cosine of a given real or complex value (given value in radians)
cosh	the hyperbolic cosine of a given real value
dim	maximum of: zero and the difference of two real or integer values
dot_product	the dot product of two given vectors of numeric or logical type
dprod	the double precision product of two single precision real values
erf	the error function
erfc	the complementary error function
erfc_scaled	the scaled complementary error function
exp	the natural exponential function (real or complex)
floor	the greatest integer not greater than a given real value
hypot	the Euclidean distance function
gamma	the Gamma function
log	the natural logarithm function (real or complex)
log10	the logarithm to the base 10 of a given real value greater than zero
log_gamma	the logarithm of the absolute value of the gamma function
matmul	matrix multiplication of two given numeric or logical matrices
max	the maximum of a set of given integer or real values
min	the minimum of a set of given integer or real values
mod	the remainder function, having the sign of the first given value
modulo	the remainder function, having the sign of the second given value
sign	apply a given sign to a given integer or real value
sin	the sine of a given real or complex value (given value in radians)
sinh	the hyperbolic sine of a given real value
sqrt	the square root of a given real or complex value greater than zero
tan	the tangent of a given real value (given value in radians)
tanh	the hyperbolic tangent of a given real value

character computation functions

adjustl	left-justify a given string value in the same-width field
adjustr	right-justify a given string value in the same-width field
index	find the location of a given substring in a given string value
len_trim	length of a given string after trailing blanks have been removed
lge	greater than or equal to ASCII comparison of two given strings
lgt	greater than ASCII comparison of two given string values
lle	less than or equal to ASCII comparison of two given string values
llt	less than ASCII comparison of two given string values
repeat	concatenate several copies of a given string value
scan	search a given string value for any of a given set of characters
trim	remove trailing blank characters from a given string value
verify	position of a character in a string that is not one of a given set

bit computation functions

bge	bitwise greater than or equal
bgt	bitwise greater than
ble	bitwise less than or equal
blt	bitwise less than
btest	the bit value of a specified position in a given integer value
dshiftl	combined left shift
dshiftr	combined right shift
iand	bit-by-bit AND of two given integer values
ibclr	set to zero the bit in a specified position in a given integer value
ibset	set the bit in a specified position to the specified (0 or 1) value
ieor	bit-by-bit exclusive-OR of two given integer values
ior	bit-by-bit OR of two given integer values
ishft	end-off shift of the bits in a specified integer value
ishftc	circular shift of the bits in a specified integer value
leadz	the number of leading zero bits
maskl	left justified mask
maskr	right justified mask
merge_bits	merge bits under mask
not	bit-by-bit complement of a given integer value
popcnt	the number one bits
poppar	parity
shifta	right shift with fill
shiftl	left shift
shiftr	right shift
trailz	the number of trailing zero bits

array computation functions

all	true if all of the elements of a given logical array are true
any	true if any of the elements of a given logical array are true
count	the number of true elements in a given logical array
cshift	circular shift the elements of a given array of any type
eoshift	end-off shift the elements of a given array of any type
maxloc	a rank-one array locating the maximum element of a given array
maxval	the maximum element value of a given integer or real array
merge	combines (merges) two arrays under control of a mask
minloc	a rank-one array locating the minimum element of a given array
minval	the minimum element value of a given integer or real array
pack	packs elements of an array into a vector, under control of a mask
product	the product of all elements of a given numeric array of any type
reshape	reshapes a given rank-one array into the specified array shape
spread	replicates an array along a new dimension
sum	the sum of all elements of a given numeric array of any type
transpose	the matrix transpose of a given rank-two array of any type
unpack	unpacks a vector into elements of an array, under control of a mask

intrinsic subroutines

cpu_ time	returns processor time
date_and_time	returns date and time information in several formats
execute_command_line	execute a command line
get_command	the command line
get_command_argument	a command line argument
get_environment_variable	an environment variable
mvbits	copies a sequence of bits between given integer values
random_number	returns one or more pseudorandom numbers
random_seed	allows setting of the random number generator seed value
system_clock	returns various data from the processor's real-time clock

pointer functions

null	returns a disassociated pointer

ALPHABETICAL LISTING OF INTRINSIC PROCEDURES

generic procedure	optional arguments	specific name	specific argument type
abs (a)		abs (a)	real
		cabs (a)	complex
		dabs (a)	double precision
		iabs (a)	integer
achar (i)			
acos (x)		acos (x)	real
		dacos (x)	double precision
acosh (x)			real, complex
adjustl (string)			
adjustr (string)			
aimag (z)		aimag (z)	complex
aint (a, kind)	kind	aint (a)	real
		dint (a)	double precision
all (mask, dim)	dim		
allocated (array)			
anint (a, kind)	kind	anint (a)	real
		dnint (a)	double precision
any (mask, dim)	dim		
asin (x)		asin (x)	real
		dsin (x)	real precision
asinh (x)			real, complex
associated (pointer, target)	target		
atan (x)		atan (a)	real
		datan (a)	double precision
atanh (x)			real, complex
atan2 (y, x)		atan2 (a)	real
		datan2 (a)	double precision
bessel_j0 (x)			
bessel_j1 (x)			
bessel_jn (n,x)			

generic procedure	optional arguments	specific name	specific argument type
bessel_j1 (n1,n2,x)			
bessel_y0 (x)			
bessel_y1 (x)			
bessel_yn (n,x)			
bessel_yn (n1,n2,x)			
bge(i,j)			
bgt(i,j)			
ble(i,j)			
blt(i,j)			
bit_size (i)			
btest (i, pos)			
ceiling (a, kind)	kind		
char (i, kind)	kind		
cmplx (x, y, kind)	y, kind		
command_argument_count()			
conjg (z)		conjg (x)	complex
cos (x)		cos (x)	real
		ccos (x)	complex
		dcos (x)	double precision
cosh (x)		cosh (x)	real
		dcosh (x)	double precision
count (mask, dim)	dim		
cshift (array, shift, dim)	dim		
date_and_time (date, time, zone, values)	date, time, zone, values		
dble (a)			
digits (x)			
dim (x, y)		dim (x,y)	real
		idim (x,y)	integer
		ddim (x,y)	double precision
dot_product (vector_a, vector_b)			
dprod (x, y)			
dshiftl(i, j, shift)			
dshiftr(i, j, shift)			
eoshift (array, shift, boundary, dim)	boundary, dim		
epsilon (x)			
erf (x)			
erfc (x)			
erfc_scaled (x)			
execute_command_line (command, wait, exitstat, cmdstat, cmdmsg)	wait, exitstat, cmdstat, cmdmsg		
exp (x)			
exponent (x)			
floor (a, kind)	kind		
fraction (x)			
gamma (x)			
get_command(command, length, status)	command, length, status		
get_command_argument(number, value, length, status)	value, length, status		
get_environment_variable(name, value, length, status, trim_name)	value, length, status, trim_name		

generic procedure	optional arguments	specific name	specific argument type
huge (x)			
hypot (x, y)			
iachar (c)			
iand (i, j)			
ibclr (i, pos)			
ibits (i, pos, len)			
ibset (i, pos)			
ichar (c)			
ieor (i, j)			
index (string, substring, back, kind)	back, kind	index (string, substring)	character default character
int (a, kind)	kind		
int _ptr_kind()			
ior (i, j)			
ishft (i, shift)			
ishftc (i, shift, size)	size		
is_iostat_end()			
is_iostat_eor()			
kind (x)			
leadz (i)			
lbound (array, dim)	dim		
len (string)		len (string)	character
len_trim (string)			
lge (string_a, string_b)			
lgt (string_a, string_b)			
lle (string_a, string_b)			
llt (string_a, string_b)			
log (x)		alog (x)	real
		clog (x)	complex
		dlog (x)	double precision
log10 (x)		alog10 (x)	real
		dlog10 (x)	double precision
log_gamma (x)			
logical (l, kind)	kind		
maskl (l, kind)	kind		
maskr (l, kind)	kind		
matmul (matrix_a, matrix_b)			
max (a1, a2, a3, ...)	a3, ...	max0 (a1,a2,…)	integer
		amax1 (a1,a2,…)	real
		dmax1 (a1,a2,…)	double precision
maxexponent (x)			
maxloc (array, dim, mask)	dim, mask		
maxval (array, dim, mask)	dim, mask		
merge (tsource, fsource, mask)			
merge_bits (i, j, mask)			
min (a1, a2, a3, ...)	a3, ...	min0 (a1,a2,…)	integer
		amin1 (a1,a2,…)	real
		dmin1 (a1,a2,…)	double precision
minexponent (x)			
minloc (array, dim, mask)	dim, mask		
minval (array, dim, mask)	dim, mask		
		dmod (a, p)	double precision

generic procedure	optional arguments	specific name	specific argument type
mod (a, p)		mod (a, p)	integer
		amod (a, p)	real
modulo (a, p)			
mvbits (from, frompos, len, to, topos)			
nearest (x, s)			
new_line(a)			character
nint (a, kind)	kind	nint (a)	real
		idnint (a)	double precision
null (mold)	mold		
pack (array, mask, vector)	vector		
precision (x)			
present (a)			
popcnt (i)			
poppar (i)			
product (array, dim, mask)	dim, mask		
radix (x)			
random_number (harvest)			
random_seed (size, put, get)	size, put, get		
range (x)			
real (x, kind)	kind		
repeat (string, ncopies)			
reshape (source, shape, pad, order)	pad, order		
rrspacing (x)			
scale (x, i)			
scan (string, set, back)	back		
selected_int_kind (r)			
selected_real_kind (p, r)	p, r		
set_exponent (x, i)			
shape (source)			
shifta (i, shift)			
shiftl (i, shift)			
shiftr (i, shift)			
sign (a, b)		sign (a, b)	real
		dsign (a, b)	double precision
		isign (a, b)	integer
sin (x)		sin (x)	real
		csin (x)	complex
		dsin (x)	double precision
sinh (x)		sinh (x)	real
		dsinh (x)	double precision
size (array, dim)	dim		
spacing (x)			
spread (source, dim, ncopies)			
sqrt (x)		sqrt (x)	real
		csqrt (x)	complex
		dsqrt (x)	double precision
storage_size (a, kind)	kind		
sum (array, dim, mask)	dim, mask		
system_clock (count, count_rate, count_max)	count, count_rate, count_max		
tan (x)		tan (x)	real
		dtan (x)	double precision

generic procedure	optional arguments	specific name	specific argument type
tanh (x)		tanh (x)	real
		dtanh (x)	double precision
tiny (x)			
transfer (source, mold, size)	size		
transpose (matrix)			
trailz (i)			
trim (string)			
ubound (array, dim)	dim		
unpack (vector, mask, field)			
verify (string, set, back)	back		

INTRINSIC PROCEDURE DETAILS

The intrinsic procedures are described in detail in this section. A "pseudo" interface block, without the **interface ... end interface** bracketing keywords, describes the interface of each procedure; the semantics is described in comments in the interface, often augmented by text following the interface. Constraints and other relevant information are also included, either in the interface comments or in the following text.

Most of the intrinsic procedures are generic over the various kinds of the argument type - for example, **sqrt** is generic for single, double precision, and quad precision real arguments. Unless explicitly mentioned otherwise, each single-argument intrinsic procedure is generic in this sense, with the result kind being the same as the argument kind. Similarly, each intrinsic function with one argument plus a **kind** argument is generic in this sense, but with the result kind as specified by the **kind** argument.

Intrinsic procedures may be classified as either elemental or transformational (most are elemental). If an intrinsic procedure is elemental the interface starts with the keyword **elemental**; otherwise that procedure is transformational. (In a call to an elemental function with two or more arguments, the actual arguments must be conformable; but note that a scalar is conformable with any array.) Similarly, some intrinsic functions are identified as inquiry functions with the keyword **inquiry**; actual arguments to inquiry functions need not be defined.

All intrinsic function arguments are **intent(in)** and so the intent for these arguments is not explicitly given in the interface; however, the argument intent is explicitly specified for each argument of the five intrinsic subroutines. The argument names can be used as actual argument keywords.

Many of the intrinsic procedures take array arguments of any rank. These arguments are shown as rank one **(:)** in the following interfaces, and the descriptions identify which arguments may be generic over rank and the resulting meaning of the different ranks.

```
abs (a)
  elemental function abs(a)   ! the |a|
    real :: abs               ! or integer if a is of type integer
    real :: a                 ! or type integer or type complex
  end function
```

If **a** is complex with value (x,y), **abs** returns an approximation to `sqrt(a**2+b**2)`.

```
achar (i)
  elemental function achar(i) ! the ith ascii character
    character :: achar        ! the result kind is kind('a')
    integer :: i
end function
```

Note that `achar(iachar(x))` is `x` for any character `x` of default kind represented by the processor.

```
acos (x)
  elemental function acos(x)  ! the arccosine (inverse cosine)
    real :: acos
    real :: x                 ! |x| ≤ 1 or of type complex
  end function
```

The result has a value equal to a processor-dependent approximation to arccos(x), expressed in radians. It lies in the range `0 ≤ acos(x) ≤ pi`. If complex, the real portion lies in the range `0 ≤ acos(x) ≤ pi`.

```
acosh (x)
  elemental function acosh(x) ! the inverse hyperbolic cosine
    real :: acosh
    real :: x                 ! real or complex
  end function
```

```
adjustl (string)
  elemental function adjustl(string)     ! remove leading blanks
    character(len(string)) :: adjustl    ! (the same number of trailing
                                         ! blanks added)
    character(*) :: string
  end function
```

```
adjustr (string)
  elemental function adjustr(string)      ! remove trailing blanks
    character(len(string)) :: adjustr     ! (the same number of leading
                                          ! blanks added)
    character(*) :: string
  end function
```

```
aimag (z)
  elemental function aimag(z) ! imaginary part of z
    real :: aimag             ! if z = (x, y), aimag is y
    complex :: z
  end function
```

```
aint (a, kind)
  elemental function aint(a,kind)   ! truncate a
    real(kind) :: aint              ! if kind is absent the result kind
                                    ! is kind(a)
    real :: a                       ! may be any kind
    integer, optional :: kind       ! if present, must be a scalar
                                    ! initialization expression
  end function
```

If |a|<1, **aint (a)** has the value 0; if a>1, **aint (a)** has the sign of **a** and a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of **a**.

```
all (mask, dim)
  function all(mask,dim)       ! returns true if all of mask (along dim)
                               ! is true or if mask has zero size
    logical :: all             ! all is an array if dim is present and
                               ! mask rank > 1
    logical :: mask(:)         ! may be any kind, any rank
    integer, optional :: dim   ! if present, 1≤dim≤n, where n is rank of
                               ! mask
  end function
```

The result is scalar if **dim** is absent or **mask** has rank one; otherwise, the result is an array of rank n-1 and of shape (d1,d2,...,ddim 1,ddim+1,...,dn) where (d1,d2,...,dn) is the shape of **mask**.

```
allocated (array)
  inquiry function allocated(array) ! check allocation status of
                                    ! argument
    logical :: allocated            ! true if array is currently
                                    ! allocated; false otherwise
    real :: array(:)                ! array can be any type and any rank
  end function
```

The actual argument for **array** must be an allocatable array with defined allocation status.

```
anint (a, kind)
  elemental function anint(a,kind)  ! integer value nearest a
  real(kind) :: anint               ! if kind is absent the result kind
                                    ! is kind(a)
  real :: a
  integer, optional :: kind         ! if present, must be a scalar
                                    ! initialization expression
end function
```

If a>0 **anint(a)** is **aint (a + 0.5)**; if a≤0 **anint (a)** is **aint (a - 0.5)**.

```
any (mask, dim)
  function any(mask,dim)       ! returns true if any of mask (along dim)
                               ! is true, false if mask has zero size
    logical :: any             ! any is an array if dim is present and
                               ! mask rank > 1
    logical :: mask(:)         ! may be any kind, any rank
    integer, optional :: dim   ! if present, 1≤dim≤n, where n is rank of
                               ! mask
  end function
```

The result is scalar if **dim** is absent or **mask** has rank one; otherwise, the result is an array of rank n-1 and of shape (d1,d2,...,ddim-1,ddim+1,...,dn) where (d1,d2,...,dn) is the shape of **mask**.

```
asin (x)
  elemental function asin(x)  ! the arcsine (inverse sine)
    real :: asin
    real :: x                 ! x ≤ 1 or of type complex
  end function
```

The result has a value equal to a processor-dependent approximation to arcsin(x), expressed in radians. It lies in the range -pi/2 ≤ asin(x) ≤ pi/2. If complex, the real portion lies in the range -pi/2 ≤ asin(x) ≤ pi/2.

```
asinh (x)
  elemental function asinh(x) ! the inverse hyperbolic sine
    real :: asinh
    real :: x                 ! real or complex
  end function
```

```
associated (pointer, target)
  inquiry function associated(pointer,target)   ! check association
                                                ! status of argument
    logical :: associated                       ! true if pointer is
                                                ! currently associated,
                                                ! else false
    real, pointer :: pointer(:)                 ! pointer can be any
                                                ! type, any rank
  real, optional :: target(:)                   ! target can be any
                                                ! type, any rank
  end function
```

