intel

# i860™ Microprocessor Fortran Language Reference Manual

Version 1

January 1990

240726-001

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

# Table of Contents

**Chapter 9. Structural Statements**



**Chapter 10. Specification Statements**



**Chapter 11. Records and Structures**



**Chapter 12. DATA Statement**

# Chapter 1
# Overview

## 1.1. The Green Hills Fortran Documentation Set

The Green Hills Fortran standard compiler documentation set includes a User's Manual and Language Reference Manual. Additional documentation on product installation and execution are provided separately. You may also need to refer to separate documentation describing the architecture, assembler and linker for your target system.

## 1.2. Documentation Conventions

The following conventions are used throughout the Green Hills Fortran document set:

- Square brackets ([]) are used to indicate optional parameters in a syntax definition.
- Ellipses (...) are used to indicate that the items preceding may be repeated one or more times.
- Uppercase words and letters in a syntax definition indicate that the letter or keyword should be entered exactly as shown.
- Lowercase and/or italicized words and letters in a syntax definition indicate substitution parameters, and are place holders for user-supplied values.

## 1.3. Language Reference Manual Structure

The Language Reference Manual describes language constructs, syntax and general programming guidelines. Language features and extensions supported by the compiler are discussed, along with any restrictions and compatibility with other Fortran compilers. The Language Reference Manual is intended as a support document and is not a language tutorial.

### Overview

The Overview describes the structure of the documentation for the compiler.

### Fortran Language Introduction

The Introduction describes the Fortran language, history, currently defined standards, and validation criteria.

### Fortran Syntax

The Syntax section describes the Fortran character set, basic structure of the language, and programming guidelines.

### Data Types, Constants, and Variables

This section describes the Fortran data types, constants and variables.

### Operators and Expressions

Fortran language expressions and operators are discussed in this section, including operator precedence tables and expression syntax.

### Fortran Program Structure

The Program Structure section specifies the language structure, statement ordering rules and subprogram definition.

### Input/Output

This section covers Fortran I/O system in general, as well as the related topics of records, files, I/O units and data transfer.

### Fortran Language Statements

Language statements are presented in alphabetical order within one or more statement category.

### Functions and Subprograms

This section covers all subroutine or function calls available to the Fortran programmer as part of the Green Hills Fortran compiler package.

### Tables and References

These sections include various tables and reference materials useful to the Fortran language programmer.

# Chapter 2
# FORTRAN Language Introduction

FORTRAN is the oldest and most proven high-level computer programming language in existence today. The origins of FORTRAN date back to the mid 1950's. Up to that time, most programs were written in complicated machine languages that required considerable coding and debugging time. FORTRAN, the first high level language, was designed to reduce the communications gap between people and computers using a language structure that people could read and understand easily. A FORTRAN compiler is a program that translates the written FORTRAN programs into the specific machine language that a computer can read.

FORTRAN was developed primarily to solve problems that involve the manipulation of numerical data. The language syntax was designed to accommodate standard algebraic notation. Using FORTRAN, programmers can employ the computer to translate existing mathematical formulas for processing. The name FORTRAN was derived from this principle of FORmula TRANslation.

Through the late 1950's and early 1960's, FORTRAN evolved as the primary scientific programming language. In 1966, the American National Standards Institute (ANSI) approved the first FORTRAN language standard providing a new level of FORTRAN program portability. Programs written in ANSI standard FORTRAN-66 could be run on a variety of different computers with little or no program modification.

Through the late 1960's and early 1970's, computers became more sophisticated and the demands on software became more complicated. FORTRAN applications expanded to include more general program tasks such as sophisticated input/output, text manipulation, and structured programming facilities. In 1978, the American National Standards Institute gave final approval to an extended FORTRAN standard that came to be known as FORTRAN-77. Today, FORTRAN is still the most widely used programming language among mathematicians, scientists, and engineers.

## 2.1. FORTRAN Standards

FORTRAN implements the ANSI FORTRAN-77 (Full Language) Standard, ANSI X3.9-1978 and the Military Standard FORTRAN, as described in the document "FORTRAN, DOD Supplement to American National Standard X3.9-1978, MIL-STD-1753". It is also compatible with the Berkeley 4.3BSD f77 compiler and the VAX/VMS FORTRAN V4.6 compiler.

## 2.2. Validation

FORTRAN is tested by running the FORTRAN Compiler Validation System Version 2.0 (1978) from the U.S. Office of Software Development and the U.S. Department of Commerce, National Technical Information Service.

## 2.3. Form of Presentation

This manual defines the form and interpretation of language elements specific to FORTRAN. It does not attempt to teach general programming concepts. Familiarity with a beginning programming language, such as BASIC, is ample background for taking on FORTRAN using this manual.

Each of the following chapters covers one general category of language concepts, such as data types, expressions, program structure, specification statements, control statements, or intrinsic functions. Concepts, such as variables and procedures, are defined specifically in terms of the FORTRAN language. Explanations are elementary in nature, but concise to accommodate more experienced programmers. This manual may be used to learn the FORTRAN programming language, or it may be used as a FORTRAN reference manual.

# Chapter 3
# FORTRAN Syntax

A computer programming language, like a natural human language, is based on rules of syntax. Syntax is the predefined order or arrangement of language elements that produces meaning.

This manual uses the following typographical conventions within examples to highlight various entities that constitute the language's syntactic structure.

- Words in UPPERCASE LETTERS indicate language keywords.
- Words in lowercase letters indicate syntactic items, such as "symbolic-name" or "number", and literal names within examples, such as the variable names "var1" and "var2".
- Items enclosed in square brackets [ ] are optional.
- A horizontal ellipsis ... indicates that you can repeat the preceding optional item any number of times. A vertical ellipsis indicates an ambiguous continuation of the preceding items.

## 3.1. The Character Set

The fundamental element of any programming language is the character. FORTRAN uses the standard ASCII character set, which consists of uppercase and lowercase letters (A through Z), digits (0 to 9), and a group of special characters that have a specific interpretation in FORTRAN. You can collectively refer to letters and digits as alphanumeric characters.

There are 20 special characters in FORTRAN as shown in the table below.

**FORTRAN Special Characters**

| Character | Name | Character | Name |
|---|---|---|---|
| ' | apostrophe | = | equals sign |
| * | asterisk | ( | left parenthesis |
| | blank | - | minus sign |
| : | colon | % | percent sign |
| , | comma | + | plus sign |
| . | decimal point | ) | right parenthesis |
| $ | dollar sign | / | slash |
| & | ampersand | _ | underscore |
| " | quotation mark | ! | exclamation mark |
| < | left angle bracket | > | right angle bracket |

FORTRAN also recognizes the ASCII control characters that signify carriage returns, tabs, new line feeds, and form feeds.

All characters conform to a collating order. The collating order defines a hierarchy among the characters for the purpose of sorting character strings. FORTRAN can compare two character strings a character at a time according to the ASCII collating sequence. Under ASCII conventions, all characters correspond to numeric values that determine the hierarchy. Please refer to the "ASCII and Hexadecimal Conversion Table" for a listing of the collating sequence.

## 3.2. Symbolic Names and Keywords

FORTRAN identifiers include symbolic names and keywords. A symbolic name is a sequence of alphanumeric characters. The dollar sign and underscore characters are valid in a symbolic name. However, the first character in a symbolic name must be a letter (A through Z or a through z). Symbolic names identify program entities, such as constants, variables, arrays, and program units.

A keyword is a sequence of letters that identifies a particular statement or serves as a separator within a statement. For example, COMMON, IMPLICIT, DATA, IF, and THEN are FORTRAN keywords that have a specific purpose in the appropriate statement. Note that certain keywords satisfy the description of a symbolic name. Keywords have a specific purpose only if used within the specific context defined in FORTRAN.

### 3.2.1. Scope of Symbolic Names

Symbolic names vary in scope, depending upon what a given symbolic name identifies. Different program entities can be divided into two classes pertaining to scope: global and local.

A global entity has a scope that covers the entire executable program. A symbolic name that identifies a global entity cannot identify a second global entity in the same executable program.

A local entity has a scope that covers a single program unit. A symbolic name that identifies a local entity cannot identify a second local entity in the same program unit. A symbolic name that identifies a global entity in a program unit cannot identify a local entity within the same program unit, except for common block and external function names.

The following program entities are classified as global to a FORTRAN executable program:

- block data subprogram
- common block
- external function
- main program
- subroutine

The following program entities are classified as local to a single FORTRAN program unit:

- array
- constant
- dummy procedure
- statement function
- variable

There are two local program entities that have a scope somewhat smaller than a program unit: a dummy argument for a statement function statement and an implied DO variable in a DATA statement.

A symbolic name that identifies a dummy argument for a statement function statement has a scope of that statement. See the "Statement Functions" section in the FORTRAN Program Structure chapter for more information.

A symbolic name that identifies the DO variable for an implied DO in a DATA statement has a scope of the implied DO list. See the "DATA Statement" chapter for more information.

## 3.3. Statements and Lines

All FORTRAN syntactical items, such as keywords, symbolic names, statement labels, constants, operators, and special characters form FORTRAN statements. The statements you write using the FORTRAN language are translated into computer instructions. Statements are classified as executable or nonexecutable.

An executable statement is any statement that specifies some processing action, such as the PRINT or CONTINUE statement. Executable statements execute sequentially in the order that they are placed in a program unit. Execution of an executable program begins with the first statement in the main program. Control statements, such as GOTO and CALL, transfer the execution sequence to a different point in the program. Statements that transfer the execution sequence are considered executable statements.

Nonexecutable statements define and classify program units, specify entry points in subprograms, specify editing information, and specify initial values and execution characteristics for data.

Each FORTRAN statement is written on one or more program lines. A program line is a sequence of character positions. You can refer to character positions as columns. The columns are numbered consecutively beginning with 1, and proceeding from left to right. There are three kinds of program lines in FORTRAN: initial lines, continuation lines, and comment lines. Each type serves a different purpose within a program.

There are two ways to format a program line in FORTRAN: the free-field format and the column-based format. This section describes both formats in detail.

## 3.4. Free-field Line Format

The free-field format is indicated by the use of a tab character before column 72 of the initial line of a statement. If the initial line of a statement is in free-field format then all continuation lines will also be in free-field format. To indicate that a continuation line and all subsequent continuations lines are in the free-field format, use an ampersand, &, in column 1 to indicate that the line is a continuation of the previous line.

The free-field format enables you to enter program lines at a standard CRT type terminal without regard to specific column-oriented line fields. In free-field format, the length of a program line is unlimited. However, if your terminal has a width of 80 character positions, you might want to limit your lines to 80 characters to prevent the lines from running off the screen and out of view. On an 80 character terminal, you can use the entire 80 columns as an initial line for statement text with or without a statement label. The initial line is the first line used to hold a statement. If the statement requires more than the 80 columns in the initial line, you can place the remainder of the statement in continuation lines.

**Example:**

```
10<tab>FORMAT('This long statement contains a tab so it is in free-field format')
```

## 3.5. Column-based Program Format

The column-based format originated in the early days of FORTRAN when programs were stored on punch cards. Program lines consist of precisely defined fields. Each field can only contain a specific kind of information. Support of the column-based format enables the FORTRAN compiler to process FORTRAN programs written before the free-field format was available. Many experienced FORTRAN programmers still prefer the structure and organization column-based formatting lends to a program. FORTRAN interprets a line in column based format unless it contains a tab character in columns 1 through 72, or an ampersand character, &, in column 1 of a continuation line.

### 3.5.1. Program Line Fields

In the column-based format, the 80 columns in a program line are divided into four fields. Each field is reserved for a specific kind of information.

Columns 1 through 5 are reserved for the statement label. You can use statement labels to identify and reference specific statements in a program.

Column 6 is reserved for a continuation mark. The continuation mark indicates that the line is a continuation of the preceding line.

Columns 7 through 72 are reserved for the actual statement text. This can be extended to columns 7 through 132 by using the OPTIONS/EXTEND_SOURCE statement, or by selecting the equivalent command line option. Please refer to your system-specific User's Guide for details on command line options and defaults.

Columns 73 through 80 can contain identifying information or notations. The compiler ignores all characters in this field. In the days of punch card storage, columns 73 through 80 contained a sequence number used to identify the position of an individual punch card within a stack of cards. The following example shows a line in column-based format. Columns 1 through 5 of the first line contain a line number, 10000. Column 6, the continuation column, contains a blank, indicating the initial line of a statement. Columns 7 through 72 contain the initial line of a FORMAT statement. Columns 73 through 80 contain a card sequence number which is ignored by FORTRAN.

**Example:**

```
10000 FORMAT('This line contains no tab, it is in column-based format') 01234567
```

### 3.5.2. Initial and Continuation Lines

A statement can span several program lines. The initial line is the first line used to hold a statement. If a statement exceeds column 72 of the initial line, you can place the remainder of the statement in continuation lines. Standard FORTRAN allows a maximum of 19 continuation lines per statement, for a total of 1320 character positions. This limitation is not enforced by the compiler, therefore any number of continuation lines are accepted, up to a total of 10,000 character positions.

You must distinguish continuation lines from initial lines in a program. Any character in column 6 except a blank or the digit 0 serves as a continuation mark, indicating that the line is a continuation line.

An alternative free-format continuation line is indicated by placing a <TAB> character in column one, followed by a number from '1' through '9'.

## 3.6. Comment Lines

Comment lines provide a method of program notation or documentation. Even high level language programs can appear somewhat confusing and ambiguous when you try to read them. The comment line enables you to place notes and explanations within a program to help other programmer's understand your programming intentions and to help you remember at a later date what the program does.

You must distinguish comment lines from the other kinds of lines in a program. If the letter C, an asterisk(*) or pound-sign (#) appears in column 1, Columns 2 through 80 may contain any characters in the FORTRAN character set including all blanks. Comment lines can appear anywhere within a program unit and do not affect the executable program in any manner.

In addition, an exclamation point (!) may be used in any column to signal that the characters following the exclamation point up through the end-of-line are comments. Therefore, an exclamation point in column one is functionally equivalent to an asterisk (*), pound-sign (#), or C in the same position.

**Example:**

```
C     Standard FORTRAN comment lines
*
      AA = A + B       ! Compute AA as sum of A and B
      C = 'Hi there!'
!
!     The INTEGER statement above includes a trailing comment line
!     However, the exclamation point in the string 'Hi there!' is
!     within a quoted string and not interpreted as a comment delimiter
```

## 3.7. Debugging Statements

A debugging statement may be distinguished from standard FORTRAN source code by placing one of the following letters in column one; D, d, X or x. Debugging statements are compiled as ordinary statements when the -X82 option is specified at compile time. The default operation (-Z82) is to treat debugging statements as comment lines.

All debugging statements must be preceded by either 'D' or 'X' in column one, either upper or lowercase, and may include labels and continuation lines, as shown in the example below.

**Example:**

```
C     The following code is only compiled when -X82 is used
C
D     DO 10 I=1,10
X     J = (I**2) + (D / 4)
D10   CONTINUE
```

## 3.8. Statement Labels

You can use statement labels to identify and reference individual statements in a program. You can label any statement, but only labeled executable statements can be referenced with the statement label. For example, the GOTO statement transfers execution to an executable statement identified with a statement label. The GOTO statement requires the statement label as a parameter for reference.

A statement label is a sequence of one to five digits. At least one of the digits must be nonzero. You can place statement labels anywhere in columns 1 through 5 of the statement's initial line. Continuation lines cannot contain the statement label.

Statement labels have the scope of a program unit. Every statement label in a program unit must be unique. Blank characters and leading zeros are insignificant for distinguishing different statement labels. For example, the label "777" is equivalent to "00777."

## 3.9. Statement Order

Different configurations and groupings of statements form program units. However, there are certain rules that apply to the order of statements and comments within a program unit.

- Comment lines can appear anywhere before the END statement.
- The PROGRAM statement can appear only as the first statement of a main program. The FUNCTION, SUBROUTINE, and BLOCK DATA statements can appear only as the first statement in a subprogram.
- FORMAT and ENTRY statements can appear anywhere before the END statement.
- PARAMETER statements can appear anywhere before DATA statements, statement function statements, and executable statements.
- IMPLICIT statements must appear before all other specification statements except PARAMETER statements and FORMAT statements.
- All other specification statements (COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, INTRINSIC, SAVE) must appear before any DATA statements.
- DATA statements can appear anywhere following the specification statements.
- All statement function statements must appear before any executable statements.
- All executable statements must appear before the END statement.
- The END statement must be the last statement in a program unit.

The table below summarizes the rules for ordering statements in a FORTRAN program unit. The horizontal lines separate the kinds of statement that you cannot mix in a program unit. The vertical lines delineate the kinds of statement that you can mix.

<table>
<tr><td rowspan="5">Comment<br>Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,<br>or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT<br>and<br>ENTRY<br>Statements</td><td rowspan="2">Parameter<br>Statements</td><td>Implicit<br>Statements</td></tr>
<tr><td>Other<br>Specification<br>Statements</td></tr>
<tr><td rowspan="2">DATA<br>Statements</td><td>Statement<br>Function<br>Statements</td></tr>
<tr><td>Executable<br>Statements</td></tr>
<tr><td colspan="3">End Statement</td></tr>
</table>

**F77 Compatibility Notes:**

When VMS compatibility mode is used, DATA statements may be used anywhere within a program unit. Standard FORTRAN (F77 compatibility mode) requires that all DATA statements appear before the first executable statement.

## 3.10. Compilation Control

### 3.10.1. INCLUDE Directive

INCLUDE is a compiler directive that causes the contents of the specified INCLUDE file to be interpreted as if they were actually part of the program body.

**Syntax**: INCLUDE 'file-spec'

where *file-spec* is the name of the file to be included, delimited by single quotes. No default filename extension is assumed.

The INCLUDE file must not start with a continuation line, but may itself contain other INCLUDE statements. The total number of nested INCLUDEs is operating system dependent, generally 16 for most UNIX systems.

The -I *string* compile time option allows the user to specify a default directory location to be searched for INCLUDE files. Please refer to the "Compile Time Options" chapter of your FORTRAN User's Manual for details on this compiler option.

**Example:**

```
FILEA.F:
      COMMON      /PCOM/I(10),J(10)
PROG.F:
      INCLUDE 'FILEA.F'
      DO 10 inum = 1,10
      j = i(inum) + 1
10    CONTINUE
      END

Compiling PROG.F results in the compiled statements:
      COMMON /PCOM/I(10),J(10)
      DO 10 inum = 1,10
      j = i(inum) + 1
10    CONTINUE
      END
```

**F77 Compatibility Notes:**

The INCLUDE directive is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 3.10.2 OPTIONS Statement

The OPTIONS statement may be used to override a subset of FORTRAN command line options in effect for a given program unit.

**Syntax:** OPTIONS option [,option]

where *option* is a valid OPTIONS string. One or more *option* values may be specified on the OPTIONS line, separated by commas.

The following table lists the valid option values, along with their command line equivalents.

| OPTIONS String | Cmd-Line Equivalent | Summary Description |
|---|---|---|
| /CHECK=ALL | none | Overflow, underflow, bounds checking |
| /CHECK=NONE | default | No run-time checks performed |
| /CHECK=OVERFLOW | n/a | Ignored |
| /CHECK=NOOVERFLOW | n/a | Ignored |
| /CHECK=UNDERFLOW | n/a | Ignored |
| /CHECK=NOUNDERFLOW | n/a | Ignored |
| /CHECK=BOUNDS | -C | Check array bounds |
| /CHECK=NOBOUNDS | none | Do not check array bounds |
| /G_FLOATING | n/a | Ignored |
| /NOG_FLOATING | n/a | Ignored |
| /I4 | none | Default operation (INTEGER*4 and LOGICAL*4) |
| /NOI4 | -i2 | Use INTEGER*2 and LOGICAL*2 as defaults |
| /F77 | none | Default operation (normal DO loop operation) |
| /NOF77 | -onetrip | One iteration DO loops |
| /NOCHECK | none | Ignored |
| /EXTEND_SOURCE | -X161 | Use columns 1 through 132 |
| /NOEXTEND_SOURCE | -Z161 | Use only columns 1 through 72 |

Any other qualifiers will generate an error at compilation time.

Only one OPTIONS statement is allowed per program unit, and must precede every other statement in the program unit, including PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements. The OPTIONS qualifiers override (or confirm) qualifiers specified on the FORTRAN command line. However, they remain in effect only to the end of the program unit which they begin, so an OPTIONS statement is needed at the beginning of every program unit in which you wish to override the command line or default qualifiers.

**Example:**

The statement

```
OPTIONS /CHECK=BOUNDS
```

will cause address references for arrays to be checked.

**F77 Compatibility Notes:**

Please note that the /F77 and /NOF77 OPTIONS specifiers do not affect compatibility mode. Rather, they affect only DO loop operations. Compatibility mode must be selected at compile-time by specifying the appropriate command line option, or taking the default operation. Please refer to your system-specific User's Guide for details on compile time options and default modes.

# Chapter 4
# Data Types

All computer programs, regardless of their degree of complexity, create, manipulate, or modify pieces of information that pertain to a given problem or task. The different values related to a problem or computing task are collectively referred to as data.

Data can consist of numeric values such as the circumference of the earth measured in kilometers or a worker's gross salary. A numeric value is represented by a series of digits. Data can also consist of textual information such as a client's name and address or abbreviations for chemical compounds in an equation. Text data is represented by strings of characters. You can use any character in the FORTRAN character set as textual data.

All data in a FORTRAN program falls into one of the following categories.

- integer
- real
- double precision
- complex
- logical
- character

FORTRAN supports the use of Hollerith data. Hollerith data provided a text processing capability for earlier versions of the FORTRAN language and is considered an extension to the FORTRAN-77 standard. It is provided primarily for compatibility with older FORTRAN standards.

## 4.1. The Implicit Data Type Convention

Programmers frequently use a symbolic name in a program to identify a particular datum or data structure. Symbolic names used in this manner serve a dual purpose. First, the name identifies the datum or data structure within a program unit. Second, the name identifies a specific data type.

You can specify the data type for a symbolic name explicitly using the type statements. Refer to the "Specification Statements" chapter for more information. In the absence of an explicit data type specification, the first letter of the symbolic name determines the data type implicitly. Symbolic names default to either the integer or real data type.

If a symbolic name has the letter I, J, K, L, M, or N as a first letter, the symbolic name has an implicit integer data type. Any other letter of the alphabet implies a real data type. For example, the following symbolic names default to the integer data type in the absence of an explicit type specification: list, increment, multiple, kilo.

The following symbolic names default to the real data type in the absence of an explicit type specification: alpha, delta, total, variation.

You can use the IMPLICIT statement to change the implicit data type convention. Refer to the "IMPLICIT Statement" section for more information. An explicit data type specification using a type statement supercedes the implicit data type convention.

The IMPLICIT NONE statement can be used to override all implicit defaults, forcing all symbolic names to be declared explicitly. Refer to the "IMPLICIT NONE Statement" section for more information.

## 4.2. "*n" Data Type Size Qualifiers

When a symbol data type declaration is made, the type and/or name may include a data type length specifier. This size qualifier overrides any length which would otherwise be implied by the statement.

**Syntax:** type[*n] symbol[*n] [,symbol[*n]...]

where type is any valid FORTRAN data type keyword and n specifies the data type length, preceded by an asterisk (*).

**Example:**

```
C                 A, B and D are INTEGER*4
C                 C and E are INTEGER*2
C
      INTEGER*4   A(5), B, C*2, D, E*2(5)
C
C     REAL        X*8, Y*16, Z*4
```

**F77 Compatibility Notes:**

The "*n" data type size qualifiers are not supported in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 4.3. Initialization in Type Declaration Statements

Variables and arrays may be initialized within a single type declaration statement, omitting the need for a separate DATA statement.

**Syntax:** type symbol/value/ [[,]symbol/value/]...

where type is a valid type declarator, symbol is a variable or array name, and value consists of one or more constants, delimited by slashes (/), that will be assigned to symbol.

**Example:**

In the following example, the value 3.1415 is assigned to the variable PI, and the values 1, 2 and 3 are assigned to the array IBUF, and the array IARRAY is filled with zeroes.

```
DIMENSION       IBUF(3)
REAL            PI /3.1415/
INTEGER         IBUF /1,2,3/, IARRAY(4) /4*0/
```

**CHARACTER Variable Initialization**

A CHARACTER variable or array of length one character may be initialized to a numeric constant within a type or DATA statement. The numeric constant must be a value from 0 to 255 decimal, specified as an integer, octal or hexadecimal value.

**Example:**

```
CHARACTER*1      CBUF /35/
CHARACTER*1      INBUF(4) /10, '4'O, 25, '5F'X/
```

**F77 Compatibility Notes:**

Initialization within a single type declaration statement is not supported in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 4.4. Integer Type

An integer is any whole positive number, any whole negative number, or zero (0). Zero is neither positive nor negative. Integers have no fractional part. Therefore, integers cannot contain decimal points.

You can write integers with a leading sign. If you omit the sign, the integer is considered to be positive. The integers 721 and +721 represent the same value.

A constant is a value in your program that does not change. Refer to the "Constants" section for more information. The following are some valid integer constants: 1, 642, +25, 9287, –41, –73268, 0.

A standard integer occupies four bytes of memory space and can represent values that range from –2147483648 up to +2147483647. However, you can specify integers that occupy one or two bytes of memory. Refer to the "Specification Statements" chapter for additional information. Your system specific User's Manual describes how integers are represented internally on your target system.

## 4.5. Real Type

In FORTRAN, a real number is any number that can express a fractional component, an exponent, or both. Real numbers can be either positive or negative. There are three forms for real numbers expressed as constants:

- basic real constant
- basic real constant with exponential notation
- integer constant with exponential notation

The basic real constant consists of an optional sign, an integer component, a decimal point, and a fractional component. Both the integer and fractional components are series of digits. The following are some examples of valid basic real constants: 1.5, +82.7, .007, 375., –794.0, –299999.

Exponential or scientific notation consists of the letter E followed by an optionally signed integer. The value of a basic real constant or an integer constant with exponential notation is the product of the constant that precedes the E (the mantissa) and the power of 10 indicated by the integer that follows the E (the exponent). The following examples show some basic real constants and integer constants with exponential notation.

```
    5.82E2  =  582.0
314159.00E–5  =  3.14159
   –.229E–3  =  –.000229
       11E5  =  1100000
      –5E–2  =  –.05
    –100E+8  =  –10000000000
```

Standard real numbers occupy four bytes of memory space. Refer to your User's Manual for information regarding the range, precision, and representation of the REAL data type on your system. You can also specify real numbers that occupy eight bytes of memory. Refer to the "Specification Statements" chapter for more information.

## 4.6. Double Precision Type

Double precision real numbers provide additional significant

digits of accuracy for real numbers. Use a double precision real number when the range and accuracy of a basic real number is inadequate for the application. There are two forms for a double precision real number.

- basic real constant with exponential notation
- integer constant with exponential notation

A double precision real number uses a slightly different exponential notation. Double precision exponential notation consists of the letter D followed by an optionally signed integer. The value of a basic real constant or an integer constant with exponential notation is the product of the constant that precedes the D (the mantissa) and the power of 10 indicated by the integer that follows the D (the exponent). The following examples show some basic real constants and integer constants with double precision exponential notation.

```
      252.7D2  =  25270.0
      .007D–1  =  .0007
   883366.0D–4  =  88.3366
837612458378183D–8  =  8376124.58378183
     –.0045D+3  =  –4.5
   –69124820D–3  =  –69124.820
```

Double precision real numbers occupy eight bytes of memory space (two numeric storage units). Refer to the your User's Manual for information regarding the range, precision, and representation of the DOUBLE PRECISION data type. You can use the DOUBLE PRECISION type statement or a REAL*8 type statement to declare double precision numbers. Refer to the "Specification Statements" chapter for more information.

## 4.7. Complex Type

To find the roots of certain mathematical equations, we must sometimes consider the roots of negative numbers. Since the set of real numbers does not allow for such a possibility, mathematicians have defined an imaginary unit that uses the symbol i and is equal to the square root of negative one. Using the imaginary unit, we can write the square root of a negative number as the product of a real number and the number i. Numbers expressed in this manner are called pure imaginary numbers.

A complex number is any number that can be expressed in the form A+Bi. A and B are real numbers and i is equal to the square root of negative one. In FORTRAN, we can write a complex number as an ordered pair of integer or real constants separated with a comma and enclosed in parentheses. The first member of the pair is the real constant and the second member of the pair is the imaginary constant as shown in the following format specification.

(real-constant,imaginary-constant)

A complex number is always stored as a pair of real values. Either constant can be positive, negative, or zero. The following examples show some valid complex numbers expressed as constants.

| | | |
|---|---|---|
| (0,2) | = | 0+2i or 2i |
| (7.5,2.2) | = | 7.5+2.2i |
| (–11,–3) | = | –11–3i |
| (–6E3,.55) | = | –600+.55i |
| (945E–2,–41E–1) | = | 9.45–4.1i |

A complex number occupies eight consecutive bytes of memory space (two consecutive numeric storage units) in a storage sequence. You can also specify complex values that occupy sixteen bytes of memory. Refer to the "Specification Statements" chapter for more information.

## 4.8. Logical Type

A logical datum can assume one of two values: true or false. The principle of binary logic is fundamental to the digital computer, having its origins in binary or Boolean arithmetic. In FORTRAN, logical data serves as a simple decision making criterion enabling a program to evaluate logical propositions. A proposition is a statement that evaluates to either true or false.

There are only two logical constants in FORTRAN: the words TRUE and FALSE, delimited on both sides with periods. You write the logical constants in a program as follows.

```
.TRUE.      or    .true.
.FALSE.     or    .false.
```

A standard logical datum occupies four bytes of memory space. However, you can specify logicals that occupy one or two bytes of memory. Refer to the "Specification Statements" chapter for additional information.

The actual value assigned to the logical operator TRUE may vary depending on operating system implementation.

FALSE is assigned a value of zero (0), and TRUE is assigned a non-zero value. The actual value of TRUE, therefore, could be any non-zero value. For example, on VAX/VMS a logical is true if and only if the low order bit is 1. On Unix, a logical is true if and only if it is non-zero.

User's wishing to rely on exact values for .TRUE. should refer to their Operating System documentation for implementation specific information.

Use the LOGICAL statement to declare a symbolic name with the logical data type. Refer to the "Specification Statements" chapter for more information.

The BYTE statement is equivalent to LOGICAL*1 and can contain signed integers in the range -127 to +128. Refer to the "BYTE Statement" section for more information.

## 4.9. Character Type

The character data type enables a program to process textual information. A character datum is a string of one or more characters delimited on both sides with apostrophes. Character data are often referred to as strings. FORTRAN recognizes and differentiates between upper- and lowercase letters. The following examples show some valid string constants.

```
'Please enter your password.'
'The limit as T goes to 0 is ... '
'PERCENTAGE OF ERROR < 7%'
```

You can use any character in the FORTRAN character set within a string including the blank character and the apostrophe. However, to represent the apostrophe within a string you must use two consecutive

apostrophes, as shown in the following example.

```
'It"s TWELVE O"Clock!' (Displays as: It's TWELVE O'Clock!)
```

The length of a character string is the number of characters, including blank characters, that appear between the apostrophes. The apostrophes used as delimiters do not count in the string length. Two consecutive apostrophes within a string count as one character. The length of a character string must be greater than zero.

```
'Please enter your password.' (String length is 27)
'It"s TWELVE O"Clock!' (String length is 20)
```

Character data occupies one character storage unit in a storage sequence for each character in the string. A character storage unit in FORTRAN is one byte of memory space.

Use the CHARACTER statement to declare a symbolic name with a character data type. Refer to the "Specification Statements" chapter for more information.

## 4.10. Hollerith Data

Hollerith data provided a text data processing capability for earlier versions for the FORTRAN language. The FORTRAN-77 standard is the first version of the language to provide the character data type that is considered superior to the Hollerith form. Hollerith data is considered an extension to the FORTRAN-77 standard provided primarily for compatibility with the earlier FORTRAN standards.

Hollerith data, like character data, is a string of characters. You can use any character in the FORTRAN character set within a Hollerith string including the blank character. The form for a Hollerith constant consists of a nonzero, unsigned, integer constant (n), the letter H, and a string of contiguous characters (c) as shown in the following format specification.

```
nHccc...c
```

where *ccc...c* represents a string of *n* characters.

The following examples show some valid Hollerith constants.

```
16HToday's date is:
11HGRAND TOTAL
4HNaCl
```

You cannot declare a symbolic name with a Hollerith data type. You can identify Hollerith data, other than constants, with a symbolic name of type integer, real, or logical using a DATA statement or READ statement.

# Chapter 5
# Constants, Variables, Arrays, and Substrings

FORTRAN provides different methods for handling the data that your program is designed to process. Different programming situations call for a different form of data representation. In certain situations, it is most appropriate to specify a data item explicitly. In other situations, it is most appropriate to specify a data item using a symbolic name that can take on new values as the program progresses. For programs that process large amounts of data, it can be most efficient to specify groups of related or similar data using one symbolic name.

To develop efficient problem solving algorithms using FORTRAN, you must understand the proper use of constants, variables, arrays, and substrings. This section describes the conceptual nature of these four entities. The descriptions refer you to the appropriate FORTRAN statements used to implement these entities in a program.

## 5.1. Constants

A constant is an explicit, literal representation of a numeric, logical, or character value in a program. Constants remain unchanged during program execution. The "Data Types" chapter shows examples of constants for the different data types.

Consider a simple program that calculates the area of a circle using the standard formula A = πr2 where r is the radius of the circle, π is equal to 3.1415926, and A is the area. The value of π remains unchanged regardless of any other value in the calculation. Therefore, to translate the area formula to FORTRAN, you can use the real constant 3.1415926 for π. If you want additional accuracy in your area calculation, you can use π expanded to the extended precision constant 3.141592653589793D0.

You can identify a constant with a symbolic name using the PARAMETER statement. When you require a long constant, such as 3.141592653589793D0, at many different locations within a program, repetitious typing of digits can become space and time consuming. To simplify the situation, you can assign a symbolic name such as "pi" to the constant 3.141592653589793D0 using the PARAMETER statement. The program substitutes the constant value wherever the assigned name "pi" appears in the program. Refer to the "PARAMETER Statement" section for more information.

### 5.1.1. Octal Integer Constants

An octal integer constant is treated as an integer data type.

Syntax: `"nnn`

where nnn is a string of digits from 0 to 7 inclusive, prefixed by a quotation mark (").

The following table shows some sample valid and invalid octal integer constants.

| Sequence | Valid | Reason |
|---|---|---|
| "01237 | YES | Correct form |
| 3456O | NO | Missing quotation mark |
| "123" | NO | No trailing quotation marks allowed |
| '01234 | NO | Incorrect punctuation form |
| "13579 | NO | Invalid octal digit (9) |

**F77 Compatibility Notes:**

Octal integer constants are not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 5.1.2. Octal and Hexadecimal Typeless Constants

Octal and hexadecimal constants may be used wherever numeric constants are allowed. A maximum of 128 bits (16 bytes) may be specified in an octal or hexadecimal constant, allowing a maximum of 43 octal digits, or up to 32 hexadecimal digits. If more digits are specified than can be stored in the corresponding data type, the constant will be truncated on the left. If the constant specified is less than the total storage for the corresponding data type, the value is zero-filled.

Octal and hexadecimal constants assume a data type based on use, and have no implicit data type beforehand.

**Syntax:** 'nnnnn'O

-or- 'nnnnn'X

where nnnnn represents a string of valid octal or hexadecimal digits, followed by the letter 'X' for hexadecimal constants or 'O' for octal constants.

#### 5.1.2.1. Octal Constants

An octal constant consists of a string of valid octal digits delimited by apostrophes, followed by the letter 'O'. The letter 'O' may be either upper or lower case. Valid octal digits are the numbers 0 through 7, inclusive.

The following table shows some sample valid and invalid octal sequences.

| Sequence | Valid | Reason |
|---|---|---|
| '01234567'o | YES | Correct form |
| '255'O | YES | Correct form |
| 3456O | NO | Missing apostrophes |
| '01234O' | NO | Apostrophes misplaced |
| '13579'o | NO | Invalid octal digit (9) |

**F77 Compatibility Notes:**

Octal constants are not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

#### 5.1.2.2. Hexadecimal Constants

A hexadecimal constant consists of a string of valid hexadecimal digits delimited by apostrophes and followed by the letter 'X'. The letter 'X' may be specified in either upper or lower case.

Valid hexadecimal digits include numbers 0 through 9, and the letters A-F, a-f. Hexadecimal letters may be specified in either upper or lower case.

The following table shows some sample valid and invalid hexadecimal sequences.

| Sequence | Valid | Reason |
|---|---|---|
| '4FFF'X | YES | Correct form |
| '4FFFX' | NO | Incorrect placement of apostrophes |
| '0ffa'x | YES | Correct form |
| 0fffX | NO | Missing apostrophes |

**F77 Compatibility Notes:**

Hexadecimal constants have a different form in F77 compatibility mode. In this mode, rather than adding an 'X' suffix to a string of hexadecimal digits, an 'X' or 'Z' is prepended to the string. The following four examples show legal hexadecimal constant specifications when F77 compatibility mode is selected.

```
x'1234'
X'5678'
z'9abc'
Z'DEF0'
```

Please refer to your User's Guide for details on compiler options and default modes.

### 5.1.3. Radix-50 Constants

Radix-50 encoding allows character data to be represented in packed form, storing up to 3 characters in 16 bits. Normal character storage allows a maximum of 2 characters per 16 bit word.

**Syntax:** nRccccccccccc

where n is a value from 1 to 12, specifying the number of characters to the right of the letter 'R', and cccccccccccc represents an ASCII character string of n characters that will be converted to Radix-50 notation.

Radix-50 encoding is accomplished by assigning a subset of the ASCII character set Radix-50 values, then calculating a single 16-bit number using the formula

```
((rad1 * 40 + rad2) * 40 + rad3)
```

where rad1, rad2, and rad3 are Radix-50 values for the selected ASCII characters from left to right. For example, the Radix-50 constant 3RABC will be stored internally as decimal 1683.

The ASCII character subset and Radix-50 equivalents are listed in the following table. Both Radix-50 and ASCII codes are shown in octal and decimal values. Note that the Radix-50 value 29 (octal 35) is not assigned. Note that upper-case and lower-case alphabetic characters are equivalent in Radix-50.

| Character | ASCII Octal | Radix-50 Octal | ASCII Decimal | Radix-50 Decimal |
|---|---|---|---|---|
| Space | 40 | 0 | 32 | 0 |
| A-Z | 101-132 | 1-32 | 65-90 | 1-26 |
| a-z | 041-062 | 1-32 | 33-50 | 1-26 |
| $ | 44 | 33 | 36 | 27 |
| . | 56 | 34 | 46 | 28 |
| unassigned | | 35 | | 29 |
| 0-9 | 60-71 | 36-47 | 48-57 | 30-39 |

Radix-50 constants may only be used in DATA statements. The data type of the variable will determine the total number of bytes which may be stored for the specified constant. If the constant value is too large to be stored in the selected data type, the rightmost bytes are truncated. If the constant evaluates to a value smaller than the maximum storage for the selected data type, the constant is blank-filled from the right.

The following table shows some sample valid and invalid Radix-50 sequences.

| Sequence | Valid | Reason |
|---|---|---|
| 5RHELLO | YES | Correct form |
| 14R12345678901234 | NO | More than 12 characters specified |
| 8RHI THERE | YES | Correct (note that embedded spaces are allowed) |
| 6RI_TST | NO | Underscore is not a valid Radix-50 character |

**F77 Compatibility Notes:**

Radix-50 constants are not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 5.2. Variables

A variable consists of a symbolic name and an associated value. A variable can represent a numeric, logical, or character value in a program. Unlike the symbolic name for a constant, the symbolic name for a variable can assume many different values during the execution of a program.

Consider, again, the simple program that calculates the area of a circle using the standard formula $A = \pi r2$ where r is the radius of the circle, $\pi$ is equal to 3.1415926, and A is the area. The value of $\pi$ remains unchanged and is represented with the real constant 3.1415926. However, the value of r varies for circles of different sizes. If you use a constant to represent r, the program would be limited to circles of one size. You would have to rewrite the program every time you wanted to calculate the area of a circle with a different radius. Instead, let the radius be a variable. The program can assign a new value to the radius variable for each new area calculation.

A variable must have an assigned value before you can reference the variable name in the program. When a reference to a variable name executes, the program uses the value that is currently assigned to the variable at that point in the execution of the program.

Variables assume values through assignment statements. For example, a simple integer variable named int assumes the value 14 in the following arithmetic assignment statement.

```
int = 12 + 2
```

The variable int can assume values other than 14 during the execution of a program. Consider a second arithmetic assignment statement.

```
int = int + 1
```

Assuming that this assignment statement executes after the previous assignment statement in a program, the value of the variable int changes from 14 to 15. Refer to the "Assignment Statements" chapter for additional information.

You can specify an initial value for a variable in a program using the DATA statement. Refer to the "DATA Statement" chapter and the "BLOCK DATA" section for more information.

## 5.3. Arrays

An array is a sequence of variables that represent numeric, logical, or character values in a program. Each variable in the sequence is called an array element. Arrays can organize large amounts of data, particularly in complex programs, because they enable you to treat a group of related variables as a unit instead of as separate entities. Using arrays, you can reference a group of similar or related values with a single symbolic name.

Arrays consist of one or more dimensions. Dimensions enable you to organize data according to various criteria defined in the context of your program. For example, a program designed to analyze political opinion poll information might organize data according to three different criteria: the voter's age, precinct, and political party affiliation. Such a program could use a three dimensional array. Each element can have a data type comprised of several bytes.

### 5.3.1. Array Declarators and Subscripts

To declare an array in a program you must use an array declarator in a DIMENSION, COMMON, or type statement. An array declarator specifies a symbolic name to identify the array and a number of dimension declarators. The form for an array declarator is shown below. The abbreviation "dim" stands for dimension declarator.

```
symbolic-name (dim [,dim ]...)
```

Dimension declarators specify the number of elements in each array dimension that you declare. You set the number of elements with a lower- and upper-bound value. The lower- and upper-bound values are called dimension bounds. The form for a dimension declarator is as follows.

```
[lower-bound:] upper-bound
```

Dimension bounds can be arithmetic constant or variable expressions that evaluate to integers. The optional lower-bound value can be negative, zero, or positive. If you do not specify a lower-bound, a value of 1 is implied. The upper-bound value can be negative, zero, positive, or an asterisk indicating an assumed-size array declarator. The use of a variable expression as a dimension bound value constitutes an adjustable array declarator. Adjustable and assumed-size array declarators are described later in this section.

The number of dimension declarators that you specify in an array declarator determines the number of dimensions for the array. An array in FORTRAN can have a maximum of seven dimensions.

The size of an array is equal to the number of elements in the array. The number of elements is equal to the product of the dimension sizes specified in the array declarator.

The following example is an array declarator that declares an array with the symbolic name DISTANCES and one dimension declarator. DISTANCES is a one-dimensional array with a lower-bound of 10 and an upper-bound of 20. DISTANCES consists of 11 elements.

```
DISTANCES (10:20)
```

The next example is an array declarator that declares an array with the symbolic name AMPS and three dimension declarators. The dimension declarators do not include lower-bound specifications. Therefore, each dimension has an implied lower-bound of 1. AMPS is a three-dimensional array. The first two dimensions have an upper-bound of 8 and the third dimension has an upper-bound of 2. AMPS consists of 128 elements.

```
AMPS (8,8,2)
```

Each element in an array is a variable that can assume different values during program execution. A program can access the values in the array by referencing the element name in an expression. To reference an element name, you specify the name of the array followed by a subscript. A subscript consists of one or more arithmetic constant expressions enclosed in parentheses. Subscript expressions must evaluate to integers. An array element name has the following form. The abbreviation "sub-exp" stands for subscript expression.

```
symbolic-name (sub-exp [,sub-exp]...)
```

The number of subscript expressions that you specify in an element name reference must match the number of dimension declarators specified in the array declarator.

In certain situations, you can specify an array name without a subscript to reference the entire array. You can reference an array name without a subscript in the following FORTRAN statements.

- type Statements
- COMMON Statement
- DATA Statement
- EQUIVALENCE Statement
- FUNCTION Statement
- SUBROUTINE Statement
- ENTRY Statement
- SAVE Statement
- Input/Output Statements

You can also use unsubscripted array names as actual arguments in a reference to a subroutine or external function.

### 5.3.2. One-dimensional Arrays

To demonstrate a one dimensional array, consider a simple program designed to calculate the average high meteorological temperature over a one week period. The program must store seven temperature readings, one for each day of the week, then calculate the average high. To store the temperatures, we declare an array using an array declarator. For this example, we declare a one-dimensional integer array with the symbolic name TEMPS. To declare the integer data type we use the array declarator in an INTEGER type statement.

```
INTEGER TEMPS(7)
```

The number 7 enclosed in parentheses is the dimension declarator specifying an upper-bound of 7 elements. Notice that no lower-bound for the dimension is specified. Therefore, a lower-bound of 1 is implied. Refer to "Array Declarators and Subscripts" in this section for additional information.

Each element in TEMPS is a variable numbered 1 through 7 that can assume an assigned temperature value. You can assign initial values to array elements with the DATA statement. During program execution, you can assign values to array elements with assignment or input statements. The following figure represents the structure of array TEMPS with assigned values.

| | SUN | MON | TUE | WED | THU | FRI | SAT |
|---|---|---|---|---|---|---|---|
| TEMPS | 63 | 60 | 55 | 68 | 72 | 73 | 64 |
| | (1) | (2) | (3) | (4) | (5) | (6) | (7) |

The program can access any temperature value in the TEMPS array for a calculation by referencing the array element name. To reference an element name you specify the array name followed by a subscript. For example, TEMPS(3) refers to the third element in TEMPS, which has been assigned the value 55 degrees. The number 3 is a subscript expression. The number 3 including the parentheses constitutes the entire subscript. Remember, the number of subscript expressions that you specify in an element name reference must match the number of dimensions specified in the array declarator.

Each array element in TEMPS is a variable that can take on new values. Therefore, at the end of each week you can assign new temperature readings to the corresponding array elements and execute the program to calculate the average high for the new week.

### 5.3.3. Multi-dimensional Arrays

One-dimensional arrays organize data in linear form according to one criterion defined in the context of the program. In our hypothetical temperature program, the elements of array TEMPS correspond, one to one, with the days of the week. Multi-dimensional arrays enable you to organize data according to more than one criterion. For example, suppose we want to expand the temperature program to calculate the average body temperature for a medical patient over a one week period using three temperature readings taken each day instead of one. Now, we have two criteria upon which to organize the temperature values; the day of the week and the time of the day.

The new program must store twenty-one temperature readings, three for each day of the week, then calculate the average. To store the temperature values, we can declare a two-dimensional array using an array declarator. For this example, we declare the array with the symbolic name TEMPS2. The data type for TEMPS2, however, must be type REAL because body temperatures for medical patients must be accurate to a tenth of a degree. Therefore, to declare the REAL data type we use the array declarator in a REAL type statement.

```
REAL TEMPS2(7,3)
```

Notice that two dimension declarators are enclosed in parentheses following the symbolic name TEMPS2. The number 7 is the dimension declarator for the first dimension specifying an upper bound of seven elements. The number 3 is the dimension declarator for the second dimension specifying an upper bound of three elements. Both dimensions have an implied lower bound of 1.

You can assign initial values to the elements in TEMPS2 with the DATA statement. During program execution, you can assign values to array elements with assignment or input statements. The following figure represents the structure of array TEMPS2 with assigned values.

| TEMPS2 | SUN | MON | TUE | WED | THU | FRI | SAT | |
|---|---|---|---|---|---|---|---|---|
| 6 A.M. | 103.4 | 103.4 | 103.2 | 103.1 | 101.1 | 102.9 | 103.3 | (1) |
| 2 P.M. | 103.0 | 102.3 | 100.4 | 99.7 | 99.1 | 98.8 | 98.7 | (2) |
| 10 P.M. | 99.2 | 100.2 | 101.6 | 102.9 | 100.7 | 99.2 | 98.6 | (3) |
| | (1) | (2) | (3) | (4) | (5) | (6) | (7) | |

Two-dimensional arrays organize data in grid form. The program can access any temperature value in the TEMPS2 array for a calculation by referencing the array element. To reference an element in a two-dimensional array, you must specify two subscript expressions after the array name. The subscript expressions serve as coordinates to locate values in the grid. For example, TEMPS2(4,2) refers to the fourth element in the first dimension and the second element in the second dimension. The reference points to the value 99.7 degrees taken at 2 P.M. on Wednesday.

### 5.3.4. Adjustable Array Declarators

All of the arrays in the previous examples are declared using constant array declarators. In a constant array declarator, each dimension bound is an integer constant or integer constant expression. In certain cases, you can specify dimension bounds using integer variables and integer variable expressions.

The use of a variable in a dimension bound specification constitutes an adjustable array declarator. Adjustable array declarators enable program units to pass various sized arrays as arguments. The adjustable array declarator serves as a dummy array in a subroutine or function procedure. The reference to the procedure contains the actual array size specifications. The adjustable array declarator must specify the same number of dimensions as the actual argument array.

The following example shows an adjustable array declarator named ADJ used in a DIMENSION statement. The DIMENSION statement is in a subroutine named TEST. Note that the adjustable array declarator name, ADJ, and the two variables used as dimension bound values, M and N, are dummy arguments for the TEST subroutine. M and N are the adjustable dimensions.

```
SUBROUTINE TEST(ADJ,M,N)
 .
 .
DIMENSION ADJ(M,N)
 .
 .
END
```

The following program unit example declares two actual arrays, ACT1 and ACT2, and contains two calls to the TEST subroutine defined above. Each call passes a different array to TEST for processing. The first CALL statement passes the array name ACT1 and the two dimension declarators 5 and 10 as actual arguments. The second CALL statement passes the array name ACT2 and the dimension declarators 25 and 50 as actual arguments.

```
DIMENSION ACT1(5,10)
DIMENSION ACT2(25,50)
 ...
CALL TEST(ACT1,5,10)
 ...
CALL TEST(ACT2,25,50)
 ...
END
```

ACT1 and ACT2 are different sized arrays, but the adjustable array declarator, ADJ, enables the TEST subroutine to process both arrays one at a time.

### 5.3.5. Assumed-size Array Declarators

Assumed-sized array declarators, like adjustable array declarators, serve as dummy arrays in a function or subroutine procedure. An assumed-size array declarator uses an asterisk (*) as an upper dimension bound for the last dimension declared in the array. The actual upper dimension bound passes to the procedure from the procedure reference. Dimension bound values in an assumed-size array declarator other than the upper bound of the last dimension can be integer constant or variable expressions.

The following example shows an assumed-size array declarator named ASM used in a DIMENSION statement. The DIMENSION statement is in a function named CALC. In this example, the lower dimension bounds for both dimensions are integer constants. The upper bound for the first dimension is a variable, W. The upper bound for the second dimension is an asterisk. ASM serves as a dummy array within the CALC function. Note that the name ASM and the variable W used as an upper dimension bound for the first dimension are dummy arguments for the CALC function.

```
FUNCTION CALC(ASM,W)
  .
  .
DIMENSION ASM(1:W,1:*)
  .
  .
END
```

The following program unit example contains a reference to the CALC function defined above. The reference passes an actual array named ACT for processing. Note that the reference to CALC passes the array name ACT and the value 10 as actual arguments. The actual upper bound for the second dimension, 30, does not pass as an argument.

```
DIMENSION ACT(10,30)
  .
  .
VALUE \(eq CALC(ACT,10)
  .
  .
END
```

The assumed-size array declarator, ASM, assumes the size of the actual array, ACT, passed to the CALC function in the function reference.

## 5.4. Substrings

A substring is a contiguous portion of the space that a character variable or character array element represents. Substring references enable you to manipulate segments of character strings in a program. A substring has a character data type.

Substring references have two forms: one for a character variable and one for a character array element. A substring reference for a character variable uses the following form.

```
variable-name([1st-expression]:[2nd-expression])
```

The character positions that a character variable represents are numbered from left to right, beginning with 1. The 1st-expression specifies the first or leftmost character of the substring that you want to reference. The 2nd-expression specifies the last or rightmost character of the substring that you want to reference. For example, the following substring reference specifies character positions three through seven in the character variable "materials".

```
materials(3:7)
```

The variable "materials" can take on a variety of values during program execution. However, the substring reference always specifies character positions three through seven regardless of the value of the variable at any given time. A substring reference for a character array element uses the following form:

```
array name (sub[, sub]...) ([1st-expression]:[2nd-expression])
```

Like a substring reference for a character variable, the character positions that a character array element represents are numbered from left to right, beginning with 1. The 1st-expression in the substring reference specifies the first or leftmost character of the substring that you want to reference. The 2nd-expression specifies the last or rightmost character that you want to reference. The abbreviation sub stands for subscript expression. You can specify any number of subscript expressions in a substring reference. For example, the following substring reference specifies character positions 5 through 10 of an element in a three dimensional character array named "products".

```
products (4,4,12) (5:10)
```

For both character variable and character array element references,

the 1st-expression must be greater than or equal to 1 and less than or equal to the 2nd-expression. The 2nd-expression must be less than or equal to the length of the variable or array element. Expressions that do not evaluate to integers convert to integers.

If you omit the 1st-expression, a leftmost character position of 1 is implied. If you omit the 2nd-expression, a rightmost character position equal to the length of the variable or array element is implied. To omit both expressions implies a reference to all the character positions in the variable or array element. If you omit both expressions, you must still specify the colon enclosed in parentheses. The following examples show some different substring references.

```
versions(2:8)
items(5:)
fourth(3,9)(:11)
sysform(:)
```

# Chapter 6
# Expressions

An expression is a sequence of characters that specifies instructions for calculating a value. An expression can be a single basic data item, such as a constant or variable, or a combination of data items and operators. Operators are special characters that specify computations to perform using values specified in the expression. The values that an operator processes are called operands. All expressions represent, or evaluate to a single value. There are four kinds of expressions in FORTRAN:

- arithmetic
- character
- relational
- logical

## 6.1. Arithmetic Expressions

Arithmetic expressions represent numeric values. An arithmetic expression uses a special set of arithmetic operators, arithmetic operands, and parentheses to control the evaluation order of the operations specified in the expression. There are five arithmetic operators in FORTRAN as shown in the following table.

| Operator | Purpose |
|---|---|
| + | Addition |
| – | Subtraction or unary negation |
| * | Multiplication |
| / | Division |
| ** | Exponentiation |

The *, /, and ** operators work with two operands and are called binary operators. The + and – operators can work as binary operators or as unary operators that work on a single operand. The following table defines the interpretation of expressions using each of the arithmetic operators.

**Interpretation of Arithmetic Operators**

| Example | Interpretation |
|---|---|
| OP1 + OP2 | Add OP1 and OP2. |
| +OP1 | Identify OP1 as positive. |
| OP1 – OP2 | Subtract OP2 from OP1. |
| –OP1 | Identify OP1 as negative. |
| OP1 * OP2 | Multiply OP1 and OP2. |
| OP1 / OP2 | Divide OP1 by OP2. |
| OP1 ** OP2 | Raise OP1 to the power OP2. |

There is a precedence among the arithmetic operators that determines the order in which operands are combined within an arithmetic expression that contains two or more operators. The precedence among the arithmetic operators follows the standard rules of algebra and is shown in the following table, with the lowest number having the highest precedence.

**Precedence Among Arithmetic Operators**

| Operator | Precedence |
|---|---|
| ** | 1 |
| * and / | 2 |
| + and – | 3 |

When an expression contains two or more operators of equal precedence, such as * and /, FORTRAN evaluates the operations algebraically from left to right. Exponentiation, however, is evaluated from right to left. For example, consider the following expression.

```
OP1 ** OP2 ** OP3
```

FORTRAN first calculates OP2 raised to the power indicated by OP3, then calculates OP1 raised to the power indicated by the value that results from the first calculation.

You can use parentheses to control the evaluation order of the operations specified in an arithmetic expression. The portions of an expression you enclose in parentheses are evaluated first. The use of parentheses supercedes the precedence among arithmetic operators. The following examples demonstrate the evaluation order of operations within arithmetic expressions according to operator precedence and the use of parentheses.

```
25 + 15 - 10 + 12 = 42          (25 + 15) - (10 + 12) = 18
   ↑    ↑    ↑                     ↑      ↑      ↑
  1st  2nd  3rd                    1st    3rd    2nd

3 + 4 * 3 - 5 = 10              3 + 4 * (3 - 5) = -5
  ↑   ↑   ↑                       ↑   ↑   ↑
 2nd 1st 3rd                     3rd 2nd 1st

100 - 10 ** 2 * 3 = -200          (100 - 10 ** 2) * 3 = 0
    ↑     ↑    ↑                       ↑    ↑     ↑
   3rd   1st  2nd                     2nd  1st   3rd

11 + 3 * 2 - 10 / 2 = 12        ((11 + 3) * 2 - 10) / 2 = 9
   ↑   ↑   ↑    ↑                    ↑    ↑   ↑     ↑
  3rd 1st 4th 2nd                   1st  2nd 3rd   4th]
```

In the preceding examples, all the operands are unsigned numeric constants. You can, however, use operands in a variety of forms within an arithmetic expression. An arithmetic operand can be any of the following entities.

- an unsigned numeric constant
- a symbolic name of an unsigned numeric constant
- a numeric variable reference
- a numeric array element reference
- an arithmetic function reference
- an arithmetic expression enclosed in parentheses

The data type of an arithmetic expression is determined by the data types of the operands specified in the expression. An arithmetic expression that contains operands of one data type evaluates to a value of the same data type. An arithmetic expression that contains operands of two or more different data types evaluates to a value of the highest ranking type in the expression. Operands of low ranking data types automatically convert to the higher ranking types. The one exception to this strict hierarchy of data type conversions is that the combination of a REAL*8 (DOUBLE PRECISION) value with a COMPLEX*8 (COMPLEX) value causes both operands to be converted to COMPLEX*16 (DOUBLE COMPLEX). The following table defines the hierarchy of data types for conversion within expressions.

**Arithmetic Data Type Conversion Hierarchy**

| Data Type | Rank |
|---|---|
| COMPLEX*16 | 1 (Highest) |
| COMPLEX*8 (COMPLEX) | 2 |
| REAL*8 (DOUBLE PRECISION) | 3 |
| REAL*4 (REAL) | 4 |
| INTEGER*4 | 5 |
| INTEGER*2 (INTEGER) | 6 |
| INTEGER*1 | 7 |
| LOGICAL | 8 (Lowest) |

According to the hierarchy for arithmetic data type conversion, an arithmetic expression that consists of real and complex operands evaluates to a complex value. An arithmetic expression that consists of INTEGER*2 and INTEGER*4 operands evaluates to an INTEGER*4 value, and so on. Here are some rules that apply to the conversion of data types within arithmetic expressions.

- In an expression that contains integer and real operands, the integer operands first receive a fractional component of 0 in the conversion to real. Then, FORTRAN evaluates the expression using real arithmetic. Consider the expression (10/5)*3.14. Note that FORTRAN first evaluates the integer division, (10/5), then converts the result of the division to real.
- To convert an operand of one REAL data type to a REAL data type with a higher precision, FORTRAN uses the existing operand as the most significant portion of the higher precision data item and the least significant part of the data item becomes zero. FORTRAN then evaluates the expression using the higher precision arithmetic.
- In an expression that contains complex and integer operands, the integer operands convert to real as described above. The converted real operand then serves as the real part of a complex number and the imaginary part becomes zero. FORTRAN then evaluates the expression using complex arithmetic. The expression evaluates to a complex value.
- Any fractional component that results from a division of integers is truncated, not rounded. For example, the expression (1/2 + 1/2) is equal to 0, not 1. The expression (12/5) is equal to 2. The fractional components are truncated.

## 6.2. Character Expressions

Character expressions enable you to manipulate character strings. A character expression uses character operands and a special character operator. All character expressions evaluate to a single character string value.

The character operator consists of two slashes, //, and is called the concatenation operator. The concatenation operator simply joins two character operands together. For example, the following character expression evaluates to the character string value 'FORTRAN'.

```
'FOR' // 'TRAN'
```

In the preceding example, all the operands are character constants. You can, however, use operands in a variety of forms within a character expression. A character operand can be any of the following entities.

- a character constant
- a symbolic name of a character constant
- a character variable reference
- a character array element reference
- a character substring reference
- a character expression enclosed in parentheses
- a character function reference

The use of parentheses does not affect the value of a character expression. For example, the following character expressions are equivalent.

```
'PRESS' // 'ANY KEY' // 'TO CONTINUE'
'PRESS' // 'ANY KEY') // 'TO CONTINUE'
'PRESS' // ('ANY KEY' // 'TO CONTINUE')
```

The length of a character expression equals the sum of the lengths of the individual operands. The length value includes any spaces that are parts of an operand. For example, the following expression has a length of 19.

```
'ENTER' // 'b/YOURb/PASSWORD'
```

Parentheses are not parts of an operand and are not included in the length value.

## 6.3. Relational Expressions

A relational expression compares the values of two operands using a special set of relational operators. A single relational expression can compare only two arithmetic expressions or two character expressions. A relational expression cannot compare an arithmetic expression with a character expression. Relational expressions evaluate to a logical value, either true or false, depending on the comparison condition.

The relational operators test for certain relationships that exist between two operands. There are five relational operators in FORTRAN as shown in the following table. Note that the periods that delimit each relational operator are required for use in an expression.

**Relational Operators**

| Operator | Meaning |
|---|---|
| .LT. | less than |
| .LE. | less than or equal to |
| .EQ. | equal to |
| .NE. | not equal to |
| .GT. | greater than |
| .GE. | greater than or equal to |

In an arithmetic relational expression, FORTRAN first evaluates the arithmetic operands, then compares the resulting values to determine if the relationship specified by the relational operator exists. Consider the following arithmetic expression.

```
100-50 .GT.  100/50
  ↑           ↑
  50          2
```

FORTRAN first evaluates the arithmetic expressions on either side of the .GT. operator. Following this first evaluation, the relational expression states that 50 is greater than 2. FORTRAN then evaluates the validity of the expression. Since 50 is greater than 2, the value of the relational expression is true.

If we change the operator in the preceding example to .LT. (less than), the value of the relational expression becomes false, because 50 is not less than 2. You can use parentheses within the arithmetic operands of a relational expression to alter the evaluation order.

In a character relational expression, FORTRAN first evaluates the character operands, then compares the resulting values to determine if the relationship specified by the relational operator exists. Consider the following character expression.

```
'APP' // 'LE' .LT.  'AP' // 'RICOT'
```

FORTRAN first evaluates the character expressions on either side of the .LT. operator. Following this first evaluation, the relational expression states that the string 'APPLE' is less than the string 'APRICOT'.

For character relational expressions, operands are evaluated according to the ASCII character collating sequence. The length of the character operands is not significant for comparison. If the two character operands have different lengths, the shorter operand is padded on the right with blank characters until the two strings are equal. FORTRAN then compares the two strings, a character at a time according to the ASCII collating sequence. Under ASCII conventions, all characters correspond to numeric values that determine a hierarchy among the characters. Refer to the "ASCII and Hexadecimal Conversion Table" for a listing of the collating sequence.

In the preceding example, the string 'APPLE' has a length of 5 and the string 'APRICOT' has a length of 7. Therefore, FORTRAN pads 'APPLE' on the right with two blank characters to make the strings equal in length. Then, FORTRAN compares the two strings a character at a time. The first two letters in both strings, AP, are equal. The third letter in each string, however, is different. 'APPLE' has a P and 'APRICOT' has an R. According to the collating sequence, P is less than R. Therefore, the string 'APPLE' is less than the string 'APRICOT' and the relational expression is true.

FORTRAN can compare complex expressions only with the .EQ. and .NE. operators. Two complex values are considered equal only if their corresponding real and imaginary parts are both equal.

A relational operator can compare two numeric expressions of different data types. FORTRAN converts the value of the expression with the lower ranked data type to the higher ranked data type before making the comparison. When a REAL*8 and a COMPLEX*8 value are compared, the values are first converted to the type COMPLEX*16 before making the comparison.

All the FORTRAN relational operators have equal precedence. However, arithmetic and character operators have a higher precedence than relational operators. Therefore, FORTRAN evaluates arithmetic and character operators before relational operators.

## 6.4. Logical Expressions

Logical expressions compare logical values (true and false) using a special set of logical operators. The operands in a logical expression evaluate to logical values. You can use parentheses to control the evaluation order of the operations specified in a logical expression. All logical expressions evaluate to a single logical value. There are six logical operators in FORTRAN as shown in the following table.

**Logical Operators**

| Operator | Meaning |
|---|---|
| .NOT. | logical negation |
| .AND. | logical conjunction |
| .OR. | logical inclusive disjunction |
| .EQV. | logical equivalence |
| .NEQV. | logical nonequivalence |
| .XOR. | equivalent to .NEQV. |

The interpretation of expressions formed using each of the logical operators is shown in the following table. OP1 denotes an operand for a unary operator or an operand to the left of a binary operator. OP2 denotes an operand to the right of a binary operator.

**Interpretation of Logical Operators**

| Example | Interpretation |
|---|---|
| OP1 .AND. OP2 | The expression is true only if both OP1 and OP2 are true. |
| OP1 .OR. OP2 | The expression is true if either OP1 or OP2, or both, is true. |
| OP1 .EQV. OP2 | The expression is true only if both OP1 and OP2 have the same logical value: either true or false. |
| OP1 .NEQV. OP2 | The expression is true if OP1 is true and OP2 is false or if OP2 is true and OP1 is false. The expression is false if both operands have the same logical value. |
| OP1 .XOR. OP2 | The .XOR. operator is equivalent to .NEQV. and has been included for compatibility. It is not available in F77 compatibility mode. |
| .NOT. OP1 | The expression is true only if OP1 is false. |

There is a precedence among the logical operators that determines the order in which operands are compared within a logical expression that contains two or more operators. The following table shows the precedence among the logical operators, with the lowest number having the highest precedence.

**Precedence Among Logical Operators**

| Operator | Precedence |
|---|---|
| .NOT. | 1 |
| .AND. | 2 |
| OR. | 3 |
| .EQV., .NEQV., and .XOR. | 4 |

When an expression contains two or more operators of equal precedence, such as .EQV. and .NEQV., FORTRAN evaluates the operations algebraically from left to right. You can use parentheses to control the evaluation order of the operations specified in a logical expression. The following examples demonstrate the evaluation order of operations within logical expressions according to operator precedence and the use of parentheses.

```
6*3+4 .LT.  25 .AND.  4 .LE.  8/2 (evaluates to true)
6*(3+4) .LT 25 .AND.  4 .LE.  8/2 (evaluates to false)
```

Two logical operators cannot appear consecutively within an expression unless the second operator is .NOT. The following example is a valid expression.

```
3.14159 .LE.  10 .AND.  .NOT.  (10/5 .EQ.  3) (evaluates to true)
```

In the preceding examples, all the operands are arithmetic relational expressions. You can, however, use operands in a variety of forms within a logical expression. A logical operand can be any of the following entities.

- an arithmetic relational expression
- a character relational expression
- a logical constant (.TRUE. or .FALSE.)
- a symbolic name of a logical constant
- a logical variable reference
- a logical array element reference
- a logical function reference

### 6.4.1. Logical Operations on Integers

Logical operations may be performed on integer operands, and are carried out bit by bit on the internal values. Logical and integer operands may be used in combination, in which case the logical operand is first converted to an integer value, then the logical operation is performed.

Similarly, logical variables and/or expressions may be used in an integer context within an expression. The logical expression is converted to an integer value before the overall expression is evaluated.

**Example:**

| Value of A | Value of B | Operation | Result |
|---|---|---|---|
| 7 | 2 | C = (A .AND. B) | C = 5 |
| 0 | 255 | C = ( .NOT. B) | C = 0 |
| 43 | -6 | C = (A .OR. B) | C = -5 |

**F77 Compatibility Notes:**

Logical operations on INTEGERs are not supported in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 6.5. FORTRAN Operator Precedence

There is a precedence among all four kinds of FORTRAN operators that determines the order in which operands are combined in expressions that contain more that one kind of operator. The following table shows the precedence among the four kinds of operators, with the lowest number having the highest precedence.

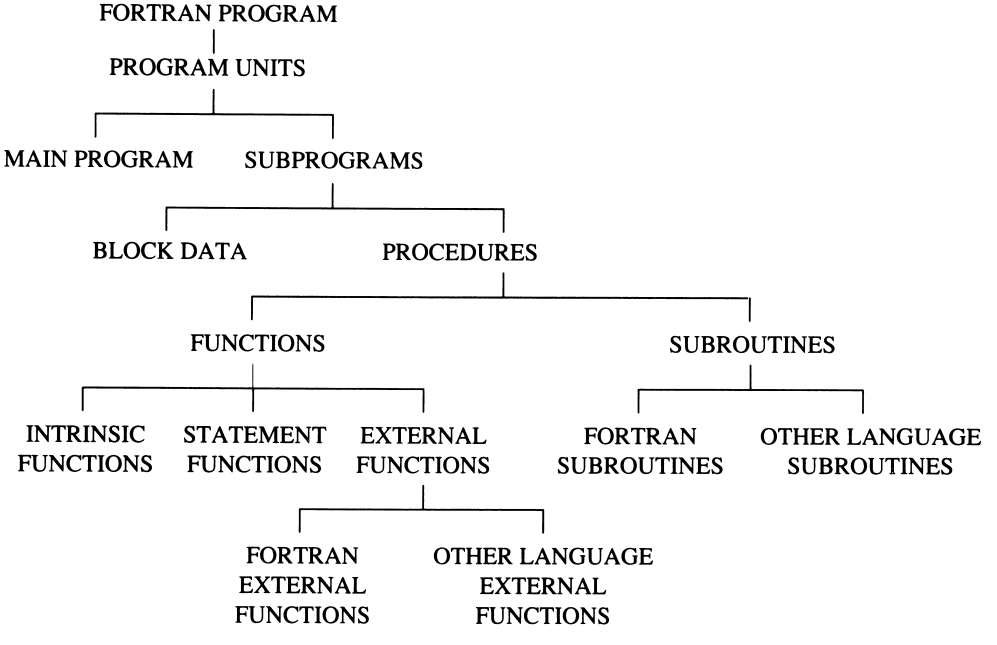
**FORTRAN Operator Precedence**

| Operator | Precedence |
|---|---|
| Arithmetic | 1 |
| Character | 2 |
| Relational | 3 |
| Logical | 4 |

Remember, you can use parentheses to control the evaluation order of the operations specified in an expression. The use of parentheses supercedes operator precedence.

# Chapter 7
# FORTRAN Program Structure

An executable FORTRAN program consists of one or more program units. A program unit is a logical, self-contained sequence of statements and optional comment lines that forms a discrete part of a larger program. There are two kinds of program unit: main programs and subprograms. This section explains the main program, the different classes of subprogram, and the relationships among program units that combine to form FORTRAN executable programs. The diagram below illustrates FORTRAN program structure.




FORTRAN Program Structure

## 7.1. Main Program

The main program serves as the center or base of all processing activity in an executable program. The main program is the program unit that receives control to begin execution. During execution, the main program can invoke a variety of subprograms that perform different tasks. Control returns to the main program to terminate execution except when a STOP statement executes. See the END and STOP statements for more information on program termination.

An executable FORTRAN program can consist of a main program with no subprograms. Each executable program can have only one program unit defined as the main program. Subprograms cannot call or reference the main program and the main program cannot call or reference itself.

Use the PROGRAM statement to define a program unit as a main program. The PROGRAM statement is not required, but if used, it must be the first statement of the main program. The PROGRAM statement specifies a symbolic name for the main program as shown in the following syntax specification.

```
PROGRAM symbolic-name
```

The main program can contain any of the FORTRAN statements except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY and RETURN statements. A STOP or END statement terminates execution of the program. The SAVE statement has no effect within the main program.

## 7.2. Subprograms

The term subprogram covers two kinds of program unit: block data subprograms and procedures. Block data subprograms initialize variables and array elements, and cannot contain executable statements. Procedures contain executable statements that define a specific computing operation. Subprograms cannot call or reference the main program. An executable FORTRAN program must contain only one main program but can contain any number of subprograms.

## 7.3. Block Data

A block data subprogram enables you to specify initial values for variables and array elements that are listed in named common blocks. Common blocks are areas of storage containing data that different program units can share. Any program unit that contains a definition of a given common block can use the data in that block. Use the COMMON statement to define common blocks.

Common blocks can be either named or unnamed. However, only variables and

array elements in named common blocks can be initialized in a block data subprogram. You can initialize data from several named common blocks in one block data subprogram. Refer to the COMMON statement for more information on common storage area management.

You identify block data subprograms with the BLOCK DATA statement. The BLOCK DATA statement has the following form.

```
BLOCK DATA [symbolic-name]
```

The symbolic-name is a global reference for the subprogram. The name is optional, but if used it must be unique. There cannot be more than one unnamed block data subprogram in one executable program. The BLOCK DATA statement must be the first statement in a block data subprogram.

A block data subprogram can contain only certain specification statements: COMMON, DIMENSION, DATA, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and type statements. You can also include comment lines. The last statement in a block data subprogram must be the END statement. An executable FORTRAN program can contain any number of block data subprograms. Block data subprograms have the following form.

```
BLOCK DATA [symbolic-name]
 .
 .  specification statements and comments
 .

END
```

The following block data subprogram example defines three named common blocks for a hypothetical program that calculates weather information: time, location, and conditions. The example specifies a data type for each variable in the named common blocks and declares initial values for some of the variables.

```
        BLOCK DATA weather
C Define named common areas
        COMMON /time/day,hour,min /location/zone,altitude
        COMMON /conditions/temp,humidity,pressure,wind,rain
C Declare data types for the variables
        INTEGER day,hour,min,zone,altitude,temp,humidity,wind
        REAL pressure
        LOGICAL rain
C Declare initial values for some of the variables
        DATA day/31/, hour/24/, zone/7/, altitude/5285/
        DATA temp/65/, pressure/29.92/, rain/.true./
        END
```

Notice that you must specify all the variables in the named common blocks using specification statements even if you do not declare initial values for all the variables with the DATA statement.

## 7.4. Procedures

A procedure is a subprogram that performs a specific task within an executable program. Unlike block data subprograms, procedures contain the executable statements that define the purpose of the program. Procedures structure programs into a series of routines that can execute repeatedly, in any order, as required to accomplish a larger task. This process of breaking up a large programming task into a series of smaller, logically individual procedures is the fundamental principle of structured programming. An executable FORTRAN program can contain any number of procedures.

A procedure receives execution control through a call or reference. Procedures can receive control from the main program or from another procedure. However, procedures cannot call or reference the main program. Procedures can share values through the use of arguments and common blocks.

There are two classes of procedure within FORTRAN: subroutines and functions. Subroutines and functions differ primarily in the method by which they are invoked during execution and in the specific result that they produce. Functions are further classified into three different categories: external functions, statement functions, and intrinsic functions. Subroutines and external functions are collectively referred to as external procedures.

### 7.4.1. Procedure Arguments

Arguments supply the values that a procedure requires to produce the desired result. A program unit, such as the main program, can invoke a procedure to perform a specific task using arguments to pass the values that the procedure needs to complete the task. Both subroutine and function procedures can change the values of the arguments during execution. Therefore, values that the procedure produces can return to the invoking program unit via the arguments. Arguments are sometimes called parameters.

Two kinds of argument must be distinguished: dummy or formal arguments and actual or calling arguments. Dummy arguments are used in the procedure definition to reserve a place and declare a data type for actual values that the procedure requires to produce the desired result. Actual arguments are used in the procedure call or reference and are substituted for the dummy arguments during the actual procedure execution.

Dummy arguments used in the procedure definition must correspond in number, order, and data type with the actual arguments in the procedure call or reference. Depending on the kind of procedure, a dummy argument can be a variable name, an array name, a dummy procedure name, or an alternate