FORTRAN 77 Language Reference

Contents

CHAPTER 1

Introduction

INTRODUCTION TO THIS MANUAL

This is the common reference manual for the Absoft Fortran 77 implementations on both Intel and PowerPC CPUs. Operating systems supported on the platforms include: Windows (95, 98, 2000, and NT), Linux/UNIX, and MacOS. Absoft Fortran 77 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 77

Absoft Fortran 77 is a complete implementation of the 1978 ANSI version of the FORTRAN programming language: FORTRAN 77. 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 excellent networking capabilities at very low prices. As a result, they are quickly replacing mainframes and workstations since they provide better performance at much lower prices. This trend is expected to continue indefinitely.

Absoft Fortran 77 is based on a completely new compiler, designed especially for these modern CPUs. It is not an evolutionary descendent of older compiler technology. It is a workstation class compiler offering superior execution speed and complete integration into modern graphical interface-intensive environments. Absoft Fortran 77 brings a complete software development tool set with exceptional flexibility and improved ease-of-use to modern personal computers.

COMPATIBILITY

Absoft Fortran 77 provides excellent compatibility with code developed on mainframes and workstations. Most popular VAX/VMS statement extensions are accepted, as well as several from IBM/VS, Cray, Sun, and the new Fortran 90 standard. See the chapter **Porting Code** in the *ProFortran User Guide* for additional compatibility information.

NOTE on Fortran 90: While Fortran 90 is the newest standard, most code in use today is actually FORTRAN 77. Since 1978, various compiler vendors have endowed FORTRAN 77 with many extensions to the original standard. Code containing such extensions will not recompile without modification under ANSI F90, because these extensions are not included in the ANSI Fortran 90 standard. For many users, FORTRAN 77 is still the best choice for porting, maintaining, and enhancing legacy FORTRAN code. Absoft F90 is available for users wishing to move to that dialect. Absoft Fortran 77 and Fortran 90 are

fully link compatible, allowing proven legacy FORTRAN 77 routines to be easily combined with new Fortran 90 routines into a single application.

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 SAMPLE
WRITE(9,*) "Hello World!"
END
```

- Absoft extensions to FORTRAN 77 are highlighted in gray in this manual.

CHAPTER 2

The FORTRAN 77 Program

FORTRAN 77 source programs consist of one program unit called the main program and any number of 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 77 compiler during the compilation of a source program. Every program unit is written using the FORTRAN 77 character set and follows a prescribed statement line format. A program unit may be one of the following:

- Main program
- Subroutine subprogram
- Function subprogram
- 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 82 characters:

- All uppercase letters (A - Z)
- All lowercase letters (a - z)
- Decimal digits 0 - 9
- The special characters below:

Character	Description
"	quotation mark
_	underscore
!	exclamation point
\	backslash
<	less than
>	greater than
%	percent

Character	Description
+	plus
-	minus
*	asterisk
/	slash
=	equals
.	decimal point
,	comma
	blank
(	opening parenthesis
)	closing parenthesis
$	dollar sign
'	apostrophe

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 77 entity, such as a variable, array, program unit, or labeled common block. The first character of a symbolic name must be a letter. The blank character is not significant in a symbolic name but may be used as a separator.

The compiler accepts symbolic names of up to thirty-one upper and lower case letters, digits, underscores and dollar signs. Upper and lower case letters are distinct unless one of the compiler case fold options (see the chapter **Using the Compilers** in the *ProFortran User Guide*). Symbolic names of greater than 31 characters are acceptable, but only the first 31 characters are significant to the compiler.

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; and common blocks are 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, 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 77, 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 is:

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

Note that the embedded blanks are not significant 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 some instances it may be impossible for the compiler to determine from the context the meaning the programmer intended. For example:

```
CHARACTER*5 CHAR
CHAR(2:3) = CHAR(64)//CHAR(2:3)
```

Such ambiguous contexts should obviously be avoided.

LABELS

A statement label may appear on a FORTRAN 77 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 and blanks are 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.

The following examples all yield the same label:

```
1101
1 101
11 01
110 1
```

The use of labels has no effect on either the ultimate size of the compiled program and/or its execution speed. However, their inclusion in the source program does increase the memory required for compilation. Labels are used in FORTRAN 77 as their name implies: to label statement lines for reference purposes. Excessive unnecessary labels slow compilation and may even prevent compilation and should therefore be avoided. Labels that are not referenced in your program have no effect on code generation.

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 77 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 7 types of nonexecutable statements:

- Specification statements
- Data initialization statements
- `FORMAT` statements
- Function defining statements
- Subprogram statements
- Main program 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 carriage return, line feed, or carriage return-line feed pair). Numerous FORTRAN statement formats are accepted by using the various compiler options shown in the table below. The "fixed" source formats are actually modern names for the FORTRAN 77 ANSI Standard source format.

Format Name	Compiler Options
FORTRAN 77 (ANSI Standard)	no additional
Fortran 90 Fixed Source Form	no additional
Fortran 90 Free Source Form	**-8**
VAX FORTRAN Fixed-Format	no additional
VAX FORTRAN Fixed-Format (wide)	**-W**
VAX FORTRAN Tab-Format	**-V**
VAX FORTRAN Tab-Format (wide)	**-V -W**
IBM VS FORTRAN Fixed-Form	no additional
IBM VS FORTRAN Free-Form	**-N112**

FORTRAN Source Formats

[Note: The FORTRAN language as described in this manual is the same regardless of the source format chosen.]

FORTRAN 77 ANSI Standard

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

The **-W** 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 -W compiler option)
Identification	73-80	
	132+	(using -W compiler option)

The Identification field is available for any purpose the programmer may desire and is ignored by the FORTRAN 77 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 77:

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 77 compiler.

End Line – the last line of a program unit.

- The word `END` must appear within the statement field.
- Each FORTRAN 77 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 19 continuation lines per statement, for a total of 20 lines per statement. Absoft FORTRAN 77 actually accepts an unlimited number of continuation lines.

Fortran 90 Free Source Form

A Fortran 90 free source form 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* columns:

```
100 I=123
J=456
200I=123
       3      0      0      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.
- The maximum size of a statement is 2640 characters.

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  &
               0       0 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 77 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 can not 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 **-W** compiler option will expand the statement field to column 132.

IBM VS FORTRAN Free-Form

An IBM VS FORTRAN free-form statement line consists of 80 character positions or columns. Although labels and statements may appear in any columns, comments must begin in column 1. All characters not in character context are folded to lower case as if the **-f** compiler option is on.

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

```
100 I=123
J=456
200I=123
       3      0      0      I=123
400
     I=123
```

Comments in IBM VS FORTRAN free-form begin with a quotation mark (") in column 1. The letter C in column 1 does *not* indicate a comment and blank lines are not permitted in this source format. The following example has one comment line:

```
"This line is a comment
A=1
C="abcdef"
```

To continue a statement across multiple lines, the minus sign character (-) must appear as the last character of a continued line. If the last two characters of a line are minus signs, the last one is treated as a continuation character while the other is treated as a minus sign. A comment may not be continued. There is a limit of 19 continued lines (20 lines total), however, the total number of characters permitted for a single statement is only 1320. Absoft FORTRAN 77 actually accepts an unlimited number of continuation lines. The following is an example of IBM VS FORTRAN free-form continuation:

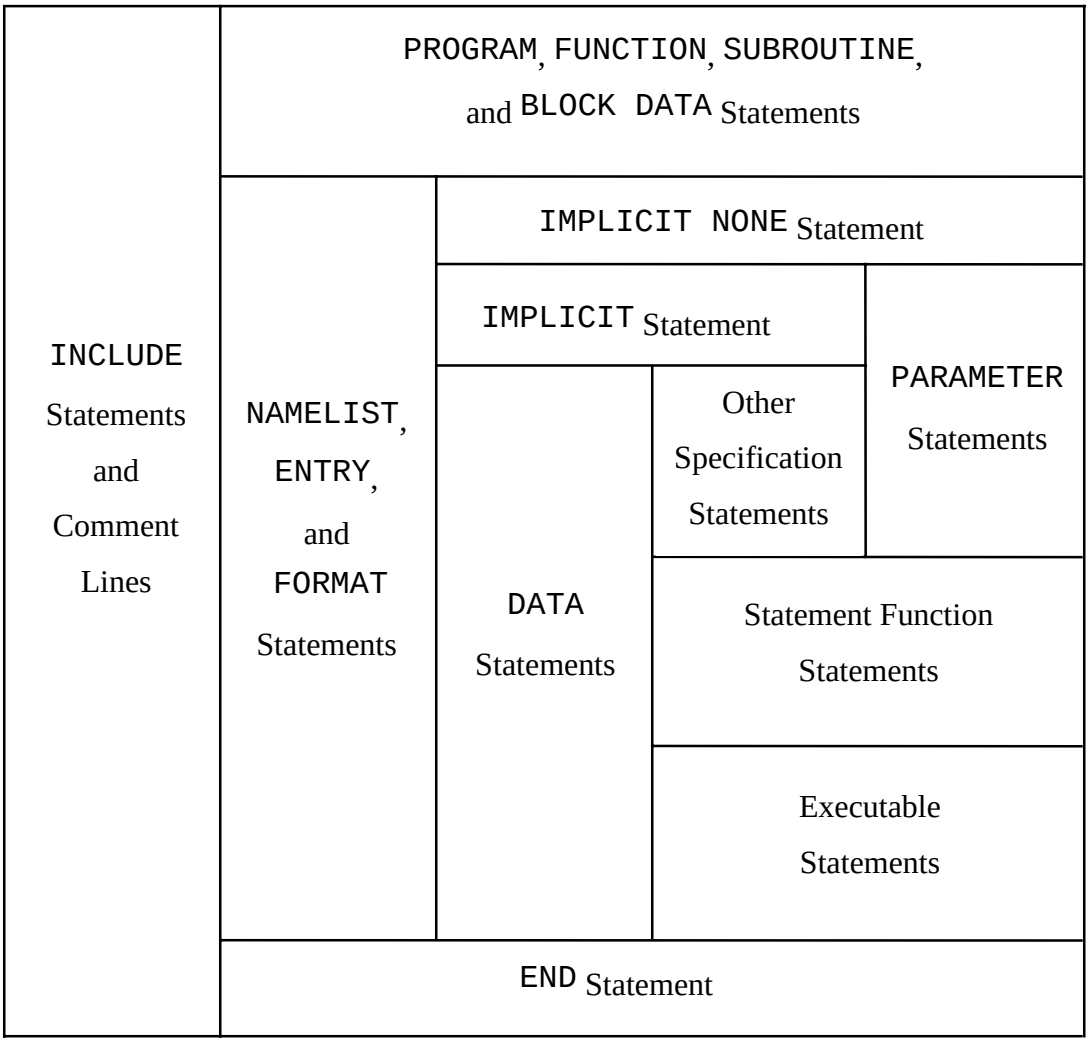
```
"This comment cannot be continued
WRITE(*,*)"-
This string contains -
no minus signs"
WRITE(*,*) "This string contains --
one minus sign"
```

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



Required Statement Order

INCLUDE Statement

This statement is a compiler directive and is provided as a convenience for copying standard declaration statements, subroutine libraries, 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 *ProFortran User Guide*, is useful for specifying directories and search paths for include files which may reside in directories other than the current one.

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

$PACKON
$PACKOFF

$IF expr
$ELSE
$ELSEIF expr
$ENDIF
```

`$PACKON` is used to turn storage compression on for `STRUCTURE` and `UNION` definitions. `$PACKOFF` is used to turn off storage compression. The default for Absoft Fortran 77 depends on the host machine architecture and operating system and is described in the *Fortran User Guide*.

name is the 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. These compiler directives operate in a similar manner to the block `IF` constructs of FORTRAN 77 (described in the **Control Statements** chapter). 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 .eq. 4
            REAL a,b,c,d(100),pi
            pi = atan(1.0)*4.0
      $ELSEIF precision .eq. 8
            DOUBLE PRECISION a,b,c,d(100),pi
            pi = datan(1d0)*4d0
      $ENDIF
      $ENDIF
```

Note the first `$ENDIF` which is required to terminate the `$ELSEIF` clause.

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.

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 .not. DEFINED(TYPES)
            INCLUDE "types.inc"
      $ENDIF
```

your source program file only needs to include "graphics.inc" to compile successfully. This strategy works best when used in a `GLOBAL DEFINE` subprogram (described in the **Programs, Subroutines, and Functions** chapter), because although the conditional compilation variables will have a scope of the entire file, the declarations made in the include file will have a scope of only the current program unit.

DATA ITEMS

The symbolic name used to represent a constant, variable, array, substring, statement function, 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 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, and block data subprogram are all identified with symbolic names, but have no data type.

Constants

FORTRAN 77 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.

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
```

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 characters 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'
```

FORTRAN 77 has no facility for specifying or representing a character constant consisting of the null string. However, to facilitate linking with the C language, the compiler will interpret the character constant '' or "" as a single zero byte. A null terminated C style string can then be created by concatenating the character constant '' or "" on to the end of a FORTRAN string.

As an extension to FORTRAN 77, 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 **-K** 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 **-K** option, it displays,

```
First line\nSecond line
```

When compiled with the **-K** 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 in the range of -2147483648 to +2147483647 with negative integers maintained in two's complement form. An integer constant is a string of decimal digits that may contain a leading sign. An integer constant requires one numeric storage unit (four bytes).

```
15
101
-72
1126
123 456 789
```

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'111100111100111001111'
J = O'17171717'
K = Z'3CF3CF'
```

As with all numeric constants, spaces may be used freely to enhance legibility. The following examples produce identical assignment statements:

```
I = B'0011 1100 1111 0011 1100 1111'
J = O'017 171 717'
K = Z'3C F3 CF'
```

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 which 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 and an integer constant in the range of a -37 to +39. If an exponent is given and the fractional part is zero, the decimal point may be omitted. A real constant requires one numeric storage unit (four bytes).

Constant	Value
1E2	= 100.0
-12.76	= -12.76
1.07E-1	= .107
0.4237E3	= 423.7

Real values are maintained in IEEE single precision floating point representation. 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 Constant

A double precision constant is formed in the same manner as a real constant except that the exponent is designated with the letter D and must always be given, even if its value is zero. The exponent range of a double precision constant is -307 to +309. A double precision constant requires two numeric storage units (eight bytes).

Constant	Value
1D2	= 100.0
-12.76D0	= -12.76
1.07D-1	= .107
0.4237D3	= 423.7

Double precision values are maintained in IEEE double precision floating point representation. 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.

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*

Complex*16 Constant

A double complex constant is stated in the same format as a single precision complex constant, except that double precision constants must be used for the real and imaginary portions. A double complex constant requires four numeric storage units (sixteen bytes).

Constant	Value
(0.62D2,-0.22D-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.

Variables

A variable is used to maintain a FORTRAN 77 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 (see below). 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.

Arrays

An array is 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.

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 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 size and shape of an array which consists of the number of dimensions, the upper and lower bounds of each dimension, and the number of array elements. The number of dimensions is determined by the number of dimension declarators. Dimension bounds specify the size or extent of an individual dimension. 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 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, or assumed size 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 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. Adjustable and assumed size array declarators do not supply sufficient information to map the static memory at compile time.

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 numeric expressions whose values fall between the lower and upper bounds of the corresponding dimension. If the value of the expression is not an integer, the compiler supplies the appropriate conversion. There must be a subscript expression for each declared dimension.

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.

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.

The order of an array element within the column major storage sequence of the array in memory is called the subscript value. 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
.	.	.	.
.	.	.	.
.	.	.	.
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 Name

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. Array names may be used in this manner 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.

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.

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.

As with arrays, 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. If a substring expression is not integer, automatic conversion to integer is supplied by the compiler.

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 which 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 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 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
Complex*16	4 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 77 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 SAVE statements.
- in named COMMON blocks.

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 *ProFortran User Guide*).

CHAPTER 3

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 77: 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.

An exception to this order occurs for operations between `COMPLEX` values and `DOUBLE PRECISION`. In this instance, results are returned as `COMPLEX*16`.

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 .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 coersion 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	same as .EQV.
.XOR.	nonequivalence	same as .EQV.

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

The integer intrinsic function SHIFT is available to perform left and right logical shifts (see the chapter **Programs, Subroutines, and Functions**).

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.

MEMORY ASSIGNMENT STATEMENT

Memory assignment statements are used to store values in absolute memory addresses:

```
ma = e
```

where: *ma* is an absolute memory address

e is an any arithmetic, logical, or character expression.

A memory address is formed as follows:

```
BYTE (e)   byte (8 bit) reference
WORD (e)   word (16 bit) reference
LONG (e)   long (32 bit) reference
```

where: *e* is an absolute memory address.

For example:

```
BYTE (Z'FFFFE0') = 10
```

stores the decimal value 10 at the hexadecimal memory byte address FFFE0.

The BYTE, WORD, and LONG keywords also represent intrinsic functions allowing indirect addressing:

```
WORD (WORD(O'4000')) = Z'FFFF'
```

stores the sixteen bit hexadecimal value FFFF at the absolute memory location whose address is the address contained at the octal address 4000.

CHAPTER 4

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 nonexecutable.

DATA statements are used to establish initial values for the variables and arrays used within a FORTRAN 77 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/]]...
```

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.

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 *ProFortran 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 *ProFortran 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 *ProFortran User Guide*).

- REAL*8 data is identical to DOUBLE PRECISION and occupies two 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 *ProFortran 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.

Character Type Statement

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

```
CHARACTER [*len [,]] v[*len] [,v[*len]]...
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')
```

produces a ten byte symbolic `CHARACTER` constant.

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 **The FORTRAN 77 Program**.

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)
COMMON A(5,5,5)
```

The VIRTUAL statement has the same effect as the DIMENSION statement in order to be compatible with other implementations of FORTRAN. Absoft Fortran 77, however, has no means of declaring virtual storage.

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.

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:

```
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|
|---------------A---------------|
        |---------------B---------------|
|---------I---------|
            |-------------J-------------|
```

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:

```
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11|
|-------------------A-------------------|
            |---------------B---------------|
```

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.

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 77 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.

INLINE STATEMENT

As an extension to standard FORTRAN, Absoft Fortran 77 supports the INLINE statement to allow programmers to insert object code directly into a FORTRAN program. This is useful for customizing the language to the host system. The syntax of the INLINE statement is as follows:

```
INLINE ([identifier1=con1[,identifier2=/con1[,con2.../]]])
…
CALL identifier1[(arg1[,arg2...])]
variable = identifier2[(arg1[,arg2...])]
```

An INLINE declaration can have the same format as a PARAMETER statement or it can substitute a list of constants for an identifier instead of a single constant. All constants must be of type INTEGER. The identifier can then be referenced as if it were a subroutine or function. Instead of generating a call to an external function, the compiler will insert the constant or constant list directly into the object code. Each constant is output as a 32-bit opcode. If an argument list is given, the actual arguments will be passed in the same fashion as they are passed to an external routine. If the identifier is referenced as a function, the result must be returned using the standard call-return methods.

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.

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')
```

Special use of the PARAMETER statement

As a means of defining CHARACTER symbolic names with non-printing ASCII characters, a character symbolic name may be defined with an INTEGER constant in the range of 1-255:

```
CHARACTER EOL,EOF
PARAMETER (EOL=10,EOF=O'377')
```

POINTER STATEMENT

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.

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 77 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.

AUTOMATIC STATEMENT

FORTRAN 77 permits dynamic as well as static storage allocation. Variables and arrays which are declared locally in subprograms and not associated in a COMMON block 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. A compile time option (see the chapter **Using the Compilers** in the *ProFortran User Guide*) is provided to change this behavior and force the memory for all variables to be

allocated statically. When this option is specified, the AUTOMATIC statement may be used to override static allocation:

```
AUTOMATIC [a [,a]...]
```

where: *a* is either 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.

Note that local variables (variables not associated in a COMMON block) are allocated dynamically by default in Absoft Fortran 77. This statement is provided for compatibility with other compilers and to override the static memory allocation compiler option.

STRUCTURE DECLARATION

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 compatability 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 extablished 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.

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 5

Control Statements

Control statements direct the flow of execution in a FORTRAN 77 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 (e) THEN
      block of statements
ELSE
      block of statements
END IF
```

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.

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 77. The standard DO loop and Absoft Fortran 77 extensions to the DO loop are 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

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*, 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.

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.

Transfer into Range of DO Loop

Under certain conditions, FORTRAN 66 permitted transfer of control into the range of a DO loop from outside the range. This was known as the “extended range of a DO”. Such a transfer is considered highly unstructured and is prohibited in ANSI FORTRAN 77. However, in Absoft Fortran 77, all DO loops may be considered extended range, although it is not good programming practice.

DO WHILE

The DO WHILE statement is an extension to standard FORTRAN 77 and provides a method of looping not necessarily governed by an iteration count. The form of the DO WHILE statement is:

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

where: *s* is the statement label of an executable statement that defines the range of the loop. The statement identified by *s* must follow the DO statement in the sequence of statements within the same program unit as the DO statement. If the label *s* is omitted, the loop must be terminated with a REPEAT or END DO statement.

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,eof; PARAMETER (eof=-1)

     DATA a,b,c /3*0.0/

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

Block DO

The block DO extension to standard FORTRAN 77 provides four additional methods for constructing a loop. They are as follows:

1.
```
DO
      block
END DO
```

2.
```
DO ( i=e1, e2 [,e3])          DO i=e1, e2 [,e3]
      block            or           block
END DO                        END DO
```

3.
```
DO (e4 TIMES)
      block
END DO
```

4.
```
DO (e4) TIMES
      block
END DO
```

All four forms of block DO require a REPEAT or END DO statement to terminate the loop. An EXIT or LEAVE statement (described below) may be used to abnormally exit the loop and a CYCLE statement (also described below) may be used to force iteration of the loop.

The first case 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).

The second case is identical to the standard DO loop without a terminating statement label. The value *i* is the DO variable, *e1* is its initial value, *e2* is its terminating value and *e3*, if present, is the increment value.

The value *e4*, in the third case, is the iteration count and may be an integer, real, or double precision expression. Where the value *e4* is not an integer, it is first converted to an integer and the truncated value becomes the iteration count. At least one blank character must be present between the iteration count expression and the keyword TIMES.

END DO and REPEAT

The END DO and REPEAT statements are extensions to standard FORTRAN 77 and are used to terminate DO WHILE loops and block DO structures. Each block DO must have a matching END DO or REPEAT statement. After execution of an END DO or REPEAT statement, the next statement executed depends on the result of the DO loop incrementation processing. The form of the END DO and REPEAT statements is:

```
END DO (or REPEAT)
```

EXIT and LEAVE statements

The EXIT and LEAVE statements are also extensions to standard FORTRAN 77 and provides a convenient means for abnormal termination of a DO loop. These statements 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 is an extension to FORTRAN 77 and 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
```

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:

```
SELECT CASE (e)
      CASE (case_selector)
            [block]
      [CASE (case_selector)
            [block]
      ...]
      [CASE DEFAULT]
            [block]
END SELECT
```

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.

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 match is found, a run-time error is reported.

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
*

IMPLICIT INTEGER(a-z)
CHARACTER line*80
PARAMETER (eof=-1)

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

  DO
    READ (*,'(a)',IOSTAT=ios) line
    IF (ios==eof) 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 must have an END statement which terminates the range of individual program units 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 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 with no other characters:

```
END
```

CHAPTER 6

Input/Output and FORMAT Specification

Input and output statements provide a channel through which FORTRAN 77 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 77 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 77 programs. For example, a text processor might be used to create and edit a document file and a FORTRAN 77 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 77 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 77 contains provisions for both of these requirements. The default manner of unformatted processing of a sequential access device is to treat it as a pure byte stream. Partial record reads and backspacing are not possible. The data transmitted is exactly what your `WRITE` statement specifies or what the external media contains. There is no limit on the length of a record.

When partial record reads and backspacing of unformatted sequential files are required, two methods may be employed:

1. On a file-by-file basis, the runtime system can be informed by adding the "`BLOCK=-1`" specifier to the `OPEN` statement. The `BLOCK` specifier is an extension normally used to specify the blocking factor applied to magnetic tape. When a file is opened for unformatted sequential access and this specifier is negatively valued, each record written will be preceded and followed by four bytes containing the length of the record.

2. Compile your program with the **-N3** compiler option. This causes all `OPEN` statements for sequential unformatted files to have embedded record information (i.e. as though "`BLOCK=-1`" had been specified for each `OPEN`).

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.

File Buffering

The I/O library uses double buffering for all external files. One buffer, referred to below as the transfer buffer, is used to hold the data transferred during the input or output of a single logical record. Since the data held in the transfer buffer is only used during the input or output of a single record, the same buffer is used for all connected files. In addition to the transfer buffer, a second buffer is associated with each connected file. This buffer, referred to below as the physical buffer, is used to hold multiple logical records before writing them to disk. By default, the physical buffer is 1024 bytes for sequential access files. For direct access files, the default physical buffer is either 1024 bytes or the logical record size, whichever is larger.

The physical buffer size can be adjusted by using the BUFFER= specifier in the OPEN statement (see below).

I/O SPECIFIERS

FORTRAN 77 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. Units 5, 6, and 9 are preconnected to "standard input", usually the keyboard, for input operations and "standard output", usually the screen, 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.

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.

To generate an end of file from the keyboard, type either Command-Return or Command-Enter.

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 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.

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:

IOSTAT	=	an I/O status specifier as described above.
ERR	=	an error specifier as described above.
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.
NAME	=	NAME= is a synonym for FILE= in the OPEN statement.

STATUS	=	a CHARACTER expression which must be OLD, NEW, SCRATCH, 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.
ACCESS	=	a CHARACTER expression which must be SEQUENTIAL, DIRECT, or APPEND and specifies the access mode or position. The default value is SEQUENTIAL.
ORGANIZATION	=	ORGANIZATION= is a synonym for ACCESS=.
FORM	=	a CHARACTER expression which must be FORMATTED or UNFORMATTED specifying the type of records in the file. The default value is UNFORMATTED for direct access files and FORMATTED for sequential access files.
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. The **-N51** option may be used for the length to be specified in 4-byte units.
RECORDSIZE	=	RECORDSIZE= is a synonym for RECL=.
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.
MAXREC	=	an INTEGER expression specifying the maximum number of records permitted with direct access files.
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.
ACTION	=	a CHARACTER expression which must be READ, WRITE, BOTH, 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.
READONLY		a specifier **without** an equal sign equivalent to ACTION=READ.
BUFFER	=	an integer expression which specifies the physical size (in bytes) of the I/O buffer. A value of zero is useful in that no buffering is used.
DISPOSE	=	a CHARACTER expression which must be KEEP, SAVE, DELETE, PRINT, PRINT/DELETE, SUBMIT or SUBMIT/DELETE. When set to KEEP or SAVE, the file is retained after closing. When set to DELETE or SUBMIT/DELETE, the file is not retained after closing. When set to PRINT/DELETE, the first character in each record is interpreted as carriage control and the file is not retained after closing. SUBMIT has no effect and is provided for compatibility only.
DISP	=	DISP= is a synonym for DISPOSE=.
BLOCK	=	an INTEGER expression. When the expression is negative (i.e -1), each record written will be preceded and followed by four bytes containing the length of the record. This specifier is to be used only with files opened for unformatted sequential access. See the **File Access** section near the beginning of this chapter for more information about BLOCK.
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.
SHARED		a specifier **without** an equal sign which has no effect, and is supplied for compatibility only.
NOSPANBLOCKS		a specifier **without** an equal sign which has no effect, and is supplied for compatibility only.
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.

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 77 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.

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.

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.

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)
```

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. Only one of UNIT= or FILE= may be specified. One or more of the following *ilist* specifiers are also used with the INQUIRE statement:

IOSTAT = an I/O status specifier as described above.

ERR = an error specifier as described above.

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.

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.

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.

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.

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.

BLANK = a CHARACTER variable or array element which is defined with either the value NULL or ZERO depending on how blanks are handled.

SIZE = an INTEGER variable or array element which is defined with the size of the file in bytes.

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:

I*w* and I*w*.*m*	integer editing
F*w*.*d*	floating point editing
E*w*.*d* and E*w*.*d*E*e*	single precision scientific editing
D*w*.*d*	double precision scientific editing
G*w*.*d* and G*w*.*d*E*e*	general floating point editing
L*w*	logical editing
A[*w*]	character editing
B*w* and B*w*.*m*	binary editing
O*w* and O*w*.*m*	octal editing
Z*w* and Z*w*.*m*	hexadecimal 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
*n*H*h1 h2 … hn*	Hollerith string
*n*X	skip positions
T*c*, TL*c*, and TR*c*	tab to column
*k*P	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 `Iw` and `Iw.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 *k*P edit descriptor.

Edit Descriptor	Absolute value of Exponent	Form of Exponent
E*w*.*d*	≤ 99	E±*nn*
E*w*.*d*	100 - 999	±*nnn*
E*w*.*d*E*e*	≤ $(10^{e})-1$	E±*n1n2…ne*
D*w*.*d*	≤ 99	D±*nn*
D*w*.*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
```

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 scale factor 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 $-d < k < d + 2$.

```
      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 L*w* 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.

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 77 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 77 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 desciptor to determine the number of characters in a record and then reads that many characters into the array CHRS.

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 `F`*w*`.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 I*w* 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 77, 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 isassumed 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 7

Programs, Subroutines, and Functions

There are seven types of procedures available in Absoft Fortran 77: main programs, subroutines, external functions, statement functions, intrinsic functions, BLOCK DATA, and GLOBAL DEFINE.

The main program is the entry point of a FORTRAN 77 program. The compiler does not require that the main program occurs first in the source file, however, every FORTRAN 77 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 77 subprograms or by means other than FORTRAN 77 such as assembly language or the C programming language.

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 77 program.

GLOBAL DEFINE subprograms are nonexecutable program units which allow for declarations which define no storage and are visible to an entire FORTRAN source file. Such declarations are STRUCTURE, PARAMETER, EXTERNAL and INLINE.

PROGRAMS

The PROGRAM statement is given in the following manner:

```
PROGRAM pgm
```

The program statement is not required to be present in a FORTRAN 77 program. If it is present it must be the first line of the main program unit.

SUBROUTINES

A subroutine is a separate 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.

While the ANSI standard prohibits a subroutine from referencing itself, directly or indirectly, this implementation of FORTRAN 77 allows recursion.

The form of a subroutine subprogram declaration is:

```
SUBROUTINE sub [([arg] [,arg]...)]
```

where: *sub* is a unique symbolic name that is used to reference the subroutine.

([*arg*] [,*arg*]…) is an optional 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 which has the form:

```
CALL sub [([arg] [,arg]...)]
```

where: *sub* is the symbolic name of a subroutine or dummy procedure.

([*arg*] [,*arg*]...) is the list of actual arguments which 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 77 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 which 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.

Absoft FORTRAN 77 allows the recursive use of external functions.

The form of a function subprogram declaration is:

```
[type [*len]] FUNCTION func ([arg] [,arg]...)
```

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.

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 77 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 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.

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 table in the **Programs, Subroutines, and Functions** chapter.

The generic names listed in that table are provided to simplify the use of intrinsic functions that take different types of arguments. Except for the type conversion functions, the type of a generic function is the same as the type of its arguments.

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.

Table
Intrinsic Functions

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
			Type Conversion		
INT	INT	INT(x)	any	integer	1
IINT	INT	IINT(x)	real	integer*2	1
JINT	INT	JINT(x)	real	integer	1
KINT	INT	KINT(x)	real	integer*8	1
IFIX	INT	IFIX(x)	real	integer	1
IIFIX	INT	IIFIX(x)	real	integer*2	1
HFIX	INT	HFIX(x)	real	integer*2	1
KFIX	INT	KFIX(x)	real	integer*8	1
JIFIX	INT	JIFIX(x)	real	integer	1
IDINT	INT	IDINT(d)	double	integer	1
IIDINT	INT	IIDINT(d)	double	integer*2	1
JIDINT	INT	JIDINT(d)	double	integer	1
REAL	REAL	REAL(x)	any	real	2
FLOAT	REAL	FLOAT(i)	integer	real	2
FLOATI	REAL	FLOATI(i)	integer*2	real	2
FLOATJ	REAL	FLOATJ(i)	integer	real	2
FLOATK	REAL	FLOATJ(i)	integer*8	real	2
SNGL	REAL	SNGL(d)	double	real	2
DBLE	DBLE	DBLE(x)	any	double	3
DREAL	DREAL	DREAL(x)	any	double	3
DFLOAT	DBLE	DFLOAT(x)	any	double	3
DFLOTI	DBLE	DFLOTI(i)	integer*2	double	3
DFLOTJ	DBLE	DFLOTJ(i)	integer	double	3
DFLOTK	DBLE	DFLOTJ(i)	integer*8	double	3
CMPLX	CMPLX	CMPLX(x)	any	complex	4
DCMPLX	DCMPLX	DCMPLX(x)	any	complex*16	4
ICHAR		ICHAR(a)	character	integer	5
CHAR		CHAR(i)	integer	character	5
			Truncation		
AINT	AINT	AINT(r)	real	real	1
DINT	AINT	DINT(d)	double	double	1
			Nearest Whole Number		
ANINT	ANINT	ANINT(r)	real	real	
DNINT	ANINT	DNINT(d)	double	double	
			Nearest Integer		
NINT	NINT	NINT(r)	real	integer	
ININT	NINT	ININT(r)	real	integer*2	
JNINT	NINT	JNINT(r)	real	integer	
KNINT	NINT	JNINT(r)	real	integer*8	
IDNINT	NINT	IDNINT(d)	double	integer	

IIDNNT	NINT	IIDNNT(d)	double	integer*2
JIDNNT	NINT	JIDNNT(d)	double	integer
KIDNNT	NINT	KDNINT(d)	double	integer*8

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Absolute Value			
ABS	ABS	ABS(x)	any	any	6
IABS	ABS	IABS(i)	integer	integer	
IIABS	ABS	IIABS(i)	integer*2	integer*2	
JIABS	ABS	JIABS(i)	integer	integer	
KIABS	ABS	KIABS(i)	integer*8	integer*8	
DABS	ABS	DABS(d)	double	double	
CABS	ABS	CABS(c)	complex	real	6
CDABS	ABS	CDABS(cd)	complex*16	double	6
		Remaindering			
MOD	MOD	MOD(x,y)	any	any	
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	
AMOD	MOD	AMOD(r,s)	real	real	
DMOD	MOD	DMOD(d,e)	double	double	
		Transfer of Sign			
ISIGN	SIGN	ISIGN(i,j)	integer	integer	
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	
SIGN	SIGN	SIGN(r,s)	real	real	
DSIGN	SIGN	DSIGN(d,e)	double	double	
		Positive Difference			
IDIM	DIM	IDIM(i,j)	integer	integer	
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	
DIM	DIM	DIM(r,s)	real	real	
DDIM	DIM	DDIM(d,e)	double	double	
		Double Precision Product			
DPROD		DPROD(r,s)	real	double	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Choosing Largest Value			
MAX	MAX	MAX(x,y,…)	any	any	
MAX0	MAX	MAX0(i,j,…)	integer	integer	
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	
AMAX1	MAX	AMAX1(r,s,…)	real	real	
DMAX1	MAX	DMAX1(d,e,…)	double	double	
AMAX0		AMAX0(i,j,…)	integer	real	
AIMAX0		AIMAX0(i,j,…)	integer*2	real	
AJMAX0		AIMAX0(i,j,…)	integer	real	
MAX1		MAX1(r,s,…)	real	integer	
IMAX1		IMAX1(r,s,…)	real	integer*2	
JMAX1		JMAX1(r,s,…)	real	integer	
KMAX1		KMAX1(r,s,…)	real	integer*8	
		Choosing Smallest Value			
MIN	MIN	MIN(x,y,…)	any	any	
MIN0	MIN	MIN0(i,j,…)	integer	integer	
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	
AMIN1	MIN	AMIN1(r,s,…)	real	real	
DMIN1	MIN	DMIN1(d,e,…)	double	double	
AMIN0		AMIN0(i,j,…)	integer	real	
AIMIN0		AIMIN0(i,j,…)	integer*2	real	
AJMIN0		AIMIN0(i,j,…)	integer	real	
MIN1		MIN1(r,s,…)	real	integer	
IMIN1		IMIN1(r,s,…)	real	integer*2	
JMIN1		JMIN1(r,s,…)	real	integer	
KMIN1		KMIN1(r,s,…)	real	integer*8	
		Imaginary Part of Complex			
AIMAG		AIMAG(c)	complex	real	6
DIMAG		DIMAG(cd)	complex*16	double	6
		Conjugate of Complex			
CONJG		CONJG(c)	complex	complex	6
DCONJG		DCONJG(cd)	complex*16	complex*16	6
		Square Root			
SQRT	SQRT	SQRT(r)	real	real	
DSQRT	SQRT	DSQRT(d)	double	double	
CSQRT	SQRT	CSQRT(c)	complex	complex	
CDSQRT	SQRT	CDSQRT(cd)	complex*16	complex*16	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Exponential			
EXP	EXP	EXP(r)	real	real	
DEXP	EXP	DEXP(d)	double	double	
CEXP	EXP	CEXP(c)	complex	complex	
CDEXP	EXP	CDEXP(cd)	complex*16	complex*16	
		Natural Logarithm			
LOG	LOG	LOG(x)	any	any	
ALOG	LOG	ALOG(r)	real	real	
DLOG	LOG	DLOG(d)	double	double	
CLOG	LOG	CLOG(c)	complex	complex	
CDLOG	LOG	CDLOG(cd)	complex*16	complex*16	
		Common Logarithm			
LOG10	LOG10	LOG10(x)	any	any	
ALOG10	LOG10	ALOG10(r)	real	real	
DLOG10	LOG10	DLOG10(d)	double	double	
		Sine			
SIN	SIN	SIN(r)	real	real	7
SIND	SIND	SIND(r)	real	real	7
DSIN	SIN	DSIN(d)	double	double	7
DSIND	SIND	DSIND(d)	double	double	7
CSIN	SIN	CSIN(c)	complex	complex	7
CDSIN	SIN	CDSIN(cd)	complex*16	complex*16	7
		Cosine			
COS	COS	COS(r)	real	real	7
COSD	COSD	COSD(r)	real	real	7
DCOS	COS	DCOS(d)	double	double	7
DCOSD	COSD	DCOSD(d)	double	double	7
CCOS	COS	CCOS(c)	complex	complex	7
CDCOS	COS	CDCOS(cd)	complex*16	complex*16	7
		Tangent			
TAN	TAN	TAN(r)	real	real	7
TAND	TAND	TAND(r)	real	real	7
DTAN	TAN	DTAN(d)	double	double	7
DTAND	TAND	DTAND(d)	double	double	7
		Arcsine			
ASIN	ASIN	ASIN(r)	real	real	
ASIND	ASIND	ASIND(r)	real	real	
DASIN	ASIN	DASIN(d)	double	double	

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
DASIND	ASIND	DASIND(d)	double	double	

Arccosine

ACOS	ACOS	ACOS(r)	real	real	
ACOSD	ACOSD	ACOSD(r)	real	real	
DACOS	ACOS	DACOS(d)	double	double	
DACOSD	DACOSD	DACOSD(d)	double	double	

Arctangent

ATAN	ATAN	ATAN(r)	real	real	
ATAND	ATAND	ATAND(r)	real	real	
DATAN	ATAN	DATAN(d)	double	double	
DATAND	ATNAD	DATAND(d)	double	double	
ATAN2	ATAN2	ATAN2(r,s)	real	real	
DATAN2	ATAN2	DATAN2(d,e)	double	double	
ATAN2D	ATAN2D	ATAN2D(r,s)	real	real	
DATAN2D	ATAN2D	DATAN2D(d,e)	double	double	

Hyperbolic Sine

SINH	SINH	SINH(r)	real	real	
DSINH	SINH	DSINH(d)	double	double	

Hyperbolic Cosine

COSH	COSH	COSH(r)	real	real	
DCOSH	COSH	DCOSH(d)	double	double	

Hyperbolic Tangent

TANH	TANH	TANH(r)	real	real	
DTANH	TANH	DTANH(d)	double	double	

Length of String

LEN		LEN(a)	character	integer	9
LEN_TRIM		LEN_TRIM(a)	character	integer	26

Location of Substring

INDEX		INDEX(a,b)	character	integer	8

Trim Trailing Blanks

TRIM		TRIM(a)	character	character	11

String Replication and Justification

REPEAT	REPEAT(a,i)	character	character	12
ADJUSTL	ADJUSTL(a)	character	character	13
ADJUSTR	ADJUSTR(a)	character	character	14

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Lexical Comparisons			
LGE		LGE(a,b)	character	logical	10
LGT		LGT(a,b)	character	logical	10
LLT		LLT(a,b)	character	logical	10
LLE		LLE(a,b)	character	logical	10
		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	
		Memory Addressing			
BYTE		BYTE(i)	integer	integer*1	20
WORD		WORD(i)	integer	integer*2	20
LONG		LONG(i)	integer	integer*4	20
[%]LOC		[%]LOC(a)	any	integer*4	20
		Pass By Value			
[%]VAL		[%]VAL(a)	any	any	22
[%]VAL4		[%]VAL4(a)	any	any	22
[%]VAL2		[%]VAL2(a)	any integer	integer*2	22
[%]VAL1		[%]VAL1(a)	any integer	integer*1	22
		Pass By Reference			
[%]REF		[%]REF(a)	any	any	24
		Pass By Descriptor			
[%]DESCR		[%]DESCR(a)	any	any	25
		Bit Move Subroutine			
MVBITS		CALL MVBITS(i,j,l,m,n)			19
		Get Size of A Data Type			
SIZEOF		SIZEOF(type)		integer	23

Specific Name	Generic Name	Usage	Argument Type	Result Type	Notes
		Bitwise Operations			
SHIFT		SHIFT(i,j)	integer	integer	21
ISHFT		ISHFT(i,j)	integer	integer	21
IISHFT		IISHFT(i,j)	integer*2	integer*2	21
JISHFT		JISHFT(i,j)	integer	integer	21
KISHFT		KISHFT(i,j)	integer*8	integer*8	21
ISHFTC		ISHFTC(i,j,k)	integer	integer	16
IISHFTC		IISHFTC(i,j,k)	integer*2	integer*2	16
JISHFTC		JISHFTC(i,j,k)	integer	integer	16
KISHFTC		KISHFTC(i,j,k)	integer*8	integer*8	16
IOR		IOR(i,j)	integer	integer	15
IIOR		IIOR(i,j)	integer*2	integer*2	15
JIOR		JIOR(i,j)	integer	integer	15
KIOR		KIOR(i,j)	integer*8	integer*8	15
IAND		IAND(i,j)	integer	integer	15
IIAND		IIAND(i,j)	integer*2	integer*2	15
JIAND		JIAND(i,j)	integer	integer	15
KIAND		KIAND(i,j)	integer*8	integer*8	15
NOT		NOT(i)	integer	integer	15
INOT		INOT(i)	integer*2	integer*2	15
JNOT		JNOT(i)	integer	integer	15
KNOT		KNOT(i)	integer*8	integer*8	15
IEOR		IEOR(i,j)	integer	integer	15
IIEOR		IIEOR(i,j)	integer*2	integer*2	15
JIEOR		JIEOR(i,j)	integer	integer	15
KIEOR		KIEOR(i,j)	integer*8	integer*8	15
IBITS		IBITS(i,j,k)	integer	integer	17
IIBITS		IIBITS(i,j,k)	integer*2	integer*2	17
JIBITS		JIBITS(i,j,k)	integer	integer	17
KIBITS		KIBITS(i,j,k)	integer*8	integer*8	17
BTEST		BTEST(i,j)	integer	logical	18
BITEST		BITEST(i,j)	integer*2	logical*2	18
BJTEST		BJTEST(i,j)	integer	logical	18
IBSET		IBSET(i,j)	integer	integer	18
IIBSET		IIBSET(i,j)	integer*2	integer*2	18
JIBSET		JIBSET(i,j)	integer	integer	18
KIBSET		KIBSET(i,j)	integer*8	integer*8	18
IBCLR		IBCLR(i,j)	integer	integer	18
IIBCLR		IIBCLR(i,j)	integer*2	integer*2	18
JIBCLR		JIBCLR(i,j)	integer	integer	18
KIBCLR		KIBCLR(i,j)	integer*8	integer*8	18

INTRINSIC FUNCTIONS NOTES

Intrinsic functions, sometimes referred to as mathematics library functions, are presented in the preceding table. This table presents all of the intrinsic functions, their definitions, number of arguments required, types of arguments and function, and the generic and specific names of each function. The following are notes referenced in the table:

1. If *a* is REAL, there are two cases: if |*a* |<1, then INT(*a*)=0; if |*a* |>1, then INT(*a*) is an integer which is rounded toward zero and has the same sign as *a*. If *a* is COMPLEX then the real part of the argument is returned.

2. The function REAL(*a*) will return as much precision as can be specified in a REAL*4 variable. If *a* is COMPLEX then the real portion is returned. If the argument is integer then the function FLOAT will return the same result.

3. This function will return a DOUBLE PRECISION result that contains all the precision of the argument passed. If the argument is of type complex then the real portion is used.

4. CMPLX may have one or two arguments. If there is one and the type is COMPLEX then the argument is returned unmodified. If there is one argument of any other type then the value is converted to a real and returned as the real part and the imaginary part is zero. If there are two arguments then they must be the same type and cannot be complex. The first argument is returned as the real part and the second is the imaginary part.

5. ICHAR provides type conversion from CHARACTER to INTEGER, based on ASCII value of the argument.

6. A COMPLEX value is expressed as an ordered pair of reals, (*ar* ,*ai*), where the first is the real part and the second is the imaginary part.

7. All arguments are expressed in radians to functions that do not end with the letter D. Functions which end in the letter D take their arguments expressed in degrees.

8. INDEX(*a1*,*a2*) returns an integer value indicating the starting position of the first occurrence of *a2* in *a1* . A zero is returned if there is no match or *a1* is shorter than *a2* .

9. The string passed to the LEN function does not need to be defined before the reference to LEN is executed.

10. LGE, LGT, LLE, and LLT return the same result as the standard relational operators.

11. TRIM(*a*) returns the value of the CHARACTER expression *a* with trailing blanks removed.

12. `REPEAT(`*a*`,`*n*`)` replicates the `CHARACTER` expression *a*, *n* times where *n* is an `INTEGER` expression.

13. `ADJUSTL(`*a*`)` returns a character result which is the same as its argument except leading blanks have been removed and sufficient trailing blanks have been added to make the result the same length as *a*.

14. `ADJUSTR(`*a*`)` returns a character result which is the same as its argument except trailing blanks have been removed and sufficient leading blanks have been added to make the result the same length as *a*.

15. The functions `IOR`, `IAND`, `NOT`, and `IEOR` are provided as part of the DOD military standard MIL-STD-1753. They produce the same results for integers as the logical operators `.OR.`, `.AND.`, `.NOT.`, and `.EOR.` respectively.

16. The function reference `ISHFTC(`*i*`,`*j*`,`*k*`)` will circularly shift the rightmost *k* bits of *i* *j* places. The unshifted bits of *i* are unchanged in the result. The bits shifted out one end are shifted into the opposite end. *k* must be in the range 1-32.

17. The function reference `IBITS(`*i*`,`*j*`,`*k*`)` extracts a field of *k* bits from the value *i* beginning at position *j*. The value *j*+*k* must be in the range 1-32.

18. The function reference `BTEST(`*i*`,`*j*`)` returns `.TRUE.` if the j^{th} bit if *i* is set, otherwise it returns `.FALSE.`. The functions `IBSET(`*i*`,`*j*`)` and `IBCLR(`*i*`,`*j*`)` return integer values equivalent to *i* except that the j^{th} bit has been set or cleared respectively.

19. The statement `CALL MVBITS(`*i*`,`*j*`,`*k*`,`*l*`,`*m*`)` moves *k* bits from positions *j* through *j*+*k*-1 of *i* through *m*+*k*-1 of *l*. *j*+*k* and *m*+*k* must be in the range 1 to 32.

20. Use of a `BYTE`, `WORD` or `LONG` function on the left side of an = in an assignment will cause data to be written to an absolute address (see the section **Memory Assignment** in the chapter **Expressions and Assignment Statements**). Use of a `BYTE`, `WORD` or `LONG` function in an expression will cause data to be read from an absolute address. The `[%]LOC` function is provided to return the address of a variable, an array, an array element or a subprogram. The `%` character is optional at the beginning of a `LOC` function reference.

21. The function reference `SHIFT(`*i*`,`*j*`)` will logically shift bits in *i* by *j* places. If *j* is positive, the shift is to the left. If *j* is negative, the shift is to the right. Zeros are shifted in from the opposite end.

22. The [%]VAL functions are used to pass actual arguments by value instead of by reference which is the default for FORTRAN. Any data type except CHARACTER can be passed by value. Most often [%]VAL is used to pass arguments to routines written in other languages which accept arguments by value as the default method of argument passing. It can also be used to pass arguments to FORTRAN subprograms which use the VALUE statement to define value arguments. The % character is optional at the beginning of any VAL function reference. The VAL function is only valid in an argument list.

    NOTE: The subprogram interface protocol for the PowerPC causes value arguments less than four bytes in length to be sign-extended to four bytes and passed as four byte entities. This means that there is no difference in the effect of the VAL1, VAL2, or VAL4 functions and the VAL function may be considered to be a generic function. However, this is not true of all machine architectures (including the Intel based Windows systems) and if portability is a consideration, the correct size function should be chosen.

23. The SIZEOF function is provided to get the size of any data type in a FORTRAN program. Its syntax is as follows:

    SIZEOF(*type*)

    where: *type* is any FORTRAN data type or a structure name inside of slashes.

    Example:

```
STRUCTURE /str/
  INTEGER i
  REAL*8 a
END STRUCTURE
INTEGER str_size

str_size = SIZEOF(/str/)
```

    The most common use of SIZEOF is to obtain the size of a RECORD type. On different systems, the size of a RECORD will vary. SIZEOF can be used to write portable code which allocates memory for RECORD structures dynamically. Data types such as INTEGER or REAL can also be passed to SIZEOF. SIZEOF may be used to define constants in PARAMETER statements.

24. The %REF function is used in subroutine CALL statements and function references and is provided to assist in porting programs from VAX compatible compilers. Since all FORTRAN arguments are normally passed by reference, this function has no affect on the compiled program.

25. The %DESCR function is used in subroutine CALL statements and function references and is provided to assist in porting programs from VAX compatible compilers. Since FORTRAN arguments do not have descriptors associated with them, this function has no affect, but does cause the compiler to issue a warning message.

26. LEN_TRIM(*a*) returns the length of the CHARACTER expression *a* with trailing blanks removed.

Argument Ranges and Results Restrictions

The second argument of the remaindering functions below must not be zero:

```
MOD
AMOD
DMOD
```

Zero is returned if the value of the first argument of the sign transfer functions below is zero:

```
ISIGN
SIGN
DSIGN
```

The argument of the square root functions below must not be negative:

```
SQRT
DSQRT
```

The following square root functions for complex numbers return the principal value with the real portion greater than or equal to zero. When the real portion is zero, the imaginary portion is greater than or equal to zero.

```
CSQRT
CDSQRT
```

The argument of the logarithmic functions below must be greater than zero:

```
ALOG
DLOG
ALOG10
DLOG10
```

Both portions of a complex number cannot be zero when passed as an argument to the logarithmic functions below:

```
CLOG
CDLOG
```

Automatic argument reduction permits the argument of the trigonometric functions below to be greater than 2π:

```
SIN
DSIN
COS
DCOS
TAN
DTAN
```

The argument of the arccosine functions below must be less than or equal to one and the range of the result is between $-\pi/2$ and $\pi/2$ inclusively:

```
ASIN
DASIN
```

The argument of the arccosine functions below must also be less than or equal to one and the range of the result is between 0 and π inclusively:

```
ACOS
DACOS
```

The range of the result for arctangent functions below is between $-\pi/2$ and $\pi/2$ inclusively:

```
ATAN
DATAN
```

If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$. The arguments must not both have the value zero. The range of the result for ATAN2 and DATAN2 is between $-\pi$ and π inclusively.

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 77 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.

GLOBAL DEFINE

A GLOBAL DEFINE subprogram takes the following form:

```
GLOBAL DEFINE
[global-declaration]
...
END
```

where: *global-declaration* is any FORTRAN declaration which does not define physical storage.

STRUCTURE declarations, EXTERNAL statements, INLINE statements, and PARAMETER statements and their related type declarations are valid within a GLOBAL DEFINE subprogram. The symbols defined are then visible to an entire FORTRAN source file. The use of a GLOBAL DEFINE subprogram can remove redundant declarations and reduce internal symbol table sizes and speed compilations. There can be any number of GLOBAL DEFINE subprograms in a source file, but a symbol name cannot be used before its declaration.

It should be noted that symbol names defined in EXTERNAL, INLINE, or PARAMETER statements can no longer be implicitly defined in a subsequent program unit. For

```
GLOBAL DEFINE
      PARAMETER (i=10)
END

INTEGER a(10)

DO i=1,10
      a(i) = i
END DO
END
```

is not a valid program since a PARAMETER cannot be used as a loop induction variable. The PARAMETER declaration can be over-ridden with a local type declaration, however, as follows:

```
GLOBAL DEFINE
      PARAMETER (i=10)
END

INTEGER i,a(10)

DO i=1,10
      a(i) = i
REPEAT
END
```

INLINE STATEMENT

As an extension to standard FORTRAN, the Absoft implementation supports the INLINE statement to allow programmers to insert object code directly into a FORTRAN program. This is useful for customizing the language to the host system. The syntax of the INLINE statement is as follows:

```
INLINE ([identifier1=con1[,identifier2=/con1[,con2.../]]])
…
CALL identifier1[(arg1[,arg2...])]
variable = identifier2[(arg1[,arg2...])]
```

An INLINE declaration can have the same format as a PARAMETER statement or it can substitute a list of constants for an identifier instead of a single constant. All constants must be of type INTEGER. The identifier can then be referenced as if it were a subroutine or function. Instead of generating a call to an external function, the compiler will insert the constant or constant list directly into the object code. Each constant is output as a 32-bit opcode. If an argument list is given, the actual arguments will be passed in the same fashion as they are passed to an external routine. If the identifier is referenced as a function, the result must be returned using the standard call-return methods.

APPENDIX A

Using Structures and Pointers

This appendix is given to provide more information and examples on using STRUCTUREs, POINTERs, and RECORDs in Absoft Fortran 77. The most common use of these data types is when mixing FORTRAN with other languages such as C or Pascal that support these types of data structures. Often, operating systems and software packages that provide graphic interfaces make use of these types of data structures. The extensions to FORTRAN 77 provided with the Absoft compiler are rich enough to allow a programmer to do all of this type of programming in FORTRAN.

COMMON USE OF STRUCTURES

A structure is a composite or aggregate data type that consists of a grouping of two or more data items. Structures are typically used to group related data items together. For example, the following structures could be used to describe the time of day and the date:

```
STRUCTURE /time/
      INTEGER hour
      INTEGER minute
      INTEGER second
END STRUCTURE

STRUCTURE /date/
      INTEGER month
      INTEGER day
      INTEGER year
END STRUCTURE
```

Instances of structures are declared with RECORD statements and fields of structures are referenced with the '.' operator as follows:

```
RECORD /time/ this_time, /date/ this_date

this_time.hour = 9
this_time.minute = 29
this_time.second = 45

this_date.month = 6
this_date.day = 19
this_date.year = 1962
```

Structures can contain structures. This is done by placing structure declarations within other structure declarations or by placing `RECORD` statements inside of structure declarations as follows:

```
STRUCTURE /time/
      INTEGER hour
      INTEGER minute
      INTEGER second
END STRUCTURE

STRUCTURE /complete_time/
      STRUCTURE /date/ d
            INTEGER month
            INTEGER day
            INTEGER year
      END STRUCTURE
      RECORD /time/ t
END STRUCTURE
```

Fields in nested structures are referenced with multiple '.' operators:

```
RECORD /complete_time/ appointment

appointment.d.month = 6
appointment.d.day = 19
appointment.d.year = 1962

appointment.t.hour = 9
appointment.t.minute = 29
appointment.t.second = 45
```

A structure field reference can be placed anywhere a scalar variable is valid in a FORTRAN program. Entire records can be passed as arguments or placed in assignment statements to assign all of the fields of one record to another of the same type as follows:

```
RECORD /complete_time/ appointment
RECORD /time/ this_time

this_time.hour = 9
this_time.minute = 29
this_time.second = 45

appointment.d.month = 6
appointment.d.day = 19
appointment.d.year = 1962

appointment.t = this_time     ! assign entire structure
```

COMMON USE OF POINTERS

A pointer in Absoft Fortran 77 is an `INTEGER*4` variable which contains an address. A pointer-based variable defines storage that is pointed to by a pointer. A pointer-based variable cannot be referenced until its associated pointer variable has been assigned. Most often, a pointer variable is assigned through a reference to the `LOC` function or by a

call to a routine which returns an address of dynamically allocated memory. For example, the following program fragment could be used to dynamically allocate an array:

```
GLOBAL DEFINE
  INCLUDE "Types.inc"
  INCLUDE "Memory.inc"
END

SUBROUTINE dynamic(size)
INTEGER*4 size
INTEGER*4 array(1)                  ! 1 defeats bounds checking
POINTER(p_array, array)

p_array = NewPtr(VAL(size*4))       ! 4 bytes for each element

DO i = 1, size
  array(i) = 0                      ! Fill new array with zeros
END DO
END
```

Pointers and Optimization

The introduction of pointers into FORTRAN gives the programmer the option of aliasing any storage which is visible to the compiler. Certain types of aliasing can invalidate optimized code. Therefore, when optimization is turned on (with the **-O** option), pointers should only be assigned with a direct use of the `LOC` function or via a call to a routine which allocates memory dynamically. Any other usage has the potential to invalidate programs which are compiled with optimization enabled. Pointers to variables in `COMMON` should also not be passed as arguments. The following is an example which may cause problems when optimized:

```
PROGRAM dont_optimize
COMMON /com/ a,b,c
POINTER (preal,r)

preal = LOC(a) + 4
...
END
```

The optimizer may not be aware that whenever the value of `b` changes, the value of `r` also changes and vice versa. This problem can be solved by declaring storage which can be aliased as `VOLATILE` (see section 12.17).

```
PROGRAM optimize
COMMON /com/ a,b,c
VOLATILE b
POINTER (preal,r)

preal = LOC(a) + 4
...
END
```

Pointers as Arguments

It is not legal to declare a pointer-based variable as a dummy argument. Pointer variables and pointer-based variables can be passed as actual arguments however. When a pointer variable is passed, the address the pointer contains is passed by reference. The corresponding dummy argument should be declared as `INTEGER`. When a pointer-based variable is passed as an actual argument, the data that the associated pointer points to is passed by reference. This is equivalent to passing the pointer variable by value.

```
INTEGER itarget
POINTER (pint, itarget)

...
CALL sub(pint, itarget)
...
END

SUBROUTINE sub(ptr, idum)
INTEGER ptr, idum, ptr_target
POINTER (p_target, ptr_target)
VOLATILE idum, ptr_target

p_target = ptr
...
END
```

In the above example, the main program passes a pointer and a pointer-based variable. In the subroutine `sub`, the dummy argument `ptr` contains the address of the other dummy argument, `idum`. After the assignment of `ptr` to `p_target`, a reference to `ptr_target` is equivalent to a reference to `idum`. Note that this is a situation where the `VOLATILE` statement must be used to defeat certain optimizations.

MIXING POINTERS AND STRUCTURES

Unlike some other FORTRAN implementations, Absoft Fortran 77 allows pointer variables and pointer-based variables to be structure fields. This is useful for building dynamic data structures such as the linked list defined in the following example:

```
STRUCTURE /list/ my_list
      RECORD /list/ next
      POINTER (pnext,next)
      INTEGER field
END STRUCTURE
```

The list is linked together by assignment to the field `my_list.pnext` with the address of the next list record. Fields in the chain of list structures can be accessed with the pointer-based field `next`. For example `my_list.next.field` references the `field` in the second element of the linked list and `my_list.next.next.field` references the `field` in the third element of the linked list.

FUNCTIONS WHICH RETURN POINTERS

On many systems, functions which return pointers can be declared as `INTEGER*4`. However, some systems and/or compilers distinguish an address from an integer in terms of how the function result is passed. A function which returns a pointer and a call to a function that returns a pointer can be declared as follows:

```
FUNCTION pointer_to_int()
INTEGER pointed_to
POINTER (pointer_to_int, pointed_to)
INTEGER get_mem_result
POINTER (get_mem, get_mem_result)
EXTERNAL get_mem

pointer_to_int = get_mem(4)         ! allocates a 4 byte integer
pointed_to = 0                      ! initializes memory to 0
RETURN
END
```

The above example returns a pointer to a freshly allocated four byte integer which is initialized to zero. The symbol `pointer_to_int` is used to set the address and the symbol `pointed_to` is used to address the memory. The function `get_mem` is defined as returning a pointer to an `INTEGER`. The symbol `get_mem_result` is used for the purpose of giving a type to `get_mem` and should not be referenced. A reference to `pointer_to_int` would be similar to the reference to `get_mem` in the above example.

Pointers to C strings

A common problem when interfacing FORTRAN with C is that functions are often written in C which return C strings. A C string is a string of characters terminated with a byte of zero. Since there is no data type in FORTRAN which matches a C string, strings returned from C functions cannot be directly manipulated as `CHARACTER` data. The following example demonstrates a method of copying from a pointer to a C string to a FORTRAN `CHARACTER` variable.

```
CHARACTER*80 space_for_result
INTEGER C_fun, C_result, Cstring_pointer
POINTER (Cstring_fun, C_fun)
POINTER (Cstring_pointer, C_result)

Cstring_pointer = Cstring_fun()
CALL copy_Cstring(C_result,space_for_result)
...
END

SUBROUTINE copy_Cstring(Cstring,target)
CHARACTER Cstring(*), target*(*)

target = ' '            ! initialize to blanks
DO i=1, LEN(target)
      IF (Cstring(i) == CHAR(0)) EXIT
      target(i:i) = Cstring(i)
END DO
END
```

POINTER-BASED FUNCTIONS

A pointer-based variable can be an external function name. When this is done the associated pointer variable must be set to the address of the function which is to be called. The following is a simple example of this type of function reference.

```
INTEGER fun, pb_fun, fun_res
EXTERNAL fun, pb_fun
POINTER (pf, pb_fun)

pf = LOC(fun)
fun_res = pb_fun()
END
```

The above example calling `pb_fun` is equivalent to calling `fun`. No checking is done to insure that the address contained in the pointer variable is valid.

Appendix B

Error Messages

The first part of this appendix lists runtime error numbers and their meanings. These numbers are assigned to the `IOSTAT` specifier variable in I/O statements. The last two sections list the possible error messages from the compiler.

RUNTIME I/O ERROR MESSAGES

This section lists runtime error numbers and their meanings. These numbers are assigned to the `IOSTAT` specifier variable in I/O statements. When using the **-C** option for better runtime error reporting, these errors appear as:

```
? System Error:
? The system cannot find the file specified
? OPEN(UNIT=1,...
File "t.f"; Line 23
```

Low-level file system errors:

1	invalid function
2	file not found
3	path not found
4	too many open files
5	access deined
6	invalid interal file identifier
7	storage control blocks destroyed
8	insufficient memory
9	invalid block address
10	environment incorrect
11	incorrect program format
12	invalid access code
13	invalid data
14	insufficient memory
15	invalid drive
16	current directory cannot be removed
17	file cannot be moved to a different disk drive
18	no more files
19	media is write protected
20	specified drive cannt be found
21	the drive is not ready
22	the device does recognize the command
23	data error
24	command length is incorrect

25	drive seek error
26	the specified disk cannot be accessed
27	the specified sector cannot be found
28	the printer is out of paper
29	cannot write to specified device
30	cannot read from specified device
31	device is not responding
32	the file is already open by another process
33	another process has locked the file
34	the wrong disk is the drive
36	too many files open for sharing
38	reached end of file
39	the disk is full

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 instrinsic are not conformable

COMPILER ERROR MESSAGES — SORTED ALPHABETICALLY

The example programs shown after each error message will produce the error.

adjustable array is not a dummy argument Adjustable arrays are only allowed as dummy arguments.

```
PROGRAM main
INTEGER m(n), n
```

alpha character expected The compiler is expecting an alpha character, but has encountered a digit or special character.

```
INTEGER i, j
= i + j
```

argument to SIZEOF is not a data type The argument of the `SIZEOF` function must be a valid data type or structure name.

```
REAL a, b
b = sizeof(a)
```

argument type mismatch When using statement functions, the data type of the actual argument must match the dummy argument of the referenced statement function.

```
REAL area, a, b, ans
INTEGER i, j
area(a, b)=a * b
ans = area(i, j)
```

array boundary error The use of option **-C** during compilation will check for attempts to exceed array boundaries.

```
CHARACTER text(10)
text(100) = 'a'
```

array declaration error An array declarator must follow the required format.

```
CHARACTER*10 text(10
```

assignment to DO variable The value of the `DO` variable cannot be altered within the `DO` loop.

```
      DO 100 i =1,5
          i = i + 1
100   CONTINUE
```

ASSIGN statement error Required syntax must be followed when using the ASSIGN statement.

```
ASSIGN 100 n
```

blank lines not valid in VS Free-Form If the **-N112** option is used, a program unit cannot contain any blank lines.

```
PRINT *, "This is an example to show that blank lines"

PRINT *, "can't appear in VS FORTRAN Free-Form"
```

branch is further than 32k: use -N11 option This message is generated by the compiler when the program being compiled contains a branch that requires long addressing. Recompile the program with the **-N11** option.

cannot have an ENTRY in a routine with VALUE VALUE statements cannot appear in a program unit which contains ENTRY statements.

```
SUBROUTINE figure(aa)
REAL aa
VALUE aa
ENTRY fig2(aa)
RETURN
END
```

cannot reference a pointer based function When a pointer is a function, the pointer based variable is only present to define the data which the returned pointer points to.

```
INTEGER pbv
POINTER (ptr, pbv)
EXTERNAL ptr
a = pbv
```

conditional compilation is nonstandard **-x** allows conditional compilation. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
      WRITE (*,*) 'This is an example of allowing'
X     WRITE (*,*) 'conditional compilation using -x'
```

continuation character expected Using option **-8**, a line must begin with a continuation character if the previous line ended with one.

```
PRINT *, "In this example, the compiler is going to be &
          expecting a continuation character."
```

DATA statement syntax error Proper syntax must be used for DATA statements.

```
DATA i, j/ 1, 2
```

%DESCR function ignored The %DESCR function is provided for compatibility with existing source code. It is recognized by the parser, but no code is generated for it. The warning indicates that the statement should be examined.

division by zero Dividing a number by zero is not allowed.

```
REAL a
a = 5.2/0.0
```

duplicate BLOCKDATA initialization of COMMON Within a program, `COMMON` block names cannot be duplicated from one `BLOCKDATA` program unit to another.

```
BLOCK DATA one
COMMON /area1/ pi, area
DATA pi/3.1415/
END

BLOCK DATA two
COMMON /area1/ p, a
DATA p/3.14/
END
```

duplicate COMMON or NAMELIST declaration A variable can only appear once in a `COMMON` block or `NAMELIST` declaration.

```
COMMON /area2/pi, b
COMMON /area1/pi, b
```

duplicate DATA initialization A variable may only be initialized once by a `DATA` statement.

```
DATA i, i/10, 20/
```

duplicate label definition A statement label must be unique to a program unit.

```
      PRINT 100, "Hello"
100   FORMAT (t37, a)
      PRINT 200, "World"
100   FORMAT (t40, a)
```

duplicate name in UNION Field declarations of `UNION` declarations cannot have been previously declared or be dummy arguments.

```
INTEGER long, med1, med2
UNION
   MAP
     INTEGER*4 long
   END MAP
   MAP
     INTEGER*2 med1, med2
   END MAP
END UNION
```

duplicate program unit declaration Each named program unit must have a unique name.

```
SUBROUTINE error14()
RETURN
END

PROGRAM error14
CALL error14()
END
```

duplicate STRUCTURE name Within a program unit, each `STRUCTURE` must have a unique name.

```
STRUCTURE /date/
    INTEGER day
END STRUCTURE

STRUCTURE /date/
    INTEGER time
END STRUCTURE
```

duplicate variable declaration A specific symbolic name can appear in only one type declaration statement per program unit.

```
CHARACTER*10 xyz
INTEGER xyz
```

ELSE or END IF without IF (e) THEN Each occurrence of an `ELSE` or `END IF` statement must be matched with a corresponding `IF (e) THEN` statement.

```
LOGICAL z
IF (a .gt. 10) z = .true.
END IF
```

$ELSE, $ELSEIF or $ENDIF without $IF Each occurrence of an $ELSE, $ELSEIF or $ENDIF statement must be matched with a corresponding $IF *expr* statement.

END DO or REPEAT without DO Every occurrence of an `END DO` or `REPEAT` statement must be matched with a `DO` statement.

```
DO i=1,2
    PRINT *, "Welcome"
REPEAT
END DO
```

END SELECT without SELECT CASE Each occurrence of an `END SELECT` statement must have a corresponding `SELECT CASE`.

```
CASE (1)
  PRINT *, 1
CASE DEFAULT
END SELECT
```

END STRUCTURE without STRUCTURE All appearances of the `END STRUCTURE` statement must be matched with a `STRUCTURE` statement.

```
INTEGER mm
INTEGER dd
INTEGER yy
END STRUCTURE
```

END UNION without UNION Each occurrence of an `END UNION` statement must correspond to a `UNION` statement.

```
MAP
   INTEGER*2 i
END MAP
MAP
   INTEGER*1 j1, j2
END MAP
END UNION
```

escape sequences in strings are nonstandard **-K** allows for escape sequences in strings. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
WRITE (*,*) 'What goes up...\n Must come down'
```

EXIT or CYCLE outside of a loop These statements may only appear within `DO` loops.

```
READ *, score
IF (score .le. 50.0) THEN
    EXIT
END IF
```

expecting a MAP statement Each `END MAP` statement must have a corresponding `MAP` statement.

```
UNION
   MAP
     INTEGER*4 long
   END MAP
   MAP
     INTEGER*2 med1, med2
   END MAP
END MAP
END UNION
```

expecting an argument list or subscript References to a specific array element must include the subscript. Functions declared by `EXTERNAL` and `INTRINSIC` statements must have an argument list when used.

```
INTEGER m(10,10)
i = m
```

expecting end of statement Unless a symbol indicating a comment or a semicolon for multiple statements is encountered, information cannot appear on a line once the statement has ended.

```
      CLOSE (10) a, b, c
```

format specifier is not repeatable A repeat factor can only precede those edit descriptors denoted to be repeatable.

```
100   FORMAT (3"Hello!", t19,a)
```

format string has unmatched parenthesis The number of opening parenthesis must match the number of closing parenthesis.

```
100   FORMAT (4(t12,a/)
```

format string has unmatched quote Character strings must have an ending quote to match the beginning quote.

```
100   FORMAT("FORTRAN, t10, a)
```

format string missing opening parenthesis Format specifications must begin with an opening parenthesis and end with a closing parenthesis.

```
100   FORMAT t35,a
```

Fortran 90 free source form is nonstandard **-8** allows this format to be used. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32.**

```
      WRITE (*,100) 'This is an example of',&
      &             'using Fortran 90 Free&
      & Source Form'  !comment line
1   0      0 FORMAT (a/a)
```

GLOBAL statement is nonstandard **-M** applies the `GLOBAL` statement to all COMMON block declarations. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
      COMMON /area2/c, d
```

GOTO non-integer label The destination of an assigned `GOTO` statement must be an integer which contains an address defined with an `ASSIGN` statement.

```
      REAL a
      GOTO a
```

IBM VS free-form is nonstandard **-N112** allows for this format to be used.This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
      WRITE (*,100) 'This is an example of-
 IBM VS Free-
Form'
"     comment line
1    0       0 FORMAT (t12,a)
```

illegal DO loop label The DO loop termination statement cannot be one of the statements listed in the Loop Statement Section.

```
      DO 100 i = 1,5
      IF (i .lt. 3) THEN
        PRINT *, "Less"
100   END IF
```

illegal DO variable A DO variable must be either an INTEGER, REAL, or DOUBLE PRECISION scalar variable.

```
      DIMENSION i(100)

      DO i=1,3
        j = i**2
      END DO
```

illegal dummy argument Once a variable has been referenced in a program unit, its name cannot be used within the same program unit as a dummy argument in an ENTRY.

```
      SUBROUTINE sub()
      INTEGER i
      i=10
      ENTRY ent(i)
      i=20
```

illegal EQUIVALENCE or UNION of COMMON blocks An EQUIVALENCE or UNION statement cannot increase the size of a COMMON block by adding storage units prior to the first item in the COMMON block.

```
      INTEGER m(10), n(10)
      COMMON/ area / m
      EQUIVALENCE (m(1), n(2))
```

illegal expression The rules and syntax governing arithmetic, character, logical and relational expressions must be followed.

```
      i = 1 (+2)
```

illegal external symbol An external symbolic name cannot be a variable declared as an array or a symbolic named constant declared by a `PARAMETER` statement.

```
INTEGER m(10)
EXTERNAL m
```

illegal format repeat count The repeat count must be a positive integer constant.

```
100   FORMAT (-3(t12,a/))
```

illegal format specifier Format specifiers must follow the correct syntax and be supported by this implementation of FORTRAN.

```
100   FORMAT (a1.5)
```

illegal function call Function calls cannot appear in the specification section of a program. Use the **-O** compiler option to compile constant function references.

```
PARAMETER (a=sqrt(3582.00))
```

illegal IF clause An `IF` clause cannot be one of the statements listed in the `IF` Statement Section.

```
IF (i .gt. 100) END
```

illegal initialization Dummy arguments and functions cannot be initialized with a DATA statement.

```
SUBROUTINE calc(m, n)
INTEGER m, n
DATA m, n/100, 200/
```

illegal metacommand The $ character is the lead-in for a compiler directive. The characters encountered after the $ in question are not recognizable as a compiler directive.

```
$DEFIND FLAG=1
```

illegal POINTER variable A `POINTER` variable cannot be one that has already been declared as a variable.

```
COMPLEX pa
INTEGER i
POINTER (pa,i)
```

illegal POINTER based variable In declaring a `POINTER` based variable, it cannot be a dummy argument or appear in `COMMON`, `GLOBAL`, `EQUIVALENCE`, or `VALUE` statements.

```
COMMON /area/a
POINTER (pa, a)
```

illegal statement function name A dummy argument cannot be used as the name of a statement function.

```
SUBROUTINE output(i)
i(j) = i**2
```

illegal statement in BLOCK DATA procedure `IMPLICIT`, `PARAMETER`, `DIMENSION`, `COMMON`, `SAVE`, `EQUIVALENCE`, `DATA`, and type declarations are the only allowable `BLOCK DATA` statements.

```
BLOCK DATA
COMMON /area/ pi ,a
DATA pi, a/ 3.14, 0.0/
RETURN
END
```

illegal statement in GLOBAL DEFINE Executable statements are not allowed in `GLOBAL DEFINE` subprograms.

```
GLOBAL DEFINE
   PARAMETER (x=3)
   PRINT *, x
END
```

illegal statement ordering The proper ordering of statements must be followed.

```
DATA pi, a/3.14, 0.0/
COMMON /area/pi, a
```

illegal structure definition The `STRUCTURE` statement must follow the proper syntax.

```
STRUCTURE calen day
    INTEGER mm
    INTEGER dd
    INTEGER yy
END STRUCTURE
```

illegal symbol in a DATA statement Record names may not appear in a `DATA` statement.

```
STRUCTURE /name/
     INTEGER i
     INTEGER J
END STRUCTURE
RECORD /name/m
DATA m /11/
```

illegal syntax A syntax error has been detected, but the compiler is unable to determine the exact problem.

```
PRINT *, (m(i), j=1,10
```

illegal use of operator .NOT. is an unary operator and may only be used with one operand.

```
IF (i .not. j) i=10
```

illegal use of RECORD name A RECORD name may be used only where permitted.

```
RECORD /str/ a, b, c
DATA a/10,20/
```

illegal use of POINTER based variable Once a POINTER based variable has been declared, it cannot appear in a COMMON, GLOBAL, or EQUIVALENCE statement.

```
POINTER (pa, i)
EQUIVALENCE (i, j)
```

illegal use of statement function argument Statement function arguments cannot be used as an array reference within the statement function expression.

```
INTEGER i(10)
ISTF(i,j) = i(10) + j
```

illegal value parameter a VALUE statement cannot appear in the main program and its arguments cannot be an array or CHARACTER variable. The VAL function is only valid in an argument list.

```
SUBROUTINE FIG(text)
CHARACTER*10 text
VALUE text
```

illegal variable in NAMELIST Symbolic constants and symbolic names declared as INTRINSIC or EXTERNAL are not allowed in NAMELIST statements.

```
PARAMETER (i=10)
NAMELIST /list/ i
```

incorrect format hollerith count The count on a hollerith string must match the number of characters appearing after the H.

```
100   FORMAT (90Hhollerith)
```

increment expression cannot be zero A DO variable cannot be incremented by zero.

```
INTEGER m(5)
DO i = 1,5,0
   m(i) = i
END DO
```

intrinsic function data type mismatch The data type of an intrinsic function's argument must match the required data type. A common mistake is to use an integer where a real number is required.

```
a = sqrt(100)
```

INTRINSIC name used as EXTERNAL An intrinsic function appearing in an `EXTERNAL` statement cannot appear in an `INTRINSIC` statement.

```
EXTERNAL max
INTRINSIC max
CALL max(max)
```

invalid argument Only valid symbolic names can appear in arguments.

```
INLINE (code=z'10')
CALL sub(code)
```

invalid argument to EQUIVALENCE statement Allowable arguments are: variable names, array element names, array names, and character substring names. Dummy argument names and function names are not allowed.

```
SUBROUTINE sub(i,j)
EQUIVALENCE (i,j)
```

invalid CASE statement Every case statement must follow the proper syntax and the data type of the case selector must match that of the `SELECT CASE` argument.

```
SELECT CASE (i)
   CASE '1'
   CASE DEFAULT
END SELECT
```

invalid constant expression A variable cannot appear where a constant is required.

```
SELECT CASE (i)
  CASE (i)
END SELECT
```

invalid data type for control list specifier Specifiers that appear in control lists must be of the data type specified.

```
OPEN (unit=10, access='direct', file='error', recl=12.0)
```

invalid $DEFINE Some element of the symbol or defining expression is not a valid FORTRAN symbol or component of a constant expression.

```
$DEFINE L.test = 1
```

invalid field name Field names must correspond to a field name within the specified `STRUCTURE` declaration.

```
STRUCTURE /data/day
   INTEGER mm
   INTEGER dd
   INTEGER yy
END STRUCTURE
day.y = 1991
```

invalid $IF or $ELSEIF Some element of the expression is not a valid component of a constant expression.

```
$IF FLAG = 1        ! should be == or .EQ.
```

invalid INCLUDE This statement must use a valid file specification and follow the proper syntax.

```
INCLUDE (mistake)
```

invalid I/O control list specifier or syntax Specifiers appearing in an I/O control list must be supported by this implementation of FORTRAN and follow the specified syntax.

```
READ (5,100,ENDI = 10) a, b, c
```

invalid I/O list or syntax The I/O list must follow the proper syntax.

```
WRITE (*,*), "Greetings"
```

invalid option Only valid compiler options may be used when compiling a program.

invalid statement function dummy argument The dummy argument of a statement function must be a variable. It cannot be a symbolic named constant or a symbolic name declared in an `INTRINSIC` or `EXTERNAL` statement.

```
PARAMETER (a = 2.0)
calc(a) = (a**2)/10.0
```

invalid statement label A statement label must be an unsigned integer in the range of 1 to 99999.

```
      PRINT 100, "Hello, World"
-100  FORMAT (t29, a)
```

invalid $UNDEFINE Some element of the symbol is not a valid FORTRAN symbol.

```
$UNDEFINE L.test
```

label missing Only statement labels that exist in a particular program unit may be referenced within that unit.

```
      WRITE (*,200) "What goes up, must come down"
250   FORMAT (t25,a)
```

local variable never referenced A warning is given for all variables that were declared, but never referenced.

```
INTEGER i
PRINT *, "FORTRAN 77"
```

MAP outside of UNION MAP declarations can only appear within a `UNION` declaration.

```
  MAP
    INTEGER*2 i
  END MAP
UNION
  MAP
    INTEGER*1 j1, j2
  END MAP
END UNION
```

missing END statement A program unit must be terminated with an `END` statement.

missing END STRUCTURE All appearances of the `STRUCTURE` statement must be matched with an `END STRUCTURE` statement.

```
STRUCTURE /date/
   INTEGER mm
   INTEGER dd
   INTEGER yy
```

missing END UNION Each `UNION` statement must be matched with an `END UNION` statement.

```
UNION
   MAP
     INTEGER*4 long
   END MAP
   MAP
     INTEGER*2 med1, med2
   END MAP
```

missing label on FORMAT statement All `FORMAT` statements must begin with a statement label.

```
FORMAT (t24, a)
```

missing operand ANSI FORTRAN 77 does not allow more that one arithmetic operator to appear consecutively.

```
a = 8.9*-7.2
```

multiple statement line is nonstandard ANSI FORTRAN 77 (option **-N32**) does not allow multiple statement lines.

```
i = 1; j = 2; k = 3
```

non-constant case expression The value selector of a `CASE` expression must be a constant.

```
SELECT CASE (i)
  CASE (i)
    PRINT *, 1
  CASE DEFAULT
END SELECT
```

nonstandard comment Comment lines must begin with the character C or an asterisk in ANSI FORTRAN 77 (option **-N32**).

```
!     This is an example of a nonstandard comment line
```

nonstandard constant Option **-N32**, for ANSI FORTRAN 77, does not allow constant extensions.

```
CHARACTER*20 greet
DATA greet/14Hgood afternoon/
```

nonstandard constant delimiter Option **-N32**, for ANSI FORTRAN 77, does not allow extensions of the standard delimiters.

```
WRITE (*,*) "Hello, World"
```

nonstandard data initialization Initialization of blank `COMMON` blocks by a `DATA` statement is not allowed in ANSI FORTRAN 77 (option **-N32**).

```
COMMON a, b
DATA a, b/ 10.2, 8.42/
```

nonstandard edit descriptor Option **-N32**, for ANSI FORTRAN 77, does not allow edit descriptor extensions.

```
      WRITE (*,100) 199,199,199
100   FORMAT (z4,o7.6,b9)
```

nonstandard intrinsic function Option **-N32**, for ANSI FORTRAN 77, does not allow intrinsic function extensions.

```
CHARACTER*20 text
text = repeat('a',20)
```

nonstandard I/O specifier Option **-N32**, for ANSI FORTRAN 77, does not allow I/O specifier extensions.

```
OPEN (unit=10, file='numbers', action='both')
```

nonstandard operator Option **-N32**, for ANSI FORTRAN 77, does not allow extensions of the standard operators.

```
IF (a<12) THEN
   WRITE (*,*) a
END IF
```

nonstandard statement When using option **-N32**, statements that are an extension of ANSI FORTRAN 77 may not appear in the program.

```
IMPLICIT none
```

nonstandard symbolic name Symbolic names longer than 6 characters and containing characters other than letters and numerals are not allowed in ANSI FORTRAN 77 (option **-N32**).

```
CHARACTER*20 hello_world
hello_world = 'Hello, World'
```

nonstandard type When using option **-N32**, data types that are an extension of ANSI FORTRAN 77 may not appear in the program.

```
INTEGER*2 i
i=145
```

non-standard use of assignment operator The assignment operator (=) can appear only between a variable and an expression in arithmetic, logical,and character assignment statements.

```
IF (m = 10) STOP
```

not an intrinsic function The symbolic name appearing in an `INTRINSIC` statement must be a valid intrinsic function name.

```
INTRINSIC sort
```

not expecting a label An initial statement line cannot contain a statement label accompanied by a continuation character.

```
100  +PRINT *, "FORTRAN 77"
```

number of continuation lines is nonstandard ANSI FORTRAN 77 (option **-N32**) limits the number of continuation lines to 19.

```
      WRITE (*,*) 'a
     +b
     +c
     +d
     +e
     +f
     +g
     +h
     +i
     +j
     +k
     +l
     +m
     +n
     +o
     +p
     +q
     +r
     +s
     +t
     +u
     +v
     +w
     +x
     +y
     +z'
```

numeric overflow The value of a variable must be within the range allowed for that data type.

```
      i = 51234567890
```

one trip do loops are nonstandard **-d** executes all `DO` loops at least once. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
      DO 100 i=3,1
         WRITE (*,*) 'FORTRAN 77'
100   CONTINUE
```

optional use of FORMAT specifier is nonstandard **-N16** allows for the optional use of `FORMAT` specifier (`FMT=`) while the `UNIT` specifier is present (`UNIT=`).This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
      WRITE (unit=6, 100) 'Hello'
100   FORMAT (a)
```

overriding dynamic storage allocation is nonstandard **-s** allows overriding dynamic storage allocation. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
CALL calc(a)
END

SUBROUTINE calc(aa)
REAL aa, bb, cc
RETURN
END
```

PARAMETER declaration error Symbolic names of data types: integer, real, double precision, and complex, must correspond to an arithmetic expression. A character type variable must be matched with a character constant, and a logical variable to a logical constant. Also, proper syntax must be used.

```
PARAMETER (a=1 b=2)
```

procedure name conflicts with symbol A procedure name cannot be the same as that of a variable declared as an array.

```
INTEGER i(10)
CALL i
```

program unit declaration syntax error Proper syntax must be followed when declaring a program unit.

```
SUBROUTINE calculate a, b
```

program unit has invalid use of COMMON name A `COMMON` name cannot be used if it has not been defined.

```
SAVE /a/
```

recursive STRUCTURE definition A field declaration in a `STRUCTURE` definition cannot make reference to the `STRUCTURE` in which it is contained.

```
STRUCTURE /date/ day
   STRUCTURE /date/ day
     CHARACTER*10 calen
   END STRUCTURE
END STRUCTURE
```

RETURN statement in main program unit `RETURN` statements cannot appear in the main program.

```
PROGRAM main
INTEGER m(10)
DO i=1,5
   m(i) = i**i
REPEAT
RETURN
END
```

SAVE statement syntax error A `SAVE` statement must follow the proper syntax.

```
SAVE (radius)
```

size of type is undefined A structure name appearing as the argument of a `SIZEOF` function must first be defined.

```
i = sizeof(/structure/)
```

specification statement syntax error Proper syntax must be followed for all specification statements.

```
DIMENSION m(100) n(100)
```

spelling error Keywords must be spelled correctly.

```
PRIN *, "Good Afternoon"
```

statement cannot be reached In every program, the possibility must exist for every executable statement to be used during execution.

```
      PRINT *, "RED"
      GOTO 200
      PRINT *, "WHITE"
200   PRINT *, "BLUE"
```

symbol defines illegal storage in GLOBAL DEFINE `GLOBAL DEFINE` subprograms cannot contain declarations that define physical storage.

```
GLOBAL DEFINE
   INTEGER i,l
END
```

symbol in UNION was in EQUIVALENCE or UNION Once a variable is used in an `EQUIVALENCE` list or `UNION` declaration, it cannot appear in another `UNION` declaration.

```
EQUIVALENCE (long, med1)
UNION
   MAP
     INTEGER*4 long
   END MAP
   MAP
     INTEGER*2 med1, med2
END UNION
```

synch error in intermediate code Internal compiler error — Call Absoft.

unbalanced parenthesis Each occurrence of an opening parenthesis must be matched with a corresponding closing parenthesis.

```
a = MOD(105/68
```

undetermined size array not valid in I/O statement Standard FORTRAN restricts whole array I/O of assumed dimension arrays in subprograms.

```
SUBROUTINE out(text)
CHARACTER*5 text(*)
PRINT *, text
```

UNION not contained in STRUCTURE When nesting `UNION` and `STRUCTURE` declarations, overlapping cannot occur.

```
STRUCTURE /str/
  UNION
     MAP
       INTEGER i
     END MAP
     MAP
       INTEGER j
     END MAP
END STRUCTURE
  END UNION
```

unit required in I/O statement A unit number, either an integer or an asterisk, is required in all I/O statements.

```
      WRITE (FMT = 400) "Hello, World"
400   FORMAT (a)
```

unmatched $ELSE No $ENDIF exists which matches the nesting level of the $ELSE.

unmatched $IF or $ELSEIF No $ENDIF exists which matches the nesting level of the $IF or $ELSEIF.

unsupported data type Only data types supported by this implementation of FORTRAN may be used within a program unit.

```
INTEGER*12 i
i = 3
```

unsupported extension Only extensions supported by this implementation of FORTRAN may be used in a program.

```
OPTION +x
```

unterminated DO loop Each `DO` loop must be terminated with a loop termination statement (`END DO`, `REPEAT`, or a statement with a corresponding statement label).

```
DO 100 i = 1,10
  j = j + 1
```

unterminated IF block Every occurrence of a block `IF` must be terminated with an `END IF` statement.

```
READ *, grade
IF (grade.le.50) THEN
   PRINT *, "FAIL"
ELSE
   PRINT *, "PASS"
```

unterminated SELECT CASE block Each occurrence of a `SELECT CASE` statement must be terminated with an `END SELECT` statement.

```
SELECT CASE (i)
   CASE (1)
   CASE DEFAULT
```

variable data type is undefined All variables in a program unit must be declared either implicitly or explicitly.

```
IMPLICIT NONE
a = 5
```

variable misaligned A variable should be aligned on a boundary which matches its size.

```
INTEGER*4 i
INTEGER*2 j
INTEGER*1 k
COMMON /area/k, i, j
```

VAX tab format is nonstandard **`-V`** allows for the use of this format This is not standard ANSI FORTRAN 77, and cannot be combined with **`-N32`**.

```
      WRITE (*,100) 'This is an example
      1 of VAX Fortran Tab-Format' !comment line
1   0   0     FORMAT (t12,a)
*comment, comment, comment
```

VOLATILE statement syntax error The rules and syntax governing the `VOLATILE` statement must be followed.

```
VOLATILE (a, i)
```

wide source format is nonstandard **-W** extends the last statement column to 132. This is not standard ANSI FORTRAN 77, and cannot be combined with **-N32**.

```
 WRITE (*,*) 'This is an example of using wide format, which accepts
+statements that exceed column 72'
```

wrong number of function arguments The number of arguments to statement functions and intrinsic functions must agree with the number of arguments that are required.

```
a = mod(1)
```

wrong number of array dimensions Arrays must be referenced with the same number of dimensions as they were declared with.

```
INTEGER m(5,5)
m(5) = 100
```

COMPILER ERROR MESSAGES — SORTED NUMERICALLY

0	illegal syntax
1	numeric overflow
2	division by zero
4	invalid statement label
5	alpha character expected
6	spelling error
7	invalid option
8	invalid INCLUDE
9	specification statement syntax error
10	invalid argument
11	program unit declaration syntax error
12	duplicate label definition
14	duplicate program unit declaration
15	local variable never referenced
16	duplicate variable declaration
17	not an intrinsic function
18	duplicate COMMON or NAMELIST declaration
19	SAVE statement syntax error
20	array declaration error
21	PARAMETER declaration error
22	invalid constant expression
23	missing END statement
24	variable data type is undefined
25	variable misaligned
26	invalid statement function dummy argument
27	illegal statement ordering
28	invalid argument to EQUIVALENCE statement
29	missing label on FORMAT statement
30	statement cannot be reached
31	synch error in intermediate code
32	END DO or REPEAT without DO
33	unbalanced parenthesis
35	illegal expression
36	wrong number of function arguments
37	missing operand
38	wrong number of array dimensions
39	illegal DO variable
40	expecting an argument list or subscript
41	ASSIGN statement error
42	label missing
43	ELSE or END IF without IF (e) THEN
44	unterminated DO loop
45	unterminated IF block
46	expecting end of statement

47	END SELECT without SELECT CASE
48	unterminated SELECT CASE block
50	invalid CASE statement
51	invalid I/O control list specifier or syntax
53	illegal variable in NAMELIST
55	illegal EQUIVALENCE or UNION of COMMON blocks
56	intrinsic function data type mismatch
57	procedure name conflicts with symbol
58	DATA statement syntax error
59	illegal symbol in a DATA statement
60	duplicate BLOCKDATA initialization of COMMON
61	illegal statement in BLOCK DATA procedure
62	illegal value parameter
64	illegal external symbol
66	illegal statement function name
67	illegal use of operator
68	array boundary error
70	EXIT or CYCLE outside of a loop
71	unsupported data type
72	non-standard use of assignment operator
73	illegal dummy argument
74	GOTO non-integer label
75	illegal DO loop label
76	illegal IF clause
77	assignment to DO variable
78	increment expression cannot be zero
79	non-constant case expression
80	not expecting a label
81	continuation character expected
82	blank lines not valid in VS Free-Form
84	duplicate DATA initialization
85	VOLATILE statement syntax error
86	illegal initialization
87	unit required in I/O statement
88	invalid I/O list or syntax
89	format string missing opening parenthesis
90	illegal format repeat count
91	illegal format specifier
92	format specifier is not repeatable
93	format string has unmatched quote
94	incorrect format hollerith count
95	format string has unmatched parenthesis
96	adjustable array is not a dummy argument
97	RETURN statement in main program unit
98	INTRINSIC name used as EXTERNAL
99	illegal use of statement function argument
100	undetermined size array not valid in I/O statement
101	unsupported extension

102 invalid data type for control list specifier
103 argument type mismatch
104 illegal function call
106 nonstandard statement
107 nonstandard comment
108 nonstandard type
109 nonstandard data initialization
110 nonstandard edit descriptor
111 nonstandard intrinsic function
112 nonstandard I/O specifier
113 nonstandard constant
114 nonstandard symbolic name
115 one trip do loops are nonstandard
116 conditional compilation is nonstandard
117 Fortran 90 free source form is nonstandard
118 extended range do loops are nonstandard
119 IBM VS free-form is nonstandard
120 escape sequences in strings are nonstandard
121 VAX tab format is nonstandard
122 wide source format is nonstandard
123 GLOBAL statement is nonstandard
124 number of continuation lines is nonstandard
125 overriding dynamic storage allocation is nonstandard
126 optional use of FORMAT specifier is nonstandard
127 nonstandard operator
128 nonstandard constant delimiter
129 multiple statement line is nonstandard
130 illegal structure definition
131 END STRUCTURE without STRUCTURE
132 invalid field name
133 duplicate STRUCTURE name
134 illegal use of RECORD name
135 missing END STRUCTURE
136 END UNION without UNION
137 MAP outside of UNION
138 duplicate name in UNION
139 UNION not contained in STRUCTURE
140 missing END UNION
141 expecting a MAP statement
142 symbol in UNION was in EQUIVALENCE or UNION
143 illegal POINTER variable
144 illegal POINTER based variable
145 illegal use of POINTER based variable
146 recursive STRUCTURE definition
147 illegal statement in GLOBAL DEFINE
148 symbol defines illegal storage in GLOBAL DEFINE
149 cannot have an ENTRY in a routine with VALUE
150 cannot reference a pointer based function

151 size of type is undefined
152 argument to SIZEOF is not a data type
153 branch is further than 32k: use N11 option
154 32 bit address to a global: do not use N12 option
155 program unit has invalid use of COMMON name
156 invalid $DEFINE
157 $ELSE, $ELSEIF or $ENDIF without $IF
158 unmatched $ELSE
159 illegal metacommand
160 invalid $UNDEFINE
161 invalid $IF or $ELSEIF
162 unmatched $IF or $ELSEIF
163 %DESCR function ignored

Appendix C

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

Character	Dec	Oct	Hex	Description
	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
&	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

Character	Dec	Oct	Hex	Description
@	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
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

Character	Dec	Oct	Hex
?	128	200	80
	129	201	81
‚	130	202	82
f	131	203	83
„	132	204	84
…	133	205	85
†	134	206	86
‡	135	207	87
ˆ	136	210	88
‰	137	211	89
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

Character	Dec	Oct	Hex
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
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 D

Bibliography

References on the FORTRAN language

These books and manuals are useful references for the FORTRAN language and the floating point math format used by Absoft Fortran 77 on Windows.

Page, Didday, and Alpert, *FORTRAN 77 for Humans*, West Publishing Company (1983) **Highly recommended for beginners**

Kruger, Anton, *Efficient FORTRAN Programming*, John Wiley & Sons, Inc. (1990) **Highly recommended for beginners**

Loren P. Meissner and Elliot I. Organick, *FORTRAN 77*, Addison-Wesley Publishing Company (1980)

Harry Katzan, Jr., *FORTRAN 77*, Van Nostrand Reinhold Company (1978)

J.N.P. Hume and R.C. Holt, *Programming FORTRAN 77*, Reston Publishing Company, Inc. (1979)

Harice L. Seeds, *FORTRAN IV*, John Wiley & Sons (1975)

Jehosua Friedmann, Philip Greenberg, and Alan M. Hoffberg, *FORTRAN IV, A Self-Teaching Guide*, John Wiley & Sons, Inc. (1975)

James S. Coan, *Basic FORTRAN*, Hayden Book Company (1980)

Brian W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley Publishing Company (1976)

Brian W. Kernighan and P.J. Plauger, *The Elements of Programming Style*, McGraw-Hill Book Company (1978)

American National Standard Programming Language FORTRAN, X3.9-1978, ANSI, 1430 Broadway, New York, N.Y. 10018

COMPUTER, *A Proposed Standard for Binary Floating-Point Arithmetic*, Draft 8.0 of IEEE Task P754, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 (1981)

M. Abramowitz and I.E. Stegun, *Handbook of Mathematical Functions*, U.S. Department of Commerce, National Bureau of Standards (1972)

Fortran Forum, Association for Computing Machinery. Phone: 1-212-869-7440.

Fortran Journal, Fortran Users Group. Phone: 1-714-441-2022.

References on Windows Programming

These books are suggested reading for learning how to program in the Win32 API for Windows. Most of these books are available in book stores.

Microsoft Win32 Programmer's Reference, Volumes 1-5, Mircosoft Press (1993)

Charles Petzold, *Programming Windows 3.1*, Mircosoft Press (1992)

Jerry Richter, *Advanced Windows, The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95,* Mircosoft Press (1995)

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 *ProFortran 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
2781 Bond Street
Rochester Hills, MI 48309

Technical Support:	(248) 853-0095	9am - 3pm EST
FAX	(248) 853-0108	24 Hours
email	support@absoft.com	24 Hours
World Wide Web	http://www.absoft.com	

Appendix F

VAX Extensions

This appendix lists the VAX FORTRAN extensions to FORTRAN 77 that are supported by Absoft Fortran 77. For details about porting code from the VAX, see the **Porting Code** chapter of the *ProFortran User Guide*.

VAX FORTRAN STATEMENT EXTENSIONS

ACCEPT
BYTE
DECODE
DO WHILE...END DO
DO...END DO
ENCODE
EXIT
IMPLICIT NONE
INCLUDE
OPEN statement extensions
- ACCESS='APPEND'
- CARRIAGECONTROL=
- DISP= (same as DISPOSE=)
- DISPOSE= (or DISP=)
  - 'KEEP' and 'SAVE'
  - 'PRINT' and 'DELETE'
  - 'PRINT/DELETE'
  - 'SUBMIT'
  - 'SUBMIT/DELETE'
- MAXREC=
- NAME=
- NOSPANBLOCKS
- ORGANIZATION=
- RECORDSIZE= (same as RECL=)
- READONLY
- SHARED
- TYPE= (same as STATUS=)

MAP...END MAP
NAMELIST
READ (NAMELIST directed)
RECORD
STRUCTURE...END STRUCTURE
TYPE
UNION...END UNION
VOLATILE
WRITE (NAMELIST directed)

VAX FORTRAN DATA TYPE EXTENSIONS

BYTE
COMPLEX*16
DOUBLE COMPLEX
INTEGER*2
INTEGER*4
LOGICAL*2
LOGICAL*4
REAL*4
REAL*8
*'nnn'*X and *'nnn'*O format for hexadecimal and octal constants

VAX FORTRAN INTRINSIC FUNCTION EXTENSIONS

ACOSD	COSD	FLOATJ	IIOR	JIBCLR	JMIN1
AIMAX0	DACOSD	IAND	IISHFT	JIBITS	JNINT
AIMIN0	DASIND	IBCLR	IISHFTC	JIBSET	JNOT
AJMAX0	DATAND	IBITS	IISIGN	JIDIM	JZEXT
AJMIN0	DATAN2D	IBSET	IMAX0	JIDINT	LOC
ASIND	DCMPLX	IEOR	IMAX1	JIDNNT	MVBITS
ATAND	DCONJG	IIABS	IMIN0	JIEOR	NOT
ATAN2D	DCOSD	IIAND	IMIN1	JIFIX	[%REF]
BITEST	[%]DESCR	IIBCLR	IMOD	JINT	SIND
BJTEST	DFLOAT	IIBITS	ININT	JIOR	SIZEOF
BTEST	DFLOTI	IIBSET	INOT	JISHFT	TAND
CDABS	DFLOTJ	IIDIM	IOR	JISHFTC	[%]VAL
CDCOS	DIMAG	IIDINT	ISHFT	JISIGN	ZEXT
CDEXP	DREAL	IIDNNT	ISHFTC	JMOD	
CDLOG	DSIND	IIEOR	IZEXT	JMAX0	
CDSIN	DTAND	IIFIX	JIABS	JMAX1	
CDSQRT	FLOATI	IINT	JIAND	JMIN0	

OTHER VAX FORTRAN EXTENSIONS

Aggregate assignment
Comment lines beginning with "!"
Conditional compilation with "D" in column 1
DATA statements mixed with declarations
Edit descriptors without field widths
Extended range DO loops
Extended source lines with 132 columns (with **-W** option)
Initialization in declaration statements (i.e. INTEGER I/31/)
Initialization of COMMON blocks outside of BLOCK DATA
Nested INCLUDE statements
Non-INTEGER array and substring indexes
PARAMETER statements without ()
RECL defines 32-bit words (with **-N51** option)
Symbol names may include "$" and "_" in names
Tab-Format source form (with **-V** option)
Use of intrinsics in PARAMETER (with **-O** or **-N41** options)
VAX file names for implicit unit connections to a file

- `O Z Q $` edit descriptors

Appendix G

Language Systems Fortran Extensions

This appendix describes the implementation of Language Systems Fortran extensions supported by the Absoft Fortran 77 Compiler.

STRING

`STRING` is a type statement and is used to declare a string entity compatible with Pascal strings. The first data byte of a `STRING` is set to the logical length of the string, limiting the length of a character entity declared in this manner to 255 bytes. With the exception of internal files and substring expressions, a `STRING` can be used anywhere a CHARACTER argument can be used.

POINTER

This is a declaration statement for declaring variables that will contain the address of other variables. The syntax is:

```
POINTER /type/ v [, v] ...
```

where:

`type` is a type or structure name.
`v` is a variable name.

A `POINTER` may be used in an assignment statement or in an integer expression to manipulate the address it contains. It is dereferenced (the value of what it points to is extracted) by appending the operator, `^`, to the name.

LEAVE

This statement has the same meaning as the `EXIT` statement (See **Control Statements**) and provides a convenient means for abnormal termination of a `DO` loop. The `LEAVE` and `EXIT` statements cause control of execution to be transferred to the statement following the terminal statement of a `DO` loop or block `DO`.

GLOBAL

The syntax of this statement is similar to the `SAVE` statement (see **Specification and `DATA` Statements**). The variables specified with a `GLOBAL` statement are made externally visible. It can be used in conjunction with `GLOBAL DEFINE` to create global variables for the whole file.

CGLOBAL

The syntax of this statement is similar to the SAVE statement (see **Specification and DATA Statements**). The variables specified with a CGLOBAL statement are not affected by case folding and are made externally visible. It can be used in conjunction with GLOBAL DEFINE to create global variables for the whole file.

PGLOBAL

The syntax of this statement is similar to the SAVE statement (see **Specification and DATA Statements**). The variables specified with a PGLOBAL statement are folded to upper case and made externally visible. It can be used in conjunction with GLOBAL DEFINE to create global variables for the whole file.

CEXTERNAL

The syntax of this statement is similar to the EXTERNAL statement (see **Specification and DATA Statements**). The functions specified with a CEXTERNAL statement pass all of their arguments by value as the default. An argument can be passed by reference if the %REF instrinsic function is used.

PEXTERNAL

The syntax of this statement is similar to the EXTERNAL statement (see **Specification and DATA Statements**). The functions specified with a PEXTERNAL statement pass all of their arguments by value as the default. An argument can be passed by reference if the %REF instrinsic function is used. Note that this statement has the same effect as the Absoft PASCAL EXTERNAL declaration.

INT1, INT2, AND INT4

INT1, INT2, and INT4 are type conversion intrinsic functions that convert their arguments to 1-, 2-, and 4-byte integers respectively.

JSIZEOF

This function returns an integer that represents the size of its argument in bytes. It is identical to the SIZEOF function (see **Programs, Subroutines, and Functions**).

%VAL, %REF, AND %DESCR

When appearing in the formal argument list of a FUNCTION or SUBROUTINE declaration statement, these statements use the syntax of a function reference, but have the affect of declaring the passing method of the argument. %DESCR has no effect (but will generate a

warning diagnostic), %REF is the default, and %VAL is the same as using the VALUE declaration statement (see **Specification and DATA Statements**). The % is not optional in this usage.

LANGUAGE SYSTEMS INCLUDE FILES

There are some syntactical difficulties in several of the Language Systems API include files that are ignored by the Language Systems Fortran compiler. These are detected by the Absoft Fortran 77 compiler which will issue a diagnostic when they are encountered.