The actual argument for **pointer** must be a pointer with defined pointer association status. The actual argument for **target**, if present, must be either a target or a pointer with defined pointer association status. If **target** is absent, the result is true if **pointer** is currently associated with a target and false otherwise. If **target** is present and is a target, the result is true if **pointer** is currently associated with **target** and false if it is not. If **target** is present and is a pointer, the result is true if both **pointer** and **target** are currently associated with the same target, and false otherwise.

```
atan (x)
  elemental function atan(x)  ! the arctangent (inverse tangent)
    real :: atan
    real :: x                 ! may be complex
  end function
```

The result has a value equal to a processor-dependent approximation to arctan(x), expressed in radians, that lies in the range -pi/2 ≤ atan(x) ≤ pi/2.

```
atanh (x)
  elemental function atanh(x) ! the inverse hyperbolic tangent
    real :: atanh
    real :: x                 ! real or complex
  end function
```

```
atan2 (y, x)
  elemental function atan2(x) ! the arctangent (inverse tangent)
    real :: atan2             ! of the nonzero complex number (x, y)
    real :: y
    real :: x
  end function
```

The result has a value equal to a processor-dependent approximation to the arctan(y/x), expressed in radians that lies in the range –pi ≤ atan2(y,x) ≤ pi. If y>0, the result is positive. If y=0, the result is zero if x>0 and pi if x<0. If y<0, the result is negative. If x=0, the absolute value of the result is pi/2.

```
bessel_j0 (x)
  elemental function bessel_j0(x)   ! Bessel function of the first kind,
    real :: bessel_j0               ! order 0
    real :: x                       ! real
  end function
```

```
bessel_j1 (x)
  elemental function bessel_j1(x)   ! Bessel function of the first kind,
    real :: bessel_j1               ! order 1
    real :: x                       ! real
  end function
```

```
bessel_jn (n,x)
  elemental function bessel_jn(n,x) ! Bessel function of the first kind,
    real :: bessel_jn               ! order n
    integer :: n                    ! integer
    real :: x                       ! real
  end function
```

```
bessel_jn (n1,n2,x)
  function bessel_jn(n1,n2,x)       ! Bessel function of the first kind,
    real :: bessel_jn               ! order n1 through n2
    integer :: n1,n2                ! integer
    real :: x                       ! real
  end function
```

The result is a rank one array with an extent `n2-n1+1`. Element `i` of the result is the Bessel function of the first kind and order `N1+i-1` of `X`.

```
bessel_y0 (x)
  elemental function bessel_y0(x)   ! Bessel function of the second
    real :: bessel_y0               ! kind, order 0
    real :: x                       ! real
  end function

bessel_y1 (x)
  elemental function bessel_y1(x)   ! Bessel function of the second
    real :: bessel_y1               ! kind, order 1
    real :: x                       ! real
  end function

bessel_yn (n,x)
  elemental function bessel_yn(n,x) ! Bessel function of the second
    real :: bessel_yn               ! kind, order n
    integer :: n                    ! integer
    real :: x                       ! real
  end function

bessel_yn (n1,n2,x)
  function bessel_yn(n1,n2,x)       ! Bessel function of the second
    real :: bessel_yn               ! kind, order n1 through n2
    integer :: n1,n2                ! integer
    real :: x                       ! real
  end function
```

The result is a rank one array with an extent `n2-n1+1`. Element `i` of the result is the Bessel function of the second kind and order `N1+i-1` of `X`.

```
bge (i,j)
  elemental function bge(i,j) ! bitwise greater than or equal
    logical :: bge
    integer :: i,j
  end function

bgt (i,j)
  elemental function bgt(i,j) ! bitwise greater than
    logical :: bgt
    integer :: i,j
  end function

ble (i,j)
  elemental function ble(i,j) ! bitwise less than or equal
    logical :: ble
    integer :: i,j
  end function

blt (i,j)
  elemental function blt(i,j) ! bitwise less than or equal
    logical :: blt
    integer :: i,j
  end function

bit_size (i)
  inquiry function bit_size(i)      ! number of bits in argument i
```

```
    integer :: bit_size
    integer :: i                       ! i may be an array of any rank
  end function


btest (i, pos)
  elemental function btest(i,pos)    ! returns true if the bit in
                                     ! position pos of i is1
    logical :: btest
    integer :: i
    integer :: pos                   ! 0 ≤ pos < bit_size(i)
  end function

ceiling (a)
  elemental function ceiling(a)      ! least integer greater than or
                                     ! equal to a
    integer :: ceiling
    real :: a
  end function

char (i, kind)
  elemental function char(i,kind)    ! character in the ith position of
                                     ! the character set
    character(kind) :: char          ! default character kind if kind is
                                     ! absent
    integer :: i                     ! in range 0 ≤ i ≤ n-1 where n is #
                                     ! of characters
    integer, optional :: kind        ! if present, must be a scalar
                                     ! initialization expression
  end function
```

Note that ichar(char(y,kind(x))) = y and char(ichar(x),kind(x)) = x.

```
cmplx (x, y, kind)
  elemental function cmplx(x,y,kind)       ! complex number with real
                                           ! part x, imaginary part y
    complex(kind) :: cmplx                 ! or default kind if kind is
                                           ! absent
    real :: x                              ! or integer or complex
    real, optional :: y                    ! or integer, or absent if x
                                           ! is complex
    integer, optional :: kind              ! if present, must be a scalar
                                           ! initialization expression
  end function
```

If y is absent and x is not complex, then the imaginary part of the result is zero.

```
command_argument_count ()
  integer function command_argument_count ()    ! the number of command
                                                ! line arguments
  end function


conjg (z)
  elemental function conjg(z) ! the conjugate of a complex number
    complex :: conjg
    complex :: z
  end function
```

```
cos (x)
  elemental function cos(x)   ! the cosine of x
    real :: cos               ! same type and kind as x
    real :: x                 ! may be complex
  end function
```

If **x** is of type real, it is regarded as a value in radians; if **x** is of type complex, its real part is regarded as a value in radians.

```
cosh (x)
  elemental function cosh(x)  ! the hyperbolic cosine of x
    real :: cosh
    real :: x                 ! may be complex
  end function

count (mask, dim)
  function count(mask,dim)    ! count the number of true elements of
                              ! mask
    integer :: count          ! count is an array if dim is present and
                              ! mask rank > 1
    logical :: mask(:)        ! may be any kind, any rank
    integer, optional :: dim  ! if present, 1 ≤ dim ≤ n, where n is rank
                              ! of mask
  end function
```

If **dim** is present and n is greater than 1 then the result is an array of rank n-1. For example, if **mask** is a 3x2 array and **dim** is 2, then the result is a one-dimensional array of size 3, with the count taking place along each row of **mask**.

```
cpu_time (time)
  subroutine cpu_time (time)   ! return cpu usage time
    real :: time               ! processor dependent approximation
                               ! in seconds
  end subroutine

cshift (array, shift, dim)
  function cshift(array,shift,dim)  ! circularly shift array
    real :: cshift(:)               ! same type, kind, and shape as
                                    ! array
    real :: array(:)                ! or any type, kind, and rank (not
                                    ! scalar)
    integer :: shift                ! amount to be shifted, positive for
                                    ! left shifts
    integer, optional :: dim        ! if present, 1≤dim≤n, where n is
                                    ! rank of array
  end function
```

Positive **shift** amounts are “left” shifts (e.g., cshift(i)=array(i+1)) and negative shifts are “right”; values shifted “off” one end are routed into the other end. If **dim** is absent it is assumed to be 1. If **array** has rank greater than 1 then n-1 “one-dimensional shifts” take place along the dimension specified by **dim**. For example, if **array** is a 3x2 array and **dim**=1, each of the two columns of **array** are shifted an amount specified by **shift**. **shift** is allowed to be an array of rank n-1, specifying a different shift amount for each “one dimensional shift”. In the preceding example, **shift** could be a one-dimensional array of

two elements, having values, say, of 2 and -1; in this case the first column of the array will be circularly shifted “up” two and the second column will be shifted “down” one.

```
date_and_time (date, time, zone, values)
  subroutine date_and_time(date,time,zone,values)     ! returns date and
                                                      ! time information
    character(*), optional, intent(out)) :: date      ! date in CCYYMODD
                                                      ! format
    character(*), optional, intent(out)) :: time      ! time in
                                                      ! HHMMSS.SSS
                                                      ! format
    character(*), optional, intent(out)) :: zone      ! zone in ±HHMM
                                                      ! format
    integer, optional, intent(out)) :: values(:)      ! date, time, and
                                                      ! zone in integer
                                                      ! form
  end subroutine
```

Returned results are compatible with the representations defined in ISO 8601:1988. CC is the century, `YY` the year of the century, `MO` the month, `DD` the day of the month, `HH` the hour of the day, `MM` the minutes of the hour , and `SS.SSS` the seconds/milliseconds. For **zone**, the result is the hours and minutes from Coordinated Universal Time. For **values**, **values(1)** is the integer form of `CCYY` - e.g., the year, **values(2)** is `MO`, **values(3)** is `DD`, **values (4)** is the zone, **values(5)** is `HH` (range 0:23), **values(6)** is `MM` (range 0:59), **values(7)** is the seconds (range 0:60), and **values(8)** is the milliseconds.

```
dble (a)
  elemental function dble(a)  ! converts a to a double precision real
                              ! value
    real(DOUBLE) :: dble      ! where DOUBLE is the double-precision
                              ! kind value or integer or complex,
    real :: a                 ! of any kind
  end function
```

If **a** is complex, then **dble** returns the real part of **a** in double precision form.

```
digits (x)
  inquiry function digits(x)  ! returns the model value of n if x is
                              ! integer, or q if x is real
    integer :: digits
    integer :: x              ! may also be real and/or an array
end function

dim (x, y)
  elemental function dim(x,y) ! returns max(0,x-y)
    integer :: dim            ! same type and kind as x (and y)
    integer :: x, y           ! may also be real; y must have same type
                              ! and kind as x
  end function
```

```
dot_product (vector_a, vector_b)
  function dot_product(vector_a,vector_b) ! the dot-product
                                          ! multiplication of numeric or
                                          ! logical vectors
    real :: dot_product                   ! may also be logical or
                                          ! integer - see discussion
    real :: vector_a(:), vector_b(:)      ! one-dimensional arrays of
                                          ! the same size and both
                                          ! either of type logical or of
                                          ! any numeric type
  end function
```

If the vectors are size zero, the result value is either zero or false; otherwise, if the vectors are of type logical the result is type logical with value and kind of **any(vector_a.and.vector_b)**, if **vector_a** is of type integer or real the result value and kind are those of **sum(vector_a*vector_b)**, and else the result value and kind are are those of **sum(conjg(vector_a)*vector_b)**.

```
dprod (x, y)
  elemental function dprod(x,y)     ! returns double-precision product
                                    ! of two default real values
    real(DOUBLE) :: dprod           ! where DOUBLE is the double-
                                    ! precision kind value
    real :: x, y
  end function

dshiftl (i, j, shift)
  elemental function dshiftl(i,j,shift)   ! rightmost shift bits of the
                                          ! result are the same as
                                          ! leftmost bits of j and
                                          ! remaining bits of result are
                                          ! are the same as rightmost
                                          ! bits of i
    integer :: i, j, shift                ! any kind
  end function

dshiftr (i, j, shift)
  elemental function dshiftr(i,j,shift)   ! leftmost shift bits of the
                                          ! result are the same as
                                          ! rightmost bits of j and
                                          ! remaining bits of result are
                                          ! are the same as leftmost
                                          ! bits of i
    integer :: i, j, shift                ! any kind
  end function
```

```
eoshift (array, shift, boundary, dim)
  function eoshift(array,shift,boundary,dim)    ! end-off shift off
                                                ! array
    real :: eoshift(:)                          ! same type, kind, and
                                                ! shape as array
    real :: array(:)                            ! or any type, kind, and
                                                ! rank (not scalar)
    integer :: shift                            ! amount to be shifted,
                                                ! positive for left
                                                ! shifts
    integer, optional :: boundary               ! same type and kind as
                                                ! array - value for
                                                ! vacated positions
    integer, optional :: dim                    ! if present, 1≤dim≤n,
                                                ! where n is rank of
                                                ! array
  end function
```

eoshift is exactly the same as **cshift**, except that values shifted “off” one end are not routed into the vacated positions on the other end; the **boundary** value is placed in the vacated positions. If **boundary** is omitted, the default value is 0, 0.0, (0.0,0.0), false, or blanks, depending on whether **array** is type integer, real, complex, logical, or character, respectively. **boundary** is allowed to be an array of rank n-1, specifying a different fill value amount for each vacated position.

```
epsilon (x)
  inquiry function epsilon(x) ! a positive value almost negligible
compared to unity
    real :: epsilon           ! same type and kind as x
    real :: x                 ! may be any kind; may be an array of any
                              ! rank
  end function

erf (x)
  elemental function erf(x)   ! the error function
    real :: erf               ! type and kind of x, must be real
    real :: x
  end function

erfc (x)
  elemental function erfc(x)  ! the complementary error function
    real :: erfc              ! type and kind of x, must be real
    real :: x
  end function

erfc_scaled (x)
  elemental function erfc_scaled(x) ! scaled complementary error
                                    ! function
    real :: erfc_scaled
    real :: x
  end function
```

```
execute_command_line (command, wait, exitstat, cmdstat, cmdmsg)
  subroutine execute_command_line (command, wait, exitstat, cmdstat,
                                   cmdmsg)
    character(len=*) :: command           ! command line
    logical, optional :: wait             ! default logical scalar
    integer, optional :: exitstat         ! default integer scalar
    integer, optional :: cmdstat          ! default integer scalar
    character(len=*), optional :: cmdmsg  ! error message
end subroutine
```

default execution is synchronous. If **wait** is present and equal to `.false.`, the command is executed asynchronously. **cmdstat** returns 0 if the command is successfully initiated and non-zero if it fails. **exitstat** returns the exit status of the command. **cmdmsg** returns text string if the command fails to initiate.

```
exp (x)
  elemental function exp(x)   ! an approximation to e**x
    real :: exp               ! type and kind of x may be complex, in
    real :: x                 ! which case its imaginary part is
                              ! regarded as a value in radians
  end function

exponent (x)
  elemental function exponent(x)    ! the real model exponent part of x
    integer :: exponent
    real :: x
  end function

floor (a)
  elemental function floor(a) ! greatest integer less than or equal to a
    integer :: floor
    real :: a
  end function

gamma (x)
  elemental function gamma(a) ! Gamma function
    real :: gamma
    real :: x
  end function

fraction (x)
  elemental function fraction(x)! the real model fractional part of x
    real :: fraction
    real :: x
  end function

get_command (command, length, status)
  subroutine get_command (command, length, status)
    character(len=*), optional :: command ! the command line
    integer, optional :: length           ! default integer kind
    integer, optional :: status           ! default integer kind
  end subroutine
```

`length` returns the actual length of the command line. If `command` is present and `len(command) < length`, `status` is set to -1, otherwise it is set to 0.

```
get_command_argument (number, value, length, status)
  subroutine get_command_argument (number, value, length, status)
    integer :: number                        ! default integer kind
    character(len=*), optional :: value     ! command line argument
    integer, optional :: length             ! default integer kind
    integer, optional :: status             ! default integer kind
  end subroutine
```

If **number** is 0, the name of the program is returned in **command**. **length** returns the actual length of the command line argument. If **value** is present and len**(value) < length**, **status** is set to -1, if number is greater than the number of command line arguments, **status** is set to 1, otherwise it is set to 0.

```
get_environment_variable (name, value, length, status, trim_name)
  subroutine get_environment_variable (name, value, length, status, &
                                       trim_name)
    character(len=*) name                    ! environment variable name
    character(len=*), optional :: value     ! environment variable value
    integer, optional :: length             ! default integer kind
    integer, optional :: status             ! default integer kind
    integer, optional :: trim_name          ! default logical kind
  end subroutine
```

length returns the actual length of the environment variable. If **value** is present and **len(value) < length**, **status** is set to -1, otherwise it is set to 0. If **name** does not exist, **status** is set to 1.

```
huge (x)
  inquiry function huge(x)     ! the largest value for the type and kind
                               ! of x
    real :: huge               ! same type and kind as x
    real :: x                  ! may be any kind; may be integer; may be
                               ! an array
  end function
```

```
hypot (x, y)
  elemental function hypot(x,y)      ! Euclidean distance
    real :: hypot                    ! same type and kind as x and y
    real :: x, y                     ! real
  end function
```

The result has a value equal to a processor-dependent approximation to `sqrt(x**2 + y**2)`.

```
iachar (c)
  elemental function iachar(c)       ! same as ichar, except the ASCII
                                     ! collating sequence (ISO 646:1983)
    integer :: iachar                ! is used instead of the processor's
    character :: c
  end function
```

```
iand (i, j)
  elemental function iand(i,j)! perform logical and on bits of i and j
    integer :: iand           ! same kind as i (and j)
    integer :: i, j           ! any kind, but both must be the same kind
  end function
```

The result is the bit-by-corresponding-bit *and* of the arguments; if both argument bits are 1 the corresponding result bit is 1, otherwise the result bit is 0.

```
ibclr (i, pos)
  elemental function ibclr(i,pos)
    integer :: ibclr          ! same as i, but with the specified bit
                              ! set to 0
    integer :: i              ! may be any kind
    integer :: pos            ! position of bit in i to set to zero;
                              ! 0≤pos<bit_size(i)
  end function
```

```
ibits (i, pos, len)
  elemental function ibits(i,pos,len) ! "extract" a string of bits
                                      ! from i
    integer :: ibits                  ! same type and kind as i
    integer :: i                      ! may be any kind
    integer :: pos                    ! position of first bit to
                                      ! extract; pos≥0
    integer :: len                    ! number of bits to extract;
                                      ! len≥0, pos+len <= bit_size(i)
  end function
```

The result has the sequence of **len** bits from **i**, beginning at bit **pos**; these bits are in the rightmost bit positions of the result, with all other bits zero.

```
ibset (i, pos)
  elemental function ibset(i,pos)   ! same as ibclr, except the
                                    ! specified bit is set to 1 instead
                                    ! of 0
    integer :: ibset
    integer :: i
    integer :: pos
  end function
```

```
ichar (c)
  elemental function ichar(c) ! position of a character in the processor
                              ! collating sequence
    integer :: ichar
    character :: c
  end function                ! same as iachar if the processor
                              ! collating sequence is ASCII
```

```
ieor (i, j)
  elemental function ieor(i,j)! perform logical exclusive-or on bits of
                              ! i and j
    integer :: ieor           ! same kind as i (and j)
    integer :: i, j           ! any kind, but both must be the same kind
  end function
```

The result is the bit-by-corresponding-bit *exclusive-or* of the arguments; if one of the argument bits is 1 and the other is 0, the corresponding result bit is 1, otherwise the result bit is 0.

```
index (string, substring, back, kind)
  elemental function index(string,substring,back,kind)
                                    ! search string for a substring
    integer :: index                ! beginning position of substring
                                    ! in string (or zero)
    character(*) :: string          ! may be any kind
    character(*) :: substring       ! same kind as string
    logical, optional :: back       ! if present and true, search is
                                    ! from back of string
    integer,optional :: kind        ! if present, the kind of the result
  end function
```

If **back** is absent or present with the value false, the result is the minimum positive value of **i** such that **string(i:i+len(substring)–1) == substring** or zero if there is no such value; if **back** is present with the value true, the result is the maximum value of **i** less than or equal to **len(string)–len(substring)+1** such that **string(i:i+len(substring)-1) == substring** or zero if there is no such value. Zero is returned if **len(string)<len(substring)** and one is returned if **len(substring)==0**.

```
int (a, kind)
  elemental function int(a,kind)    ! convert a to an integer value (of
                                    ! specified kind)
    integer(kind) :: int            ! the converted integer value
    real :: a                       ! may be any numeric type
    integer, optional :: kind       ! if present, must be a scalar
                                    ! initialization expression. if kind
  end function                      ! is absent, the result kind is
                                    ! default integer
```

If **a** is of type integer, the result is this same value, but possibly of a different kind. If **a** is of type real, the real value is truncated toward zero (e.g., **int(3.7)** is 3 and **int(-3.7)** is -3). If **a** is of type complex, the result is **int(real(a))**.

```
int_ptr_kind ()
  integer function int_ptr_kind()   ! return the INTEGER KIND large
                                    ! enough to hold an address
    integer :: int_ptr_kind         ! the result kind is default integer
  end function
```

The int_ptr_kind() intrinsic function is an extension to the Fortran 95 language.

```
ior (i, j)
  elemental function ior(i,j) ! perform logical inclusive-or on bits of
                              ! i and j
    integer :: ior            ! same kind as i (and j)
    integer :: i, j           ! any kind, but both must be the same kind
  end function
```

The result is the bit-by-corresponding-bit *inclusive-or* of the arguments; if either one, or both, of the argument bits is 1, the corresponding result bit is 1, otherwise the result bit is 0.

```
ishft (i, shift)
  elemental function ishift(i,shift)      ! logically end-off shift the
                                          ! bits of i the amount shift
    integer :: i                          ! may be any kind
    integer :: shift                      ! amount to shift the bit
                                          ! pattern of i; shift must
                                          ! have a magnitude less than
  end function                            ! or equal to bit_size(i)
```

The result is the value obtained by shifting the bits of **i** by **shift** positions. If **shift** is positive, the shift is to the left. Bits shifted out are lost; zeros are shifted in at the opposite end.

```
ishftc (i, shift, size)
  elemental function ishift(i,shift, size)  ! same as ishft, but the
                                            ! shift is circular
                                            ! rather than end-off
    integer :: i                            ! may be any kind
    integer :: shift                        ! amount to shift the
                                            ! bit pattern of i;
                                            ! shift ≤ size
    integer, optional :: size               ! 0 ≤ size < bit_size(i)
  end function                              ! if size is omitted,
                                            ! bit_size(i) is assumed
```

The result has the value obtained by shifting the bits of **i** by **shift** positions. If **shift** is positive, the shift is to the left. Bits shifted out are shifted into at vacated positions at the opposite end.

```
is_iotstat_end (i)
  logical function is_iotstat_end(i)      ! returns .true. if i equals
                                          ! the value returned in an
                                          ! IOSTAT specifier for end of
                                          ! file
    integer :: i                          ! default integer kind
  end function

is_iotstat_eor (i)
  logical function is_iotstat_eor(i)      ! returns .true. if i equals
                                          ! the value returned in an
                                          ! IOSTAT specifier for end of
                                          ! of record
    integer :: i                          ! default integer kind
  end function
```

```
kind (x)
  inquiry function kind(x)    ! the kind type parameter value of any
                              ! object
    integer :: kind           ! the kind value of x
    real :: x                 ! may be any intrinsic type and any kind;
                              ! may be an array
  end function

leadz (i)
  elemental function leadz (i)      ! number of leading zero bits
    integer :: leadz                ! default integer kind
    integer :: i                    ! integer
  end function

lbound (array, dim)
  inquiry function lbound(array,dim)  ! the lower bound(s) of array
    integer :: lbound                 ! scalar if dim is present; a rank
                                      ! one array otherwise
    real :: array(:)                  ! may have any type, kind, and
                                      ! rank
    integer, optional :: dim          ! if present, 1 ≤ dim ≤ n, where n
                                      ! is the rank of array
  end function
```

If **dim** is present the result is a scalar and is the lower bound of **array** along the dimension **dim**. If **dim** is absent the result is a one-dimensional array whose size is the rank of **array**, and the value of each element of the result is the lower bound of that dimension of **array**. If **array** is an array expression other than an array name (e.g., an array section), the lower bound for each dimension is 1.

```
len (string)
  inquiry function len(string)      ! the length (numbers of characters)
                                    ! of a string
    integer :: len                  ! the length of string
    character(*) :: string          ! may be any kind; may be an array
  end function                      ! if string is an array, len is the
                                    ! length of each element

len_trim (string)
  elemental function len_trim(string) ! same as len(trim(string))
    integer :: len_trim                ! the length of string with all
                                       ! trailing blanks removed
    character(*) :: string
  end function

lge (string_a, string_b)
  elemental function lge(string_a,string_b) ! string comparison, based
                                            ! on ASCII
    logical :: lge                          ! true if string_a >=
                                            ! string_b in the ASCII
                                            ! collating sequence,
    character(*) :: string_a, string_b      ! false otherwise
  end function
```

```
lgt (string_a, string_b)
  elemental function lgt(string_a,string_b) ! string comparison, based
                                            ! on ASCII
    logical :: lgt                          ! true if string_a >
                                            ! string_b in the ASCII
                                            ! collating sequence,
    character(*) :: string_a, string_b      ! false otherwise
  end function

lle (string_a, string_b)
  elemental function lle(string_a,string_b) ! string comparison, based
                                            ! on ASCII
    logical :: lle                          ! true if string_a <=
                                            ! string_b in the ASCII
                                            ! collating sequence,
    character(*) :: string_a, string_b      ! false otherwise
  end function

llt (string_a, string_b)
  elemental function llt(string_a,string_b) ! string comparison, based
                                            ! on ASCII
    logical :: llt                          ! true if string_a <
                                            ! string_b in the ASCII
                                            ! collating sequence,
    character(*) :: string_a, string_b      ! false otherwise
  end function

log (x)
  elemental function log(x)   ! the natural logarithm, log_e x
    real :: log               ! same type and kind as x
    real :: x                 ! may be complex; if real, x > 0
  end function                ! if complex, x must not be zero

log10 (x)
  elemental function log10(x) ! base-10 logarithm, log_10 x
    real :: log10             ! same kind as x
    real :: x                 ! x > 0
  end function

log_gamma (x)
  elemental function log_gamma (x)  ! Logarithm of the absolute value of
                                    ! the gamma function
    real :: log_gamma               ! same type and kind as x
    real :: x                       ! may be complex; if real, x > 0
  end function                      ! if complex, x must not be zero

logical (l, kind)
  elemental function logical(l,kind) ! convert between logical kinds
    logical(kind) :: logical         ! the value of l, but with kind
                                     ! kind
    logical :: l                     ! may be any kind
    integer, optional :: kind        ! if present, must be a scalar
                                     ! initialization expression
  end function                       ! if kind is absent, the result
                                     ! kind is default logical

maskl (i, kind)
  elemental function integer(i,kind) ! left justified mask
    integer(kind) :: maskl           !
```

```
    integer :: i                       ! non-negative
    integer, optional :: kind          ! if present, kind of result
  end function
```

The result has its leftmost **i** bits set to 1 and the remaining bits set to 0.

```
maskr (i, kind)
  elemental function integer(i,kind) ! right justified mask
    integer(kind) :: maskl           !
    integer :: i                     ! non-negative
    integer, optional :: kind        ! if present, kind of result
  end function
```

The result has its rightmost **i** bits set to 1 and the remaining bits set to 0.

```
matmul (matrix_a, matrix_b)
  function matmul(matrix_a,matrix_b)      ! matrix multiplication of two
                                          ! numeric or logical matrices
    real :: matmul(:,:)                   ! type and kind determined by
                                          ! the arguments - see below
    real :: matrix_a(:,:), matrix_b(:,:)  ! may be any kind; may be
                                          ! integer, complex or logical;
  end function                            ! one of the argument matrices
                                          ! (but not both) may be rank
                                          ! one
```

The two arguments must be both of type logical or of numeric (integer, real, complex) type. The size of the first (or only) dimension of **matrix_b** must equal the size of the last (or only) dimension of **matrix_a**. There are three cases: (1) matrix_a has shape (n,k) and matrix_b has shape (k,m), in which case the result has shape (n,m); (2) matrix_a has shape (k) and matrix_b has shape (k,m), in which case the result has shape (m); (3) matrix_a has shape (n,k) and matrix_b has shape (k), in which case the result has shape (n). For case (1) the (i,j) element of the result has the value and kind of **sum(matrix_a(i,:)*matrix_b(:,j))** if the arguments are of numeric type and **any(matrix_a(i,:).and.matrix_b(:,j))** otherwise. For case (2) the (i) element of the result has the value and kind of **sum(matrix_a(:)*matrix_b(:,i))** if the arguments are of numeric type and **any(matrix_a(:).and.matrix_b(:,j))** otherwise, and for case (3) the (i) element of the result has the value and kind of **sum(matrix_a(i,:)*matrix_b(:))** if the arguments are of numeric type and **any(matrix_a(i,:).and.matrix_b(:))** otherwise.

```
max (a1, a2, a3, ...)
  elemental function max(a1,a2,a3,...)  ! return the maximum of the
                                        ! argument values
    real :: max                         ! same type and kind as the
                                        ! arguments
    real :: a1, a2                      ! may be integer, but all
    real, optional :: a3, ...           ! arguments must have the same
                                        ! type and kind
  end function
```

```
maxexponent (x)
  inquiry function maxexponent(x)   ! maximum model exponent that this
                                    ! kind of real can have
    integer :: maxexponent
   real :: x                        ! may be any kind; may be an array
  end function

maxloc (array, dim, mask)
  function maxloc(array,dim,mask)           ! location in array of
                                            ! element with the maximum
                                            ! value
    integer :: maxloc(:)                    ! result element values are
                                            ! the subscripts of the
                                            ! location
    real :: array(:)                        ! any kind, any rank; can
                                            ! be type integer
    integer, optional :: dim                ! dimension along which to
                                            ! determine maximum location
    logical, optional :: mask(size(array))  ! must be same shape as
                                            ! array
  end function
```

When **dim** is omitted, the result is a rank one array with length equal to the rank of **array.** If **array** has rank one and **dim** is present, the result is a scalar. If array has rank n greater than 1 and **dim** is present, 1≤dim≤n and **dim** specifies the dimension along which to determine the maximum values; in this case the result is an array of rank n-1 and shape (d1,d2,...,ddim-1, ddim+1,...,dn) where (d1,d2,...,dn) is the shape of **array** and each value of the result is the location of the maximum value along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are searched for the maximum value.

```
maxval (array, dim, mask)
  function maxval(array,dim,mask)           ! the maximum value in
                                            ! array, or along one of its
                                            ! diimensions
    real :: maxval                          ! same kind and type as
                                            ! array
    real :: array(:)                        ! any kind, any rank; can be
                                            ! type integer
    integer, optional :: dim                ! dimension along which to
                                            ! determine maximum values
    logical, optional :: mask(size(array))  ! must be same shape as
                                            ! array
  end function
```

The result is scalar if **dim** is omitted or **array** has rank 1 (as illustrated in the interface), in which case the value returned is the maximum element value in **array**. If array has rank n greater than 1 and **dim** is present, 1≤dim≤n and **dim** specifies the dimension along which to determine the maximum values; in this case the result is an array of rank n-1 and shape (d1,d2,...,ddim-1, ddim+1,...,dn) where (d1,d2,...,dn) is the shape of **array** and each value of the result is the maximum value along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are searched for the maximum value.

```
merge (tsource, fsource, mask)
  elemental function merge(tsource,fsource,mask) ! select one of two
                                                 ! arguments, based on a
                                                 ! mask
    real :: merge                                ! same type and kind as
                                                 ! tsource
    real :: tsource                              ! may be of any type
                                                 ! and kind
    real :: fsource                              ! same type and kind as
                                                 ! tsource
    logical :: mask                              ! return tsource if
                                                 ! mask is true,
  end function                                   ! fsource if mask is
                                                 ! false
```

merge is an elemental function and is most often used to merge two arrays, based on the (conformable) **mask**.

```
merge_bits (i, j, mask)
  elemental function merge_bits(i,j,mask)        ! merge bits under mask
    integer:: merge_bits
    integer:: i                                  ! integer or boz
    integer:: j                                  ! integer or boz
    integer :: mask                              ! integer or boz
  end function

min (a1, a2, a3, ...)
  elemental function min(a1,a2,a3,...)  ! return the minimum of the
                                        ! argument values
    real :: min                         ! same type and kind as the
                                        ! arguments
    real :: a1, a2                      ! may be integer,
    real, optional :: a3, ...           ! but all arguments must have
                                        ! the same type and kind
  end function

minexponent (x)
  inquiry function minexponent(x)  ! minimum model exponent that this
                                   ! kind of real can have
    integer :: minexponent
    real :: x                      ! may be any kind; may be an array
  end function

minloc (array, dim, mask)
  function minloc(array,dim, mask)  ! same as maxloc,
    integer :: minloc(:)            ! but with minimum value rather than
                                    ! maximum value
    real :: array(:)
    integer, optional :: dim
    logical, optional :: mask(size(array))
end function

minval (array, dim, mask)
  function minval(array,dim,mask)  ! same as maxval,
    real :: minval                 ! but with minimum value rather than
                                   ! maximum value
    real :: array(:)
    integer, optional :: dim
```

```
    logical, optional :: mask(size(array))
  end function

mod (a, p)
  elemental function mod(a,p)   ! the remainder function (has sign of a)
    integer :: mod              ! same type and kind as a; value is
                                ! a-int(a/p)*p
    integer :: a                ! any kind; may be real
    integer :: p                ! same type and kind as a; the value of
                                ! p must not be zero
  end function                  ! mod and modulo are the same for
                                ! positive values of a and p

modulo (a, p)
  elemental function modulo(a,p)  ! the modulo function (has sign of p)
    integer :: modulo             ! same type and kind as a; value is
                                  ! a-floor(a/p)*p
    integer :: a                  ! any kind; may be real
    integer :: p                  ! same type and kind as a; the value
                                  ! of p must not be zero
  end function                    ! mod and modulo are the same for
                                  ! positive values of a and p


move_alloc (from, to)
  pure subroutine move_alloc(from,to)     ! move an allocation
    any, allocatable() :: from            ! any kind
    any, allocatable() :: to              ! any kind
  end function
```

from may be any kind and rank, but must be allocatable. **to** must be the same kind and rank and must be allocatable.

```
mvbits (from, frompos, len, to, topos)
  elemental subroutine mvbits(from,frompos,len,to,topos) ! move bits
                                     ! from from to to
    integer, intent(in) :: from      ! may be any kind
    integer, intent(in) :: frompos   ! 0 £ frompos < bit_size(from)
    integer, intent(in) :: len       ! 0 £ len
    integer, intent(inout) :: to     ! same kind as from (and may be the
                                     ! same object)
    integer, intent(in) :: topos     ! 0 £ topos; (topos+len) <
                                     ! bitsize(to)
  end subroutine
```

Copies **len** bits from object **from**, starting at bit position **frompos** in **from**, to object **to**, starting at bit position **topos** in **to**.

```
nearest (x, s)
  elemental function nearest(x,s) ! the nearest value in the specified
                                  ! direction
    real :: nearest               ! same kind as x; value is that
                                  ! nearest x, but not x
    real :: x                     ! may be any kind
    real :: s                     ! must not be zero; s > 0 means
                                  ! nearest > x
  end function                    ! s < 0 means nearest < x
```

```
function new_line (a)
  character(len=1)new_line       ! new line character
  character :: a                 ! scalar or array
end function
```

This function returns char(10).

```
nint (a, kind)
elemental function nint(a,kind)  ! integer nearest to a
  integer(kind) :: nint          ! value is int(a+sign(0.5,a))
    real :: a
    integer, optional :: kind    ! if present, must be a scalar
                                 ! initialization expression
  end function                   ! if kind is absent, the result kind is
                                 ! default integer
```

```
not (i)
  elemental function not(i)   ! logical bit-wise complement
    integer :: not            ! the bit complement of i
    integer :: i
  end function                ! 1 bits in i become 0 in not; 0 bits in i
                              ! become 1 in not
```

```
null (mold)
  function null(mold)         ! disassociated pointer
    any :: null               ! type determined by mold or context, see
                              ! below
    any :: mold
  end function
```

`mold` is a pointer of any type. Its association status can be undefined, disassociated, or associated. If its status is associated, the target does not have to be defined. If *`mold`* is present the result type is the same as *`mold`*; otherwise the result type is determined by the context

```
pack (array, mask, vector)
  function pack(array,mask,vector)  ! pack an array of any shape into an
                                    ! array of rank 1
    real :: pack(:)                 ! same type and kind as array
    real :: array(:)                ! maybe any type, kind, and shape
    logical :: mask(size(array))    ! same shape as array
    real, optional :: vector(:)     ! if present, must have at least
                                    ! count(mask) elements
  end function
```

If **vector** is present the size of **pack** is the size of **vector**; otherwise the size of **pack** is **count(mask)**. The elements of **array** that correspond to true values in **mask** are placed in **pack**, starting with the first element of **pack** and in array element order from **array**.

```
popcnt (i)                       ! the number of one bits
  integer function popcnt(i)
    integer :: popcnt            ! default integer kind
    integer :: i                 ! integer
  end function
```

```
poppar (i)                          ! parity: 0 or 1
  integer function poppar (i)
    integer :: poppar               ! default integer kind
    integer :: i                    ! integer
  end function

precision (x)
  inquiry function precision(x)  ! the decimal precision of x
    integer :: precision
    real :: x                       ! may be any kind, may be complex, may
                                    ! be an array
  end function

present (a)
  inquiry function present(a)  ! determine whether an optional argument
                               ! is present
    logical :: present         ! true if a is present, false otherwise
    real :: a                  ! may be any type and kind; a must be an
                               ! optional argument
  end function                 ! of the procedure referencing the
                               ! present function

product (array, dim, mask)
  function product(array,dim,mask)  ! product of the elements of array
    real :: product                 ! same type and kind as array
    real :: array(:)                ! may be any kind, any numeric type,
                                    ! and any rank
    integer, optional :: dim        ! if present, 1 £ dim £ n, where n
                                    ! is rank of array
    logical, optional :: mask       ! if present, mask must have same
                                    ! shape as array
  end function
```

The result is scalar if **dim** is omitted or **array** has rank 1, in which case the value returned is the product of the elements of **array**. If array has rank n greater than 1 and **dim** is present, **dim** specifies the dimension along which to compute the products; in this case the result is an array of rank n-1 and shape (d1,d2,...,ddim-1, ddim+1,...,dn) where (d1,d2,...,dn) is the shape of **array** and each value of the result is the product of the elements along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are used in computing
the product(s).

```
radix (x)
  inquiry function radix(x)       ! radix (base) of the number model for
                                  ! x
    integer :: radix
    real :: x                     ! may be any kind, any numeric type;
                                  ! may be an array
  end function

random_number (harvest)
  subroutine randon_number(harvest) ! generates one or more pseudorandom
                                    ! numbers
  real, intent(out) :: harvest      ! may be any kind, may be an array
end subroutine
```

If **harvest** is a scalar, a single pseudorandom number from the uniform distribution between 0 and 1 is generated and assigned to **harvest**; if **harvest** is an array, **size(harvest)** such numbers are generated and assigned to **harvest**.

```
random_seed (size, put, get)
  subroutine random_seed(size,put,get)       ! set or retrieve the
                                             ! random_number seed
    integer, optional, intent(out) :: size   ! number of integers (n)
                                             ! used for the value of the
                                             ! seed
    integer, optional, intent(in) :: put(:)  ! size(put) must be equal
                                             ! to n; set the seed to put
    integer, optional, intent(out) :: get(:) ! size(get) must be equal
                                             ! to n; retrieve the seed
                                             ! into get
  end subroutine                             ! a given call to
                                             ! random_seed has at most
                                             ! one argument
```

If a call to **random_seed** is made without any arguments, the seed is set to an implementation determined value. When the argument is **put**, the seed is reinitialized to this value; when the argument is **get**, the current value of the seed is retrieved.

```
range (x)
  inquiry function range(x)  ! decimal exponent range for x
    integer :: range         ! value is int(log10(huge(x))) - but see
                             ! comment below
    real :: x                ! may be any kind, any numeric type; may be
                             ! an array
  end function
```

If **x** is of type integer, **huge(x)** returns an integer, which is not legal for **log10**; the effect for **range** is, however, as if the equivalent real value had been returned for huge. If **x** is of type real the value actually returned by **range** is min(int(log10(huge(x))),-int(log10(tiny(x)))).

```
real (a, kind)
  elemental function real(a,kind)   ! convert a to the equivalent real
                                    ! value of specified kind
    real(kind) :: real              ! the converted real value
    real :: a                       ! may be any kind and any numeric
                                    ! type
    integer, optional :: kind       ! if present, must be a scalar
                                    ! initialization expression
  end function ! if kind is absent, the result kind is default real
```

If **a** is of type real, the result is this same value, but possibly of a different kind. If **a** is an integer, the equivalent real value is returned. If **a** is complex, the result is the real part of **a**.

```
repeat (string, ncopies)
  function repeat(string,ncopies)             ! concatenate several
                                              ! copies of a string
    character(len(string)*ncopies) :: repeat  ! ncopies of string
                                              ! concatenated ; same kind
                                              ! as string
    character(*) :: string                    ! may be any kind
    integer :: ncopies                        ! 0 ≤ ncopies
  end function
```

```
reshape (source, shape, pad, order)
  function reshape(source,shape,pad,order)   ! reshape source into the
                                             ! array shape specified
                                             ! shape
    real :: reshape(:)                       ! same type and kind as
                                             ! source, with shape shape
    real :: source(:)                        ! may be any type, kind,
                                             ! and rank
    integer :: shape(:)                      ! all element values must
                                             ! be positive
    real, optional :: pad(:)                 ! same type and kind as
                                             ! source; may be any rank
    integer, optional :: order(size(shape))  ! a permutation of (1, 2,
                                             ! 3, ..., size(shape))
  end function
```

Values are copied from **source** (and then, if needed, from **pad**) to **reshape**, in array element order. If **size(source)** > **product(shape)**, the extra values of **source** are ignored. If **size(source)** < **product(shape)**, **pad** must be supplied, with **size(pad)** ≥ **product(shape)-size(source)**. If **order** is present it specifies the array element order of the **reshape** subscripts.

```
rrspacing (x)
  elemental function rrspacing(x)  ! reciprocal of the relative spacing
                                   ! of values near x
    real :: rrspacing              ! value is x/(nearest(x,1.)-x) for x
                                   ! > 0
    real :: x
  end function
```

```
scale (x, i)
  elemental function scale(x,i)    ! scales x by a specified amount
    real :: scale                  ! same kind as x; value is
                                   ! x*radix(x)**i
    real :: x                      ! may be any kind
    integer :: i                   ! scaling may be up (i > 0) or down
                                   ! (i < 0) (or i may be zero)
  end function
```

```
scan (string, set, back)
  elemental function scan(string,set,back)  ! search string for an
                                             ! occurrence of any
                                             ! character in set
    integer :: scan                          ! value is first such
                                             ! position in string
    character(*) :: string                   ! may be any kind
    character(*) :: set                      ! same kind as string
    logical, optional :: back                ! if present and true,
                                             ! search is from back of
                                             ! string instead
  end function                               ! if no match, zero is
                                             ! returned
```

```
selected_char_kind (name)
  function selected_int_kind(name)        ! select a character kind
    integer :: selected_char_kind
    character :: name                     ! default character kind
  end function
```

If **name** has the value DEFAULT, then the result has a value equal to that of the kind type parameter of default character. If **name** has the value ASCII, then the result has a value equal to that of the kind type parameter of ASCII character; otherwise the result has the value -1.

```
selected_int_kind (r)
  function selected_int_kind(r)   ! determines kind value for specifed
                                  ! integer range
    integer :: selected_int_kind  ! the kind value, or -1 if there is no
                                  ! such integer type
    integer :: r                  ! specifies an integer range of at
                                  ! least -10**r to +10**r
  end function
```

If more than one integer type meets the criteria, the kind value for the one with the smallest decimal exponent range is returned or, if there are several such, the smallest of these kind values.

```
selected_real_kind (p, r)
  function selected_real_kind(p,r)  ! determines kind value for real
                                    ! type with specified properties
    integer :: selected_real_kind   ! the kind value, or a negative
                                    ! value if there is no such real
                                    ! type
    integer, optional :: p          ! specifies a real type with at
                                    ! least p decimal digits of
                                    ! precision
    integer, optional :: r          ! specifies an exponent range of at
                                    ! least 10**-r to 10**r
  end function                      ! at least one argument must be
                                    ! present
```

The result is the kind type parameter of a real data type with decimal precision, as returned by the **precision** function, of at least **p** digits and a decimal exponent range, as returned by the **range** function, of at least **r**; if no such type is available on the processor, the result is –1 if the precision is not available, –2 if the exponent range is not available, and –3 if neither is available. If more than one real type meets the criteria, the kind value for the one with the smallest decimal precision is returned or, if there are several such, the smallest of these kind values.

```
set_exponent (x, i)
  elemental function set_exponent(x,i)  ! a value with the fractional
                                        ! part of x and exponent i
    real :: set_exponent                ! same kind as x; value is
                                        ! x*radix(x)**(i-exponent(x))
    real :: x                           ! may be any kind
    integer :: i                        ! i may be positive, negative,
                                        ! or zero
end function
```

```
shape (source)
  inquiry function shape(source)  ! determine the shape of an array
    integer :: shape(:)           ! element values are the extents of
                                  ! source
    real :: source(:)             ! may be any type, kind, and rank; may
                                  ! be scalar
  end function
```

The value of the kth element of **shape** is the size of the kth dimension of **source**. **size(shape)** is n, where n is the rank of **source**; if **source** is a scalar, n is zero. **source** must not be a disassociated pointer array, an unallocated allocatable array, or an assumed-size array.

```
shiftl (i, shift)
  elemental function shiftl(i,shift)      ! logically end-off left shift
                                          ! the bits of i the amount
                                          ! shift
    integer :: i                          ! integer
    integer :: shift                      ! amount to shift the bit
                                          ! pattern of i; shift must
                                          ! have a magnitude less than
  end function                            ! or equal to bit_size(i)
```

```
shifta (i, shift)
  elemental function shifta(i,shift)       ! logically end-off right
                                           ! shift the bits of i the
                                           ! amount shift with fill
    integer :: i                           ! integer
    integer :: shift                       ! amount to shift the bit
                                           ! pattern of i; shift must
                                           ! have a magnitude less than
  end function                             ! or equal to bit_size(i)
```

The leftmost bit of **i** is replicated **shift** times.

```
shiftr (i, shift)
  elemental function shiftr(i,shift)       ! logically end-off right
                                           ! shift the bits of i the
                                           ! amount shift
    integer :: i                           ! integer
    integer :: shift                       ! amount to shift the bit
                                           ! pattern of i; shift must
                                           ! have a magnitude less than
  end function                             ! or equal to bit_size(i)
```

```
sign (a, b)
  elemental function sign(a,b)  ! set the sign of a value
    real :: sign                ! value is |a| if b ≥ 0, -|a| if b < 0;
                                ! type and kind of a
    real :: a                   ! may be any kind; may be integer
    real :: b                   ! same type and kind as a
end function
```

```
sin (x)
  elemental function sin(x)     ! the sine of x
    real :: sin                 ! same type and kind as x
    real :: x                   ! may be complex
  end function
```

If **x** is of type real, it is regarded as a value in radians; if **x** is of type complex, its real part is regarded as a value in radians.

```
sinh (x)
  elemental function sinh(x)  ! the hyperbolic sine of x;
    real :: sinh              ! may be complex
    real :: x
  end function
```

```
size (array, dim)
  inquiry function size(array,dim)  ! determine number of elements in (a
                                    ! dimension of) an array
    integer :: size                 ! value is product(shape(array)) if
                                    ! dim is absent
    real :: array(:)                ! may be any type, kind, and rank
    integer, optional :: dim        ! if present, 1 £ dim £ n, where n
                                    ! is the rank of array, and
  end function                      ! the returned value is the extent
                                    ! of the dim dimension
```

```
spacing (x)
  elemental function spacing(x)  ! absolute spacing near x
    real :: spacing              ! value is nearest(x,1.)-x for x > 0
    real :: x
  end function

spread (source, dim, ncopies)
  function spread(source,dim,ncopies)  ! makes ncopies of source along a
                                       ! new dimension
    real :: spread(:,:)                ! same type and kind as source;
                                       ! rank 1 greater than source
    real :: source(:)                  ! may be any type, kind, and
                                       ! rank; may be scalar
    integer :: dim                     ! 1 ≤ dim ≤ n+1, where n is the
                                       ! rank of source
    integer :: ncopies                 ! number of copies to spread is
                                       ! max(0,ncopies)
  end function
```

spread broadcasts several copies of **source** along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater than **source**. If **source** is a scalar then **spread** is an array of rank one and size max(0,**ncopies**) and all element have the value of **source**. If **source** is an array with shape (d1,d2,...,dn), **spread** has shape (d1,d2,...,ddim-1, max(0,**ncopies**),ddim,ddim+1,...,dn).

```
sqrt (x)
  elemental function sqrt(x)  ! the square root of x
    real :: sqrt              ! same type and kind as x; value is
    real :: x                 ! may be complex
  end function                ! if x is of type real, x > 0
```

If **x** is complex, the real part of the result is nonnegative; if the real part is zero, the imaginary part is nonnegative.

```
storage_size (a, kind)
  function storage_size(a,kind)  ! size in bits of data object a
  integer(kind) :: storage_size  !
    any type :: a
    integer, optional :: kind    ! if present, must be a scalar
                                 ! integer constant expression
  end function                   ! if kind is absent, the result kind is
                                 ! default integer

sum (array, dim, mask)
  function sum(array,dim,mask)  ! sum of the elements of array
    real :: sum                 ! same type and kind as array
    real :: array(:)            ! may be any kind, any numeric type,
                                ! and any rank
    integer, optional :: dim    ! if present, 1 ≤ dim ≤ n, where n is
                                ! rank of array
    logical, optional :: mask   ! if present, mask must have same shape
                                ! as array
  end function
```

The result is scalar if **dim** is omitted or **array** has rank 1, in which case the value returned is the sum of the elements of **array**. If array has rank n greater than 1 and **dim** is present,

1≤dim≤n and **dim** specifies the dimension along which to compute the sums; in this case the result is an array of rank n-1 and shape (d1,d2,...,ddim-1, ddim+1,...,dn) where (d1,d2,...,dn) is the shape of **array** and each value of the result is the sum of the elements along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are used in computing the sum(s).

```
system_clock (count, count_rate, count_max)
  subroutine system_clock(count,count_rate,count_max) ! integer data
                                                      ! from a real-time
                                                      ! clock
    integer, intent(out), optional :: count      ! 0 ≤ count ≤ count_max
    integer, intent(out), optional :: count_rate ! clock counts per
                                                 ! second
    integer, intent(out), optional :: count_max  ! the maximum count can
                                                 ! have
  end subroutine
```

All values are processor dependent; **count_rate** indicates how many times **count** is incremented each second; when it reaches **count_max**, it resets to zero. If there is no processor clock **count** always returns -huge(0), **count_rate** returns zero, and **count_max** returns zero.

```
tan (x)
  elemental function tan(x)   ! the tangent of x
    real :: tan
    real :: x                 ! value is assumed to be radians, may be
                              ! complex
  end function

tanh (x)
  elemental function tanh(x)  ! the hyperbolic tangent of x
    real :: tanh              ! may be complex
    real :: x
  end function

tiny (x)
  inquiry function tiny(x)    ! the smallest positive value for the type
                              ! and kind of x
    real :: tiny ! same kind as x
    real :: x ! may be any kind; may be an array
  end function

trailz (i)
  elemental function trailz (i)     ! number of trailing zero bits
    integer :: trailz               ! default integer kind
    integer :: i                    ! integer
  end function

transfer (source, mold, size)
  function transfer(source,mold,size)  ! same bit pattern, but different
                                       ! type/kind
    real :: transfer                   ! bit pattern of source;
                                       ! type/kind of mold; may be an
                                       ! array
    real :: source                     ! may be any type and kind; may
                                       ! be scalar or array
```

```
    real :: mold                       ! may be any type and kind; may
                                       ! be scalar or array
    integer, optional :: size          ! specifies the shape of the
                                       ! result; the value must be
                                       ! positive
  end function
```

If **mold** is scalar and **size** is absent the result is scalar. If **mold** is an array and **size** is absent the result is a rank 1 array; its size is the smallest possible to hold all of the bits of **source**. If **size** is present, the result is a rank 1 array of this size; if this makes transfer longer than source, the extra part of transfer is undefined and if it makes transfer shorter than source the extra bits of source are not transferred.

```
transpose (matrix)
  function transpose(matrix)                        ! transpose matrix
    real :: transpose(size(matrix,2),size(matrix,1))  ! element (i,j) is
                                                    ! matrix(j,i)
    real :: matrix(:,:)                             ! may be any type
                                                    ! and kind
  end function
```

```
trim (string)
  function trim(string)        ! removes trailing blanks from a string
    character(*) :: trim       ! same as string, except all trailing
                               ! blanks removed
    character(*) :: string     ! may be any kind
  end function
```

```
ubound (array, dim)
  inquiry function ubound(array,dim)  ! the upper bound(s) of array
    integer :: ubound                 ! scalar if dim is present; a
                                      !rank one array otherwise
    real :: array(:)                  ! may have any type, kind, and
                                      ! rank
    integer, optional :: dim          ! if present, 1 ≤ dim ≤ n, where
                                      ! n is the rank of array
end function
```

If **dim** is present the result is a scalar and is the upper bound of **array** along the dimension **dim**. If **dim** is absent the result is a one-dimensional array whose size is the rank of **array**, and the value of each element of the result is the upper bound of that dimension of **array**. If **array** is an array expression other than an array name (e.g., an array section), the upper bound value is based on the lower bound value being 1.

```
unpack (vector, mask, field)
  function unpack(vector,mask,field)  ! unpack a vector into an array,
                                      ! under control of a mask
    real :: unpack(:)                 ! same type and kind as vector;
                                      ! same shape as mask
    real :: vector(:)                 ! may be any type and kind
    logical :: mask(:)                ! may be any rank
    real :: field(size(mask))         ! same type and kind as vector;
                                      ! same shape as mask -
  end function                        ! field may be a scalar, in which
                                      ! case it is broadcast
```

The element of the result that corresponds to the kth true element of **mask**, in array element order, is vector(k), for all the true values in **mask**. Each other element of the result is the value of the corresponding element of **field**.

```
verify (string, set, back)
  elemental function verify(string,set,back)   ! check that set contains
                                               ! all the characters in
                                               ! string
    integer :: verify                          ! value is first position
                                               ! in string that is not a
                                               ! set character
    character(*) :: string                     ! may be any kind
    character(*) :: set                        ! same kind as string
    logical, optional :: back                  ! if present and true,
                                               ! check is from back of
                                               ! string instead
  end function                                 ! if all characters in
                                               ! string are in set, zero
                                               ! is returned
```

Appendix A

ASCII Table

ASCII codes 0 through 31 are control codes that may or may not have meaning on Linux. They are listed for historical reasons and may aid when porting code from other systems. Codes 128 through 255 are extensions to the 7-bit ASCII standard and the symbol displayed depends on the font being used; the symbols shown below are from the Times New Roman font. The Dec, Oct, and Hex columns refer to the decimal, octal, and hexadecimal numerical representations.

Character	Dec	Oct	Hex	Description
NULL	0	000	00	null
SOH	1	001	01	start of heading
STX	2	002	02	start of text
ETX	3	003	03	end of text
ECT	4	004	04	end of trans
ENQ	5	005	05	enquiry
ACK	6	006	06	acknowledge
BEL	7	007	07	bell code
BS	8	010	08	back space
HT	9	011	09	horizontal tab
LF	10	012	0A	line feed
VT	11	013	0B	vertical tab
FF	12	014	0C	form feed
CR	13	015	0D	carriage return
SO	14	016	0E	shift out
SI	15	017	0F	shift in
DLE	16	020	10	data link escape
DC1	17	021	11	device control 1
DC2	18	022	12	device control 2
DC3	19	023	13	device control 3
DC4	20	024	14	device control 4
NAK	21	025	15	negative ack
SYN	22	026	16	synch idle
ETB	23	027	17	end of trans blk
CAN	24	030	18	cancel
EM	25	031	19	end of medium
SS	26	032	1A	special sequence
ESC	27	033	1B	escape
FS	28	034	1C	file separator
GS	29	035	1D	group separator
RS	30	036	1E	record separator
US	31	037	1F	unit separator
	32	040	20	space
!	33	041	21	exclamation
"	34	042	22	quotation mark
#	35	043	23	number sign
$	36	044	24	dollar sign
%	37	045	25	percent sign
	32	040	20	space
!	33	041	21	exclamation
"	34	042	22	quotation mark

Character	Dec	Oct	Hex	Description
#	35	043	23	number sign
$	36	044	24	dollar sign
%	37	045	25	percent sign
&	38	046	26	ampersand
'	39	047	27	apostrophe
(	40	050	28	opening paren
)	41	051	29	closing paren
*	42	052	2A	asterisk
+	43	053	2B	plus
,	44	054	2C	comma
-	45	055	2D	minus
.	46	056	2E	period
/	47	057	2F	slash
0	48	060	30	zero
1	49	061	31	one
2	50	062	32	two
3	51	063	33	three
4	52	064	34	four
5	53	065	35	five
6	54	066	36	six
7	55	067	37	seven
8	56	070	38	eight
9	57	071	39	nine
:	58	072	3A	colon
;	59	073	3B	semicolon
<	60	074	3C	less than
=	61	075	3D	equal
>	62	076	3E	greater than
?	63	077	3F	question mark
@	64	100	40	commercial at
A	65	101	41	upper case letter
B	66	102	42	upper case letter
C	67	103	43	upper case letter
D	68	104	44	upper case letter
E	69	105	45	upper case letter
F	70	106	46	upper case letter
G	71	107	47	upper case letter
H	72	110	48	upper case letter
I	73	111	49	upper case letter
J	74	112	4A	upper case letter
K	75	113	4B	upper case letter

Character	Dec	Oct	Hex	Description
L	76	114	4C	upper case letter
M	77	115	4D	upper case letter
N	78	116	4E	upper case letter
O	79	117	4F	upper case letter
P	80	120	50	upper case letter
Q	81	121	51	upper case letter
R	82	122	52	upper case letter
S	83	123	53	upper case letter
T	84	124	54	upper case letter
U	85	125	55	upper case letter
V	86	126	56	upper case letter
W	87	127	57	upper case letter
X	88	130	58	upper case letter
Y	89	131	59	upper case letter
Z	90	132	5A	upper case letter
[	91	133	5B	opening bracket
\	92	134	5C	back slash
]	93	135	5D	closing bracket
^	94	136	5E	circumflex
_	95	137	5F	underscore
`	96	140	60	grave accent
a	97	141	61	lower case letter
b	98	142	62	lower case letter
c	99	143	63	lower case letter
d	100	144	64	lower case letter
e	101	145	65	lower case letter
f	102	146	66	lower case letter
g	103	147	67	lower case letter
h	104	140	68	lower case letter
i	105	151	69	lower case letter
j	106	152	6A	lower case letter
k	107	153	6B	lower case letter
l	108	154	6C	lower case letter
m	109	155	6D	lower case letter
n	110	156	6E	lower case letter
o	111	157	6F	lower case letter
p	112	160	70	lower case letter
q	113	161	71	lower case letter
r	114	162	72	lower case letter
s	115	163	73	lower case letter
t	116	164	74	lower case letter
u	117	165	75	lower case letter
v	118	166	76	lower case letter
w	119	167	77	lower case letter
x	120	170	78	lower case letter
y	121	171	79	lower case letter
z	122	172	7A	lower case letter
{	123	173	7B	opening brace
|	124	174	7C	vertical bar
}	125	175	7D	closing brace
~	126	176	7E	tilde
	127	177	7F	delete
□	128	200	80	
□	129	201	81	
‚	130	202	82	
ƒ	131	203	83	
„	132	204	84	
…	133	205	85	
†	134	206	86	
‡	135	207	87	
ˆ	136	210	88	
‰	137	211	89	

Character	Dec	Oct	Hex
Š	138	212	8A
‹	139	213	8B
Œ	140	214	8C
□	141	215	8D
□	142	216	8E
□	143	217	8F
□	144	220	90
‘	145	221	91
’	146	222	92
“	147	223	93
”	148	224	94
•	149	225	95
–	150	226	96
—	151	227	97
˜	152	230	98
™	153	231	99
š	154	232	9A
›	155	233	9B
œ	156	234	9C
□	157	235	9D
□	158	236	9E
Ÿ	159	237	9F
	160	240	A0
¡	161	241	A1
¢	162	242	A2
£	163	243	A3
¤	164	244	A4
¥	165	245	A5
¦	166	246	A6
§	167	247	A7
¨	168	250	A8
©	169	251	A9
ª	170	252	AA
«	171	253	AB
¬	172	254	AC
-	173	255	AD
®	174	256	AE
¯	175	257	AF
°	176	260	B0
±	177	261	B1
²	178	262	B2
³	179	263	B3
´	180	264	B4
µ	181	265	B5
¶	182	266	B6
·	183	267	B7
¸	184	270	B8
¹	185	271	B9
º	186	272	BA
»	187	273	BB
¼	188	274	BC
½	189	275	BD
¾	190	276	BE
¿	191	277	BF
À	192	300	C0
Á	193	301	C1
Â	194	302	C2
Ã	195	303	C3
Ä	196	304	C4
Å	197	305	C5
Æ	198	306	C6
Ç	199	307	C7

Character	Dec	Oct	Hex
È	200	310	C8
É	201	311	C9
Ê	202	312	CA
Ë	203	313	CB
Ì	204	314	CC
Í	205	315	CD
Î	206	316	CE
Ï	207	317	CF
Ð	208	320	D0
Ñ	209	321	D1
Ò	210	322	D2
Ó	211	323	D3
Ô	212	324	D4
Õ	213	325	D5
Ö	214	326	D6
×	215	327	D7
Ø	216	330	D8
Ù	217	331	D9
Ú	218	332	DA
Û	219	333	DB
Ü	220	334	DC
Ý	221	335	DD
Þ	222	336	DE
ß	223	337	DF

Character	Dec	Oct	Hex
à	224	340	E0
á	225	341	E1
â	226	342	E2
ã	227	343	E3
ä	228	344	E4
å	229	345	E5
æ	230	346	E6
ç	231	347	E7
è	232	350	E8
é	233	351	E9
ê	234	352	EA
ë	235	353	EB
ì	236	354	EC
í	237	355	ED
î	238	356	EE
ï	239	357	EF
ð	240	360	F0
ñ	241	361	F1
ò	242	362	F2
ó	243	363	F3
ô	244	364	F4
õ	245	365	F5
ö	246	366	F6
÷	247	367	F7
ø	248	370	F8
ù	249	371	F9
ú	250	372	FA
û	251	373	FB
ü	252	374	FC
ý	253	375	FD
þ	254	376	FE
ÿ	255	377	FF

Appendix B

Exceptions and IEEE Arithmetic

Three modules are provided to support floating-point exceptions and IEEE arithmetic: `IEEE_FEATURES`, `IEEE_ARITHMETIC`, and `IEEE_EXCEPTIONS`. Use of these modules and the procedures in them ensure portability of programs exploiting features of IEEE arithmetic across platforms. The module `IEEE_ARITHMETIC` contains a `USE` statement for `IEEE_EXCEPTIONS`. Any procedure that uses `IEEE_ARITHMETIC` will have access to the public features of `IEEE_EXCEPTIONS`.

IEEE_FEATURES

This module defines the derived type `IEEE_FEATURES_TYPE` whose components are all private. Its purpose is to express the need for particular IEEE features.

IEEE_FEATURES_TYPE

The only possible values are the following constants:

IEEE_DATATYPE	IEEE data types are available
IEEE_DENORMAL	IEEE denormalized values are supported
IEEE_DIVIDE	IEEE division to the required precision is supported
IEEE_HALTING	control of halting is supported
IEEE_INEXACT_FLAG	inexact exceptions are supported.
IEEE_INF	IEEE infinities (positive and negative) are supported
IEEE_INVALID_FLAG	invalid exceptions are supported
IEEE_NAN	IEEE NaN (Not a Number) values are supported
IEEE_ROUNDING	all IEEE rounding modes are supported
IEEE_SQRT	SQRT is supported to the IEEE standard
IEEE_UNDERFLOW_FLAG	underflow exceptions supported

IEEE_ARITHMETIC

This module defines the two derived types `IEEE_CLASS_TYPE` and `IEEE_ROUND_TYPE` whose components are all private. The purpose of `IEEE_CLASS_TYPE` is to identify the class of a value. The purpose of `IEEE_ROUND_TYPE` is to specify or inquire the rounding mode. This module also defines two elemental operators for each of these types: `==` and `/=`. The `==` operator returns true if two values of these types are equal and false if they are not. The `/=` operator returns true if two values of these types are not equal and false if they are equal.

The `IEEE_ARITHMETIC` module further provides a number of subroutines and functions for inquiry, performing operations, and setting the IEEE rounding environment.

IEEE_CLASS_TYPE

The only possible values are the following constants:

IEEE_SIGNALING_NAN	NaN (Not a Number)
IEEE_QUIET_NAN	NaN (Not a Number)
IEEE_NEGATIVE_INF	negative infinity
IEEE_NEGATIVE_NORMAL	negative number
IEEE_NEGATIVE_DENORMAL	negative number smaller than the normal representation
IEEE_NEGATIVE_ZERO	negative zero
IEEE_POSITIVE_ZERO	zero
IEEE_POSITIVE_DENORMAL	positive number smaller than the normal representation
IEEE_POSITIVE_NORMAL	positive number
IEEE_POSITIVE_INF	positive infinity

IEEE_ROUND_TYPE

The only possible values are the following constants:

IEEE_NEAREST

IEEE_TO_ZERO

IEEE_UP

IEEE_DOWN

Subroutines and Functions

The `IEEE_ARITHMETIC` module provides the following *inquiry* functions:

```
logical function IEEE_SUPPORT_DATATYPE(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if the data type of the argument is supported in conformance with section 14.8 of the Fortran 2003 Draft Standard.

```
logical function IEEE_SUPPORT_DENORMAL(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if denormalized numbers are supported for the data type of the argument.

```
logical function IEEE_SUPPORT_DIVIDE(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if division to the accuracy specified by the IEEE standard is supported for the data type of the argument.

```
logical function IEEE_SUPPORT_INF(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if infinities numbers are supported for the data type of the argument.

```
logical function IEEE_SUPPORT_IO(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if IEEE rounding is supported during formatted input and output conversions for the data type of the argument.

```
logical function IEEE_SUPPORT_NAN(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if NaN (Not a Number) values are supported for the data type of the argument.

```
logical function IEEE_SUPPORT_ROUNDING(round_value, x)
type(IEEE_ROUND_TYPE), intent(in) :: round_value
real(kind=SP), intent(in), optional :: x
```

returns the value `.TRUE.` if the specified rounding type is supported for the data type of the argument.

```
logical function IEEE_SUPPORT_SQRT(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if `SQRT` to the accuracy specified by the IEEE standard is supported for the data type of the argument.

```
logical function IEEE_SUPPORT_STANDARD(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if all capabilities and facilities specified by the IEEE standard are supported for the data type of the argument.

```
logical function IEEE_SUPPORT_UNDERFLOW(x)
real(kind=any), intent(in), optional :: x
```

returns the value `.TRUE.` if control of underflow mode is supported for the data type of the argument. Control of underflow mode allows specifying gradual or abrupt underflow. On some processors, chopped underflow can produce code that executes faster.

The `IEEE_ARITHMETIC` module provides the following *elemental* functions for data values that `IEEE_SUPPORT_DATATYPE(x)` returns `.TRUE.`:

```
elemental type(IEEE_CLASS_TYPE) function IEEE_CLASS(x)
real(kind=any), intent(in) :: x
```

returns the class of the input argument `x`.

```
elemental real function IEEE_COPY_SIGN(x, y)
real(kind=any), intent(in) :: x, y
```

returns a result that has the value of x and the sign of y, even when x is a special value such as NaN or infinity.

```
elemental logical function IEEE_IS_FINITE(x)
real(kind=any), intent(in) :: x
```

returns the value .TRUE. if the value of x is finite. That is its class is one of:

```
IEEE_NEGATIVE_NORMAL         IEEE_POSITIVE_DENORMAL
IEEE_NEGATIVE_DENORMAL       IEEE_POSITIVE_NORMAL
IEEE_NEGATIVE_ZERO           IEEE_POSITIVE_ZERO
```

```
elemental logical function IEEE_IS_NAN(x)
real(kind=any), intent(in) :: x
```

returns the value .TRUE. if the value of x is a NaN.

```
elemental logical function IEEE_IS_NORMAL(x)
real(kind=any), intent(in) :: x
```

returns the value .TRUE. if the value of x is normal. That is its class is one of:

```
IEEE_NEGATIVE_NORMAL         IEEE_POSITIVE_NORMAL
IEEE_NEGATIVE_ZERO           IEEE_POSITIVE_ZERO
```

```
elemental logical function IEEE_IS_NEGATIVE(x)
real(kind=any), intent(in) :: x
```

returns the value .TRUE. if the value of x is negative. That is its class is one of:

```
IEEE_NEGATIVE_NORMAL         IEEE_NEGATIVE_ZERO
IEEE_NEGATIVE_DENORMAL       IEEE_NEGATIVE_INFINITY
```

```
elemental real function IEEE_LOGB(x)
real(kind=4 or kind=8), intent(in) :: x
```

returns the unbiased exponent of x as a floating-point value. If x is 0.0, -infinity is returned and IEEE_DIVIDE_BY_ZERO is signaled.

```
elemental real function IEEE_NEXT_AFTER(x, y)
real(kind=4 or kind=8), intent(in) :: x, y
```

returns the neighbor of x in the direction of y. The *kind* of the result is the same as x. If x is 0.0, the result is the smallest denormalized number.

```
elemental real function IEEE_REM(x, y)
real(kind=4 or kind=8), intent(in) :: x, y
```

returns the exact remainder of x/y. The function is defined as x-y*n where n is the nearest integer to x/y. If |n-x/y|=½, n is even. The *kind* of the result is the same as x.

```
elemental real function IEEE_RINT(x)
real(kind=4 or kind=8), intent(in) :: x
```

returns the rounded to integer value of x. x is rounded according to the current rounding mode.

```
elemental real function IEEE_SCALB(x, i)
real(kind=4 or kind=8), intent(in) :: x
integer (kind=4), intent(in) :: i
```

returns the floating-point value of x*2**i. The *kind* of the result is the same as x.

```
elemental logical function IEEE_UNORDERED(x, y)
real(kind=any), intent(in) :: x, y
```

returns if either x or y is a NaN.

```
elemental real function IEEE_VALUE(x, class)
real(kind=any), intent(in) :: x
type(IEEE_CLASS_TYPE), intent(in) :: class
```

returns a value of the specified IEEE_CLASS_TYPE.

The IEEE_ARITHMETIC module provides the following non-*elemental* subroutines:

```
subroutine IEEE_GET_ROUNDING_MODE(round_value)
type(IEEE_ROUND_TYPE), intent(out) :: round_value
```

the current rounding mode is returned in the round_value variable.

```
subroutine IEEE_GET_UNDERFLOW_MODE(gradual)
logical(kind=4). Intent(out) :: gradual
```

the current underflow mode (gradual/abrupt) is returned in the gradual variable. If the mode is gradual, the value of gradual of will be set to .TRUE.. If IEEE_SUPPORT_UNDERFLOW_CONTROL(x) returns .FALSE. this subroutine will produce a runtime error and should not be called.

```
subroutine IEEE_SET_ROUNDING_MODE(round_value)
type(IEEE_ROUND_TYPE), intent(in) :: round_value
```

the rounding mode is set to the mode specified in the `round_value` variable.

```
subroutine IEEE_SET_UNDERFLOW_MODE(gradual)
logical(kind=4), intent(in) :: gradual
```

the underflow mode is set to gradual if the value of the `gradual` variable is `.TRUE.`. If the value of the `gradual` variable is `.FALSE.`, the underflow mode is set to abrupt. If `IEEE_SUPPORT_UNDERFLOW_CONTROL(x)` returns `.FALSE.` this subroutine will produce a runtime error and should not be called.

The `IEEE_ARITHMETIC` module provides the following *kind* function:

```
integer(kind=4) function IEEE_SELECTED_REAL_KIND(p, r)
integer(kind=4), intent(in), optional :: p
integer(kind=4), intent(in), optional :: r
```

returns the *kind* of an IEEE floating-point value with the requested precision and exponent range. This function is similar to `SELECTED_REAL_KIND`, but only returns IEEE reals.

IEEE_EXCEPTIONS

This module defines the two derived types `IEEE_FLAG_TYPE` and `IEEE_STATUS_TYPE` whose components are all private. The purpose of `IEEE_FLAG_TYPE` is to identify the exception flags. The purpose of `IEEE_STATUS_TYPE` is to save and restore the current floating-point environment.

The `IEEE_EXCEPTIONS` module also provides a number of subroutines and functions for inquiry and getting and setting exception flags.

An important feature of exception flags is that they may or may not halt execution of the program depending on the state of the halting modes. A floating-point operation (such as divide-by-zero) will cause an exception flag to signal, but unless the halting mode for that flag is set to true, execution will continue with an IEEE default value. The `IEEE_GET_FLAG` subroutine can be used to retrieve the current state (signaling or quiet) of a specific exception flag.

IEEE_FLAG_TYPE

The only possible values are the following constants:

IEEE_INVALID

IEEE_OVERFLOW

IEEE_DIVIDE_BY_ZERO

IEEE_UNDERFLOW

IEEE_INEXACT

and the array constants:

IEEE_USUAL

```
type(IEEE_FLAG_TYPE), parameter, dimension(3) :: IEEE_USUAL = &
        (/IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID/)
```

IEEE_ALL

```
type(IEEE_FLAG_TYPE), parameter, dimension(5) :: IEEE_ALL = &
                 (/IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT/)
```

IEEE_STATUS_TYPE

This type is used to save and restore the floating-point environment.

Subroutines and Functions

The `IEEE_EXCEPTIONS` module provides the following *elemental* subroutines:

```
elemental subroutine IEEE_GET_FLAG(flag, flag_value)
type(IEEE_FLAG_TYPE), intent(in) :: flag
logical(kind=4), intent(out) :: flag_value
```

retrieves the state of the specified flag. If the `flag` is signaling, `flag_value` is set to `.TRUE.`. If the `flag` is quiet, `flag_value` is set to `.FALSE.`.

```
elemental subroutine IEEE_GET_HALTING_MODE(flag, halting)
type(IEEE_FLAG_TYPE), intent(in) :: flag
logical(kind=4), intent(out) :: halting
```

retrieves the halting mode of the specified flag. If the `flag` mode is halting, `halting` is set to `.TRUE.`. If the `flag` mode is continue, `halting` is set to `.FALSE.`.

The `IEEE_EXCEPTIONS` module provides the following non-*elemental* subroutines:

```
subroutine IEEE_GET_STATUS(status_value)
type(IEEE_STATUS_TYPE) :: status_value
```

retrieves the state of the floating-point environment.

```
subroutine IEEE_SET_FLAG(flag, flag_value)
type(IEEE_FLAG_TYPE), intent(in) :: flag
logical(kind=4), intent(in) :: flag_value
```

sets the state of the specified flag. If `flag_value` is `.TRUE.`, the `flag` is set to signaling. If `flag_value` is `.FALSE.`, the `flag` is set to quiet.

```
elemental subroutine IEEE_SET_HALTING_MODE(flag, halting)
type(IEEE_FLAG_TYPE), intent(in) :: flag
logical(kind=4), intent(in) :: halting
```

sets the halting mode of the specified flag. If `halting` is `.TRUE.`, the `flag` mode is set to halting. If `halting` is `.FALSE.`, the `flag` mode is set to continue.

```
subroutine IEEE_SET_STATUS(status_value)
type(IEEE_STATUS_TYPE) :: status_value
```

sets the state of the floating-point environment.

EXAMPLES

The following example demonstrates the sequence necessary to exercise program control over detection of an exception.

```
subroutine safe_divide(a, b, c, fail)
use IEEE_EXCEPTIONS
real a, b, c
logical fail
type(IEEE_STATUS_TYPE) status

! save the current floating-point environment, turn halting for
! divide-by-zero off, and clear any previous divide-by-zero flag
call IEEE_GET_STATUS(status)
call IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO, .false.)
call IEEE_SET_FLAG(IEEE_DIVIDE_BY_ZERO, .false.)

! perform the operation
c = a/b

! determine if a failure occurred and restore the floating-point
! environment
call IEEE_GET_FLAG(IEEE_DIVIDE_BY_ZERO, fail)
call IEEE_SET_STATUS(value)

end subroutine safe_divide
```

Appendix C

Language Binding Modules

ISO_FORTRAN_ENV MODULE

The ISO_FORTRAN_ENV provides constants and functions describing the Fortran environment to assist in porting source code between different Fortran compilers. All constants in the module are default integer kind, INTEGER(KIND=4), values except the CHARACTER functions COMPILER_VERSION() and COMPILER_OPTIONS().

NAME	DESCRIPTION
CHARACTER_STORAGE_SIZE	Size in bits of a character
COMPILER_VERSION()	Character scalar result describing the compiler version at the compile time
COMPILER_OPTIONS()	Character scalar result describing the compiler options at the compile time
ERROR_UNIT	Preconnected unit used for error reporting
FILE_STORAGE_SIZE	Size in bits of the file storage unit
INPUT_UNIT	Preconnected unit used for input
INT8	Kind of 8 bit integer
INT16	Kind of 16 bit integer
INT32	Kind of 32 bit integer
INT64	Kind of 64 bit integer
INTEGER_KINDS	Rank 1 integer array of available integer kinds
IOSTAT_END	Value of IOSTAT= on end-of-file
IOSTAT_EOR	Value of IOSTAT= on end-of-record
LOGICAL_KINDS	Rank 1 integer array of available logical kinds
NUMERIC_STORAGE_SIZE	Size in bits of numeric storage unit
OUTPUT_UNIT	Preconnected unit used for output
REAL32	Kind of 32 bit real
REAL64	Kind of 64 bit real
REAL128	Kind of 128 bit real
REAL_KINDS	Rank 1 integer array of available real kinds

ISO_C_BINDING MODULE

The ISO_C_BINDING module, part of the Fortran 2003 standard, provides constants, types, and procedures that are useful in programs which are built using a mix of C and Fortran routines.

KIND Parameters

The KIND parameters contained in the ISO_C_BINDING module provide interoperability between Fortran intrinsic types and C types. The KIND parameters are all of type INTEGER(KIND=1) and are summarized in the table below.

FORTRAN TYPE	KIND NAME	C TYPE
INTEGER	C_INT	int
	C_SHORT	short
	C_LONG	long
	C_LONG_LONG	long long
	C_SIGNED _CHAR	signed char
	C_SIZE_T	size_t
FORTRAN TYPE	KIND NAME	C TYPE
INTEGER	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intprt_t
	C_PTRDIFF_T	intprt_t
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_CONPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char

CHARACTER CONSTANTS

The ISO_C_BINDING module provides character constants for C character values used as escape sequences. These constants are of type CHARACTER(LEN=1) and are summarized in the table below.

NAME	DEFINITION	C CHARACTER
C_NULL_CHAR	null character	'\0'
C_ALERT	alert	'\a'
C_BACKSPACE	backspace	'\b'
C_FORM_FEED	form feed	'\f'
C_NEW_LINE	new line	'\n'
C_CARRIAGE_RETURN	carriage return	'\r'
C_HORIZONTAL_TAB	horizontal tab	'\t'
C_VERTICAL_TAB	vertical tab	'\v'

POINTER CONSTANTS

The ISO_C_BINDING module defines the constant C_NULL_PTR that is of type C_PTR and has the value of a C null pointer. The module also defines the constant C_NULL_FUNPTR that is of type C_FUNPTR and has the value of C null function pointer.

TYPE DEFINITIONS

The ISO_C_BINDING module defines two data types: C_PTR and C_FUNPTR. The C_PTR type is interoperable with any C data pointer. The C_FUNPTR type is interoperable with a C function pointer. The components of both types are private and not visible to code that uses the ISO_C_BINDING module.

PROCEDURES

The ISO_C_BINDING module provides five procedures to facilitate interoperability with the C language: C_ASSOCIATED, C_F_POINTER, C_F_PROCPOINTER, C_FUNLOC and C_LOC.

The C_ASSOCIATED function is use to determine if a variable of type C_PTR or C_FUNPTR has been pointer associated. It is also used to compare the values of two variables of type C_PTR (or two variables of type C_FUNPTR) for equality.

```
logical function c_associated(c_ptr1,c_ptr2)
type(C_PTR), intent(in) :: cptr1
type(C_PTR), intent(in), optional :: c_ptr2
```

```
logical function c_associated(c_fun_ptr1,c_fun_ptr2)
type(C_FUNPTR), intent(in) :: c_fun_ptr1
type(C_FUNPTR), intent(in), optional :: c_fun_ptr2
```

When called with one argument, C_ASSOCATED returns the value .FALSE. if the argument is a C null pointer and .TRUE. otherwise. When called with two arguments, C_ASSOCIATED returns the value .TRUE. if the arguments compare equal to each other and .FALSE. otherwise.

The C_F_POINTER subroutine is used to associates a Fortran variable of type C_PTR with a C pointer variable.

```
subroutine C_F_POINTER(cptr,fptr,shape)
type(C_PTR), intent(in) :: cptr
any type, pointer, intent(out) :: fptr
integer, dimension( : ),intent(in),optional :: shape
```

The value of cptr is the address of an interoperable data entity or the result of a reference to the C_LOC function with a non-interoperable argument. The value of cptr must not be the C address of a Fortran variable that does not have the target attribute.

The argument fptr is a pointer. If the value of cptr is the C address of an interoperable data entity, fptr must be a data pointer with type and type parameters that are interoperable. In this case, fptr becomes pointer associated with the target of cptr. If fptr is an array, its shape is specified by shape and each lower bound is 1. If the value of cptr is the result of a reference to C_LOC with a noninteroperable argument X, fptr must be a scalar pointer with the same type and type parameters as X and fptr becomes pointer associated with X.

The argument shape is present if and only if fptr is an array and its size is equal to the rank of fptr.

The C_F_PROCPOINTER subroutine associates a Fortran variable of type C_FUNPTR with the address of a C function.

```
subroutine C_F_PROCPOINTER(cptr,fptr)
type(C_FUNPTR), intent(in) :: cptr
integer(kind=intptr_t), intent(out) :: fptr
```

The value of `cptr` is the address of a C procedure which is interoperable and `fptr` is type integer with size large enough to hold the value of `cptr`.

The C_FUNLOC function returns the address of an interoperable C procedure as a C_FUNPTR.

```
type(C_FUNPTR) function C_FUNLOC(x)
integer(kind=intptr_t), intent(in) :: x
```

The value of x is the address of a procedure which is interoperable. The result is a value that can be used in a call to C_F_PROCPOINTER.

The C_LOC function returns the C address of its argument.

```
type(C_PTR) function C_LOC(x)
any type, intent(in) :: x
```

The argument `x` has interoperable type and type parameters and is a variable that has the TARGET attribute, is an allocated allocatable array that has the TARGET attribute and not of zero size, or is an associated scalar pointer. The argument `x` may also be a scalar with no length type parameters and be a non-allocatable, non-pointer variable that has the TARGER attribute, an allocated allocatable variable that has the TARGET attribute, or an associated pointer.

The C_SIZEOF function returns the number of bytes of storage used by its argument.

```
integer(C_SIZE_T) function C_SIZEOF(x)
any type, intent(in) :: x
```

If the argument x has the POINTER attribute, the function result is the number of bytes the argument points to. If the argument is a derived type with ALLOCATABLE or POINTER components, the result does not include the sizes of the allocated or pointed to storage for these components.

Appendix D

Error Messages

his appendix lists runtime error numbers and their meanings. These numbers are assigned to the `STAT=` specifier in `ALLOCATE` statements and to the `IOSTAT=` specifier variable in I/O statements.

ALLOCATE statement errors

10045	attempt to `DEALLOCATE` an unallocated array
10046	attempt to `DEALLOCATE` an item with a modified size
10047	attempt to `ALLOCATE` an array that has already been allocated
10070	attempt to allocate array of requested size failed

FORTRAN I/O errors:

10000	File not open for read
10001	File not open for write
10002	File not found
10003	Record length negative or 0
10004	Buffer allocation failed
10005	Bad iolist specifier
10006	Error in format string
10007	Illegal repeat count
10008	Hollerith count exceeds remaining format string
10009	Format string missing opening "("
10010	Format string has unmatched parens
10011	Format string has unmatched quotes
10012	Non-repeatable format descriptor
10013	Attempt to read past end of file
10014	Bad file specification
10015	Format group table overflow
10016	Illegal character in numeric input
10017	No record specified for direct access
10018	Maximum record number exceeded
10019	Illegal file type for namelist directed I/O
10020	Illegal input for namelist directed I/O
10021	Variable not present in current namelist
10022	Variable type or size does not match edit descriptor
10023	Illegal direct access record number
10024	Illegal use of internal file
10025	`RECL=` only valid for direct access files

10026	BLOCK= only valid for unformatted sequential files
10027	Unable to truncate file after rewind, backspace, or endfile
10028	Can't do formatted I/O on an entire structure
10029	Illegal (negative) unit specified
10030	Specifications in re-open do not match previous open
10031	No implicit OPEN for direct access files
10032	Cannot open an existing file with STATUS='NEW'
10033	Command not allowed for unit type
10034	MRWE is required for that feature
10035	Bad specification for window
10036	Endian specifier not BIG_ENDIAN or LITTLE_ENDIAN
10037	Cannot ENDIAN convert entire structures
10038	Attempt to read past end of record
10039	Attempt to read past end of record in non-advancing I/O
10040	Illegal specifier for ADVANCE=
10041	Illegal specifier for DELIM=
10042	Illegal specifier for PAD=
10043	SIZE= specified with ADVANCE=YES
10044	EOR= specified with ADVANCE=YES
10045	Cannot DEALLOCATE disassociated pointer or unallocated array
10046	Cannot DEALLOCATE a portion of an original allocation
10047	An allocatable array has already been allocated
10048	Internal or unknown runtime library error
10049	Unknown data type passed to runtime library
10050	Illegal DIM argument to array intrinsic
10051	Size of SOURCE argument to RESHAPE smaller than SHAPE array
10052	SHAPE array for RESHAPE contains a negative value
10053	Unallocated or disassociated array passed to inquiry function
10054	The ncopies argument to REPEAT is negative
10055	The S argument to NEAREST is negative
10056	The ORDER argument to RESHARE contains an illegal value
10057	Result of TRANSFER with no SIZE is smaller than source
10058	SHAPE array for RESHAPE is zero sized array
10059	VECTOR argument to UNPACK contains insufficient values
10060	Attempt to write a record longer than specified record length
10061	ADVANCE= specified for direct access or unformatted file
10062	NAMELIST name is longer than specified record length
10063	NAMELIST variable name exceeds maximum length
10064	PAD= specified for unformatted file
10065	NAMELIST input contains multiple strided arrays
10066	Expected & or $ as first character for NAMELIST input
10067	NAMELIST group does not match current input group
10068	Pointer or allocatable array not associated or allocated
10069	NAMELIST input contains negative array stride
10070	Runtime memory allocation fails
10071	Illegal rank for matrix argument to MATMUL array intrinsic
10072	Matrix arguments to MATMUL array intrinsic are not conformable

10073	Non-conformal flag value array or conflicting flag values
10074	Control of gradual/abrupt underflow is not supported in environment
10075	Infinite loop detected in execution of I/O implied do loop
10076	Data transfer format descriptor was not found during format reversion

Appendix E

Technical Support

The Absoft Technical Support Group will provide technical assistance to all registered users. They will *not* answer general questions about operating systems, operating system interfaces, graphical user interfaces, or teach the FORTRAN language. For further help on these subjects, please consult this manual and any of the books and manuals listed in the bibliography.

Before contacting Technical Support, please study this manual and the *Fortran User Guide* to make sure your problem is not covered here. Specifically, look at the chapter **Using The Compilers** in the *Pro Fortran User Guide* and the **Error Messages** appendices of both manuals. To help Technical Support provide a quick and accurate solution to your problem, please include the following information in any correspondence or have it available when calling.

Product Information:

Name of product .
Version number.
Serial number.
Version number of the operating system.

System Configuration:

Hardware configuration (hard drive, etc.).
System software release (i.e. 4.0, 3.5, etc).
Any software or hardware modifications to your system.

Problem Description:

What happens?
When does it occur?
Provide a small (20 line) reproducible program or step-by-step example if possible.

Contacting Technical Support:

Address: Absoft Corporation
Attn: Technical Support
2111 Cass Lake Road, Suite 102
Keego Harbor, MI 48324

Technical Support:	(248) 220-1191	9am - 3pm EST
FAX	(248) 220-1194	24 Hours
email	support@absoft.com	24 Hours
World Wide Web	http://www.absoft.com	

Appendix F

Floating Point Numbers

A floating point number is an approximation of a real number. The model for floating point numbers is:

$$v = s \times r^e \times \sum_{i=0}^{q} d_i \times r^{i-1}$$

The most common implementation is IEEE P754 single and double precision, where s is ±1, q is 24 or 53, r is 2, d is 0 or 1, and -126≤e≤127 or -1022≤e≤1023.

The range of representable single precision number is approximately:

`±0.3402823E+39` to `±0.1175494E-37`

and the range of representable double precision number is approximately:

`±0.1797693134862320D+309` to `±0.2225073858507202D-307`

The number of values in these ranges is finite and not all numbers are representable. For example, just as 1/3 cannot be stated exactly as a decimal, 1/10 or 0.1 cannot represented exactly in IEEE floating point. Because of this potential for inexact representation of real values, the equality relational operators (**==** and **/=**) may at times yield “unreliable” results and thus should be used with care (if at all) with real operands.

Consider the `SQRT` function. The square roots of the values in the interval (.25,1.0) map into the interval (0.5,1.0). There are twice as many numbers in the first interval as there are in the second, so on average, the square roots of 2 numbers in the first interval must map into one number in the second.

Floating point numbers have a finite number of digits, either binary or decimal. The $\log_2 10$ is about 3.322 so about 3.322 binary digits are needed to represent a decimal digit.

REAL(4) has a 24 bit mantissa => 7.2 decimal digits
REAL(8) has a 53 bit mantissa => 15.9 decimal digits

The results of operations on numbers with large exponent differences may not be representable. For example, both of the single precision numbers 2.0×10^{5} and 2.0×10^{-5} are representable, but their sum is not:

```
   200000.0
 +      0.00002
   200000.0
```

```
        a = 2.0*10.0**(+5)
        b = 2.0*10.0**(-5)
        c = a+b
        print *,a, b, c
        print *,a /= a+b, a /= c
        end

   200000.  2.000000E-05  200000.
  T F
```

Loss of precision with floating point numbers is a significant problem with numerically sensitive programs. Consider the sine function. It is defined over the interval (-π/2,π/2) and returns values in the interval (0.0,1.0). Range reduction is automatically performed for arguments outside of the interval. However, although range reduction is carefully performed, loss of significant digits can occur to the extent that the function result becomes useless.

```
      integer, parameter :: rp = selected_real_kind(6,30)
      real(rp) halfpi, a
      halfpi = atan(1.0_rp)*2.0_rp
      do i=1,7
        a = halfpi * 10.0_rp**i
        print 100,i,a,sin(a)
      end do
100   format (i2, f18.6, f10.6)
      end

 1         15.707963 -0.000001
 2        157.079630  0.000003
 3       1570.796300  0.000060
 4      15707.963000  0.000599
 5     157079.640000  0.007945
 6    1570796.300000  0.048186
 7   15707964.000000  0.668397
 8  157079630.000000 -0.628396
 9 1570796400.000000  0.945949
10 15707963000.000000 -0.999123
```

Appendix G

Extensions To Intrinsic Functions

The functions listed in the following table are historical extensions to the Fortran programming language. They are provided as a convenience to porting legacy programs. Their use is not recommended for new programs, as they are non-standard.

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Type Conversion			
IINT	INT	IINT(x)	real	integer*2	
JINT	INT	JINT(x)	real	integer	
KINT	INT	KINT(x)	real	integer*8	
IIFIX	INT	IIFIX(x)	real	integer*2	
HFIX	INT	HFIX(x)	real	integer*2	
KFIX	INT	KFIX(x)	real	integer*8	
JIFIX	INT	JIFIX(x)	real	integer	
IIDINT	INT	IIDINT(d)	double	integer*2	
JIDINT	INT	JIDINT(d)	double	integer	
FLOATI	REAL	FLOATI(i)	integer*2	real	
FLOATJ	REAL	FLOATJ(i)	integer	real	
FLOATK	REAL	FLOATK(i)	integer*8	real	
DREAL	DREAL	DREAL(x)	any	double	
DFLOTI	DBLE	DFLOTI(i)	integer*2	double	
DFLOTJ	DBLE	DFLOTJ(i)	integer	double	
DFLOTK	DBLE	DFLOTK(i)	integer*8	double	
DCMPLX	DCMPLX	DCMPLX(x)	any	complex*16	
		Nearest Integer			
ININT	NINT	ININT(r)	real	integer*2	
JNINT	NINT	JNINT(r)	real	integer	
KNINT	NINT	KNINT(r)	real	integer*8	
IIDNNT	NINT	IIDNNT(d)	double	integer*2	
JIDNNT	NINT	JIDNNT(d)	double	integer	
KIDNNT	NINT	KDNINT(d)	double	integer*8	
		Absolute Value			
IIABS	ABS	IIABS(i)	integer*2	integer*2	
JIABS	ABS	JIABS(i)	integer	integer	
KIABS	ABS	KIABS(i)	integer*8	integer*8	
CDABS	ABS	CDABS(cd)	complex*16	double	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Remaindering			
IMOD	MOD	IMOD(i,j)	integer*2	integer*2	
JMOD	MOD	JMOD(i,j)	integer	integer	
KMOD	MOD	KMOD(i,j)	integer*8	integer*8	
		Transfer of Sign			
IISIGN	SIGN	IISIGN(i,j)	integer*2	integer*2	
JISIGN	SIGN	JISIGN(i,j)	integer	integer	
KISIGN	SIGN	KISIGN(i,j)	integer*8	integer*8	
		Positive Difference			
IIDIM	DIM	IIDIM(i,j)	integer*2	integer*2	
JIDIM	DIM	JIDIM(i,j)	integer	integer	
KIDIM	DIM	KIDIM(i,j)	integer*8	integer*8	
		Choosing Largest Value			
IMAX0	MAX	IMAX0(i,j,…)	integer*2	integer*2	
JMAX0	MAX	JMAX0(i,j,…)	integer	integer	
KMAX0	MAX	KMAX0(i,j,…)	integer*8	integer*8	
AIMAX0		AIMAX0(i,j,…)	integer*2	real	
AJMAX0		AIMAX0(i,j,…)	integer	real	
IMAX1		IMAX1(r,s,…)	real	integer*2	
JMAX1		JMAX1(r,s,…)	real	integer	
KMAX1		KMAX1(r,s,…)	real	integer*8	
		Choosing Smallest Value			
IMIN0	MIN	IMIN0(i,j,…)	integer*2	integer*2	
JMIN0	MIN	JMIN0(i,j,…)	integer	integer	
KMIN0	MIN	KMIN0(i,j,…)	integer*2	integer*2	
AIMIN0		AIMIN0(i,j,…)	integer*2	real	
AJMIN0		AIMIN0(i,j,…)	integer	real	
IMIN1		IMIN1(r,s,…)	real	integer*2	
JMIN1		JMIN1(r,s,…)	real	integer	
KMIN1		KMIN1(r,s,…)	real	integer*8	
		Imaginary Part of Complex			
DIMAG		DIMAG(cd)	complex*16	double	
		Conjugate of Complex			
DCONJG		DCONJG(cd)	complex*16	complex*16	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
Square Root					
CDSQRT	SQRT	CDSQRT(cd)	complex*16	complex*16	
Exponential					
CDEXP	EXP	CDEXP(cd)	complex*16	complex*16	
Natural Logarithm					
CDLOG	LOG	CDLOG(cd)	complex*16	complex*16	
Common Logarithm					
DLOG10	LOG10	DLOG10(d)	double	double	
Sine					
SIND	SIND	SIND(r)	real	real	
DSIND	SIND	DSIND(d)	double	double	
CDSIN	SIN	CDSIN(cd)	complex*16	complex*16	
Cosine					
COSD	COSD	COSD(r)	real	real	
DCOSD	COSD	DCOSD(d)	double	double	
CDCOS	COS	CDCOS(cd)	complex*16	complex*16	
Tangent					
TAND	TAND	TAND(r)	real	real	
DTAND	TAND	DTAND(d)	double	double	
Arcsine					
ASIND	ASIND	ASIND(r)	real	real	
DASIND	ASIND	DASIND(d)	double	double	
Arccosine					
ACOSD	ACOSD	ACOSD(r)	real	real	
DACOSD	DACOSD	DACOSD(d)	double	double	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Arctangent			
ATAND	ATAND	ATAND(r)	real	real	
DATAND	ATNAD	DATAND(d)	double	double	
ATAN2D	ATAN2D	ATAN2D(r,s)	real	real	
DATAN2D	ATAN2D	DATAN2D(d,e)	double	double	
		Zero Extension			
ZEXT	ZEXT	ZEXT(i)	integer	integer	
IZEXT	ZEXT	IZEXT(i)	integer*2	integer	
JZEXT	ZEXT	JZEXT(i)	integer	integer	
KZEXT	ZEXT	KZEXT(i)	integer*8	integer	
		Pass By Value			
[%]VAL		[%]VAL(a)	any	any	
[%]VAL4		[%]VAL4(a)	any	any	
[%]VAL2		[%]VAL2(a)	any integer	integer*2	
[%]VAL1		[%]VAL1(a)	any integer	integer*1	
		Pass By Reference			
[%]REF		[%]REF(a)	any	any	
		Pass By Descriptor			
[%]DESCR		[%]DESCR(a)	any	any	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
Bitwise Operations					
SHIFT		SHIFT(i,j)	integer	integer	
IISHFT		IISHFT(i,j)	integer*2	integer*2	
JISHFT		JISHFT(i,j)	integer	integer	
KISHFT		KISHFT(i,j)	integer*8	integer*8	
IISHFTC		IISHFTC(i,j,k)	integer*2	integer*2	
JISHFTC		JISHFTC(i,j,k)	integer	integer	
KISHFTC		KISHFTC(i,j,k)	integer*8	integer*8	
IOR		IOR(i,j)	integer	integer	
IIOR		IIOR(i,j)	integer*2	integer*2	
JIOR		JIOR(i,j)	integer	integer	
KIOR		KIOR(i,j)	integer*8	integer*8	
IIAND		IIAND(i,j)	integer*2	integer*2	
JIAND		JIAND(i,j)	integer	integer	
KIAND		KIAND(i,j)	integer*8	integer*8	
INOT		INOT(i)	integer*2	integer*2	
JNOT		JNOT(i)	integer	integer	
KNOT		KNOT(i)	integer*8	integer*8	
IEOR		IEOR(i,j)	integer	integer	
IIEOR		IIEOR(i,j)	integer*2	integer*2	
JIEOR		JIEOR(i,j)	integer	integer	
KIEOR		KIEOR(i,j)	integer*8	integer*8	
IIBITS		IIBITS(i,j,k)	integer*2	integer*2	
JIBITS		JIBITS(i,j,k)	integer	integer	
KIBITS		KIBITS(i,j,k)	integer*8	integer*8	
BITEST		BITEST(i,j)	integer*2	logical*2	
BJTEST		BJTEST(i,j)	integer	logical	
IIBSET		IIBSET(i,j)	integer*2	integer*2	
JIBSET		JIBSET(i,j)	integer	integer	
KIBSET		KIBSET(i,j)	integer*8	integer*8	
IIBCLR		IIBCLR(i,j)	integer*2	integer*2	
JIBCLR		JIBCLR(i,j)	integer	integer	
KIBCLR		KIBCLR(i,j)	integer*8	integer*8	

/ editing, 132
\ editing, 132
\f, 19
\n, 19
\t, 19
`3 editing, 132
`4 editing, 133
=>, 37
A editing, 129
ABS, 156
Absoft address, 216
ACCEPT, 109
ACCESS, 110, 117
ACHAR, 156
ACOS, 156
ACTION, 110
ADJUSTL, 156
ADJUSTR, 156
ADVANCE, 106
AIMAG, 156
AINT, 157
ALL, 157
ALLOCATABLE, 40
ALLOCATE error messages, 211
ALLOCATE statement, 63
ALLOCATED, 157
ampersand, 12
ANINT, 157
ANSI standard, 9
ANY, 158
apostrophe editing, 133
arithmetic
    assignment statement, 81
    constant expression, 77
    expressions, 75
    IF statement, 84
arithmetic expressions
    data type, 76
array, 59
    actual, 60
    adjustable, 60
    dummy, 60
    storage sequence, 62
    subscript, 60
array conformance, 67
array constructor, 66
array declarator, 59
array operations, 68
array rank, 60
array sections, 65
array shape, 60
array valued functions, 68
ASCII conversion, 121
ASCII table, 191
ASIN, 158
ASSIGN, 82
assigned GOTO, 83
ASSOCIATED, 39, 158
assumed shape array, 63
ASYNCHRONOUS, 40, 111, 118
ASYNCHRONOUS (I/O specifier), 103
ATAN, 158
ATAN2, 159
AUTOMATIC, 40
automatic array, 63
B editing, 125
backslash editing, 132
BACKSPACE, 114
binary constants, 21
BIT_SIZE, 161
BLANK, 103, 111, 118
blank control editing, 131
BLOCK DATA, 146
block IF, 84
BN editing, 131
BTEST, 161
BZ editing, 131
CALL statement, 140
CARRIAGECONTROL, 112
CASE block, 89
CASE DEFAULT, 89
case selector, 90
CASE statement, 89
CEILING, 161
CHAR, 161
character, 19
    assignment statement, 82
    constant delimiter, 19
    editing, 129
    expressions, 78
    set, 5
    storage unit, 25
    substring, 24
CLOSE, 113
CMPLX, 161
colon editing, 133
comment line, 10
COMMON, 41
    restrictions, 43
COMMON blocks, 145
compatibility, introduction, 1
compiler options
    -f fixed, FORTRAN 77 fixed format, 9
    -f free, Fortran 90 free format, 9
    -f, case fold, 6
    -s, static storage, 27
    -W132, wide format, 9
    -x, conditional compilation, 10, 13
    -YCSLASH=1, escape sequences, 19
COMPLEX, 23
complex editing, 126
`COMPLEX*16`, 34
COMPLEX*8, 32
computed GOTO, 83
conditional compilation, 10, 13, 15
CONJG, 162
constants
    blanks in, 19
    character, 19
    COMPLEX, 23
    Hollerith, 23
    INTEGER, 20

LOGICAL, 20
PARAMETER, 19
real, 22
contacting Absoft, 216
CONTAINS, 42
continuation lines, 11
CONTINUE statement, 89
control statements, 83
conventions used in the manual, 2
CONVERT, 112
COS, 162
COSH, 162
COUNT, 162
CPU_TIME, 162
CSHIFT, 162
CYCLE statement, 88
D. see conditional compilation
D editing, 127
data length specifiers, 30
COMPLEX*16, 34
COMPLEX*8, 32
INTEGER*1, 31
INTEGER*2, 31
INTEGER*4, 31
INTEGER*8, 31
LOGICAL*1, 31
LOGICAL*2, 30
LOGICAL*4, 30
REAL*4, 32
REAL*8, 32
DATA statement, 57
data type, 18
character, 19
COMPLEX, 23
Hollerith, 23
IMPLICIT, 18
INTEGER, 20
intrinsic function, 18
LOGICAL, 20
name, 18
real, 22
DATE_AND_TIME, 163
DBLE, 163
DC editing, 131
DEALLOCATE statement, 63
DECIMAL, 104, 111, 118
decimal constants, 21
decimal symbol editing, 131
declaration initialization, 30
DECODE, 121
deferred shape array, 63
DELIM, 104, 111, 118
derived types, 35
DESCR function, 222
DIGITS, 164
DIM, 164
DIMENSION, 40
dimension bound, 59
dimension declarator, 59
DIRECT, 117
DO, 85, 87
DO variable, 86
DO WHILE, 87
do-construct-name, 89
documentation conventions, 2
dollar editing, 132
DOT_PRODUCT, 164
double precision
editing, 126
DP editing, 131
DPROD, 164
E editing, 127
edit descriptor, 122
elemental functions, 198
elemental subroutines, 202
ELSE, 84
EN editing, 128
ENCODE, 121
ENCODING, 111, 118
END (I/O specifier), 103
END DO, 88
END ENUM, 44
END IF, 84
END MAP statement, 55
END SELECT, 89
END statement, 92
END UNION statement, 55
ENDFILE, 115
endfile record, 98
ENTRY, 144
ENUM, 44
ENUMERATOR, 44
EOSHIFT, 165
EPSILON, 165
EQUIVALENCE
arrays, 43
restrictions, 43
statement, 26, 42
substrings, 43
ERF, 165
ERFC, 165
ERR, 103
error messages, 211
runtime, 211
ES editing, 128
escape sequences, 19
exclamation point, 12, 13
EXIST, 117
EXIT statement, 88
EXP, 166
EXPONENT, 166
expressions, 75
arithmetic, 75
character, 78
relational, 78, 79
extensions to FORTRAN 77, 3
EXTERNAL, 44
external files, 98
external function, 141
F editing, 126
field width, 122
FILE, 110
files, 98
access, 99
internal, 100
name, 98

position, 99
files, including, 15
floating point
editing, 126
floating point numbers, 217
FLOOR, 166
FLUSH, 115
FMT, 102
FORALL statement, 93
FORM, 111, 118
FORMAT, 122
format specification, 122
FORMATTED, 118
formatted data transfer, 109
formatted record, 97
Fortran 77
introduction, 1
FORTRAN 77 extensions, 3, 5, 6, 8, 9, 10, 12, 13, 18, 19, 30, 31, 32, 34, 41, 47, 50, 52, 53, 55, 102, 109, 112, 121, 122, 124, 125, 132, 134, 136, 222
ACCEPT, 109
ACTION specifier, 109
B editing, 126
backslash editing, 132
CARRIAGECONTROL specifier, 112
character set, 5
character type statement, 34
comment, 9
compiler directives, 8
COMPLEX*8, 32
conditional compilation, 10
data length specifiers, 30
declaration initialization, 30
DECODE statement, 121
DESCR function, 222
dollar sign editing, 132
edit descriptors, 122
ENCODE statement, 121
escape sequences, 19
Fortran 90 Free Source Form, 12
GLOBAL statement, 41
Hollerith Constant, 23
IMPLICIT NONE statement, 18
input validation, 124
integer editing, 125
INTEGER*1, 31
INTEGER*2, 31
INTEGER*4, 31
INTEGER*8, 31
intrinsic functions, 47
LOGICAL*1, 31
LOGICAL*2, 30
LOGICAL*4, 30
MAP declaration, 55
Namelist Specifier, 102
namelist directed editing, 136
O editing, 126
POINTER statement, 50
Q editing, 134
READONLY specifier, 112
REAL*4, 32
REAL*8, 32
RECORD statement, 52
REF function, 222
statement field, 9
STRUCTURE declaration, 53
symbolic names, 6
TYPE, 109
UNION declaration, 55
VAL function, 222
VAX FORTRAN Tab-Format, 13
Z editing, 126
ZEXT function, 222
Fortran 90 free source form, 12
FRACTION, 167
FUNCTION statement, 141
functions, 141
external, 141
statement, 142
G editing, 128
GLOBAL, 41
GOTO, statement, 83
graying of text, 3
H editing, 133
hexadecimal constants, 21
Hollerith constant, 23
Hollerith editing, 133
HUGE, 167
I editing, 125
IACHAR, 168
IAND, 168
IBCLR, 168
IBITS, 168
IBSET, 169
ICHAR, 169
ID, 104, 118
IEEE floating point representation, 22
IEEE_ALL, 202
IEEE_CLASS, 198
IEEE_CLASS_TYPE, 200
IEEE_COPY_SIGN, 199
IEEE_DATATYPE, 195
IEEE_DENORMAL, 195
IEEE_DIVIDE, 195
IEEE_DIVIDE_BY_ZERO, 199, 202
IEEE_DOWN, 197
IEEE_FEATURES_TYPE, 195, 196
IEEE_FLAG_TYPE, 201
IEEE_GET_FLAG, 202
IEEE_GET_HALTING_MODE, 203
IEEE_GET_ROUNDING_MODE, 200
IEEE_GET_STATUS, 203
IEEE_GET_UNDERFLOW_MODE, 200
IEEE_HALTING, 195
IEEE_INEXACT, 202
IEEE_INEXACT_FLAG, 195
IEEE_INF, 195
IEEE_INVALID, 202
IEEE_INVALID_FLAG, 195
IEEE_IS_FINITE, 199
IEEE_IS_NAN, 199
IEEE_IS_NEGATIVE, 199
IEEE_IS_NORMAL, 199
IEEE_LOGB, 199
IEEE_NAN, 195

IEEE_NEAREST, 197
IEEE_NEGATIVE_DENORMAL, 196
IEEE_NEGATIVE_INF, 196
IEEE_NEGATIVE_NORMAL, 196
IEEE_NEGATIVE_ZERO, 196
IEEE_NEXT_AFTER, 199
IEEE_OVERFLOW, 202
IEEE_POSITIVE_DENORMAL, 196
IEEE_POSITIVE_INF, 196
IEEE_POSITIVE_NORMAL, 196
IEEE_POSITIVE_ZERO, 196
IEEE_QUIET_NAN, 196
IEEE_REM, 200
IEEE_RINT, 200
IEEE_ROUND_TYPE, 196
IEEE_ROUNDING, 195
IEEE_SCALB, 200
IEEE_SELECTED_REAL_KIND, 201
IEEE_SET_FLAG, 203
IEEE_SET_HALTING_MODE, 203
IEEE_SET_ROUNDING_MODE, 201
IEEE_SET_STATUS, 203
IEEE_SET_UNDERFLOW_MODE, 201
IEEE_SIGNALING_NAN, 196
IEEE_SQRT, 195
IEEE_STATUS_TYPE, 201
IEEE_SUPPORT_DATATYPE, 197, 198
IEEE_SUPPORT_DENORMAL, 197
IEEE_SUPPORT_DIVIDE, 197
IEEE_SUPPORT_INF, 197
IEEE_SUPPORT_IO, 197
IEEE_SUPPORT_NAN, 198
IEEE_SUPPORT_ROUNDING, 198
IEEE_SUPPORT_SQRT, 198
IEEE_SUPPORT_STANDARD, 198
IEEE_SUPPORT_UNDERFLOW, 198
IEEE_SUPPORT_UNDERFLOW_CONTROL, 200, 201
IEEE_TO_ZERO, 197
IEEE_UNDERFLOW, 202
IEEE_UNDERFLOW_FLAG, 196
IEEE_UNORDERED, 200
IEEE_UP, 197
IEEE_USUAL, 202
IEEE_VALUE, 200
IEOR, 169
IF, 84
IMPLICIT, 45
Implicit File Connections, 101
implied DO list, 58, 107
INCLUDE statement, 15
INDEX, 169
initial line, 11
input and output, 97
input validation, 124
INQUIRE, 116
inquiry functions, 197
INT, 169
INT_PTR_KIND, 170
INTEGER, 20
integer constant expression, 77
integer editing, 125
INTEGER*1, 31
INTEGER*2, 31
INTEGER*4, 31
INTEGER*8, 31
interfaces, 69, 70
internal files, 98, 100
INTRINSIC, 46
intrinsic functions, 18, 143
IOLENGTH, 119
IOMSG, 106
IOR, 170
IOSTAT, 106, 211
IOSTATE error messages, 211
IS_IOSTAT_END, 171
IS_IOSTAT_EOR, 171
ISHFT, 170
ISHFTC, 171
ISO C Binding, 206
italicized text, defined, 2
iteration count, 86
keywords, 6
kind, 18
KIND, 171
kind function, 201
kind parameter, 21, 23
L editing, 130
labels, 7
LBOUND, 172
LEADZ, 171
LEN, 172
LEN_TRIM, 172
LGE, 172
LGT, 172
list directed
  editing, 134
  input, 135
  output, 135
LLE, 173
LLT, 173
LOG, 173
LOG10, 173
logical
  assignment statement, 81
  expressions, 79
  IF statement, 84
  operators, 80
LOGICAL, 20, 173
logical editing, 129
LOGICAL*1, 31
LOGICAL*2, 30
LOGICAL*4, 30
looping, 85
MAP statement, 55
MASKL, 173
MASKR, 174
MATMUL, 174
MAX, 174
MAXEXPONENT, 174
MAXLOC, 175
MAXVAL, 175
MERGE, 176
MERGE_BITS, 176

MIN, 176
MINEXPONENT, 176
MINLOC, 176
MINVAL, 177
MOD, 177
modifier keys, 2
modules, 69
MODULO, 177
MOVE_ALLOC, 177
multiple statement lines, 14
MVBITS, 178
NAME, 117
NAMED, 117
NAMELIST, 48
namelist directed
  editing, 136
  input, 136
  output, 138
NEAREST, 178
NEW_LINE, 178
NEWUNIT, 111
NEXTREC, 119
NINT, 178
NML, 102
non-elemental subroutines, 200, 203
NOT, 178
NULL, 39, 179
NULLIFY, 39
NUMBER, 117
numeric bases, 21
  decimal, 21
  hexadecimal, 21
  octal, 21
numeric basis
  binary, 21
numeric storage unit, 25
O editing, 125
octal constants, 21
OPEN statement, 110
OPENED, 117
operator precedence, 80
OPTIONAL, 46, 48
options, manual convention, 2
P editing, 129
PACK, 179
PAD, 104, 111
PARAMETER, 19, 49
parentheses in expressions, 77
PAUSE statement, 92
PENDING, 119
phone number, technical support, 216
POINTER, 49
POINTER, 64
pointer arrays, 64
pointer assignment, 37
pointer assignment operator, 37
POINTER statement, 50
pointers, 37
POPCNT, 179
POPPAR, 179
POS, 105, 119
POSITION, 112, 119
positional editing, 131
precedence, 80
PRECSISON, 179
PRESENT, 179
PRINT, 107
printing, 109
PRIVATE, 50, 51, 70
PRIVATE statement, 50
problems, technical support for, 215, 219
PROCEDURE statement, 51
PRODUCT, 180
PROGRAM statement, 139
PUBIC, 50, 51, 70
PUBLIC statement, 50
Q editing, 134
RADIX, 180
RANDOM_NUMBER, 180
RANDOM_SEED, 181
RANGE, 181
rank, 60
READ, 107
READONLY, 112
real, 22
REAL, 181
real editing, 126
REAL*4, 32
REAL*8, 32
REC, 103
RECL, 112, 119
RECORD statement, 52
records, 97
  endfile, 98
  formatted, 97
  unformatted, 98
recursion, 143
RECURSIVE, 143
REF function, 222
relational expressions, 78
relational operators, 78
REPEAT, 182
repeat factor, 122
RESHAPE, 182
RETURN, 144
REWIND, 114
RRSPACING, 182
runtime error messages, 211
S editing, 130
SAVE statement, 52
scalar variable, 58
SCALE, 182
scale factor, 129
SCAN, 183
SELECT CASE, 89
SELECTED_CHAR_KIND, 183
SELECTED_INT_KIND, 183
SELECTED_REAL_KIND, 183
SEQUENCE statement, 53
SEQUENTIAL, 117
SET_EXPONENT, 184
shape, 60
shared data, 41
SHIFT functions, 223
SHIFTA, 185
SHIFTL, 185

SHIFTR, 185
SIGN, 105, 112, 119, 185
sign control editing, 130
SIN, 185
SINH, 185
SIZE, 119, 186
slash editing, 132
SOURCE, 184
source format, 8
    ANSI standard, 9
    VAX FORTRAN tab-format, 13
SP editing, 130
SPACING, 186
SPREAD, 186
SQRT, 186
square brackets, defined, 2
SS editing, 130
STAT, 211
statement format, 8
statement functions, 142
statement labels, 7
statement line
    comment, 10
    continuation, 11
    END, 10
    initial, 11
statement size
    Fortran 90, 12
statements, 7
statements
    executable, 8
statements
    nonexecutable, 8
statements, 29
STATUS, 112
STDCALL statement, 53
STOP statement, 91
storage, 25
storage association, 26
storage definition, 26
storage sequence, 26
storage unit, 25
    character, 25
    numeric, 25
STORAGE_SIZE, 187
STREAM, 119
STRUCTURE declaration, 53
SUBROUTINE, 140
subroutines, 139
subscript, 60
    expression, 61
substring, 24
    expressions, 24
SUM, 187
symbolic names, 6
    global, 6
    local, 6
SYSTEM_CLOCK, 187
T editing, 132
TAN, 187
TANH, 188
TARGET, 56
technical support, 215, 219
TINY, 188
TL editing, 132
TR editing, 132
TRAILZ, 188
TRANSFER, 188
TRANSPOSE, 188
TRIM, 188
type, 17
TYPE, 109
type statement, 29
    CHARACTER, 34
    COMPLEX, 29
    DOUBLE PRECISION, 29
    INTEGER, 29
    LOGICAL, 29
    REAL, 29
UBOUND, 189
underlined text, defined, 2
UNFORMATTED, 118
unformatted data transfer, 109
unformatted record, 98
UNION statement, 55
UNIT, 100
    preconnected, 100
UNPACK, 189
user defined data types, 35
VAL function, 222
value separator, 134
VALUE statement, 56
variable, 24
variable format expressions, 134
VAX FORTRAN
    tab-format, 13
VAX hexadecimal format, 21
vector subscript, 66
VERIFY, 189
VOLATILE statement, 57
W option, 9
W132 option, 13
WAIT, 116
WHERE statement, 92
WRITE, 107
X. see conditional compilation
X editing, 131
Z editing, 125
ZEXT function, 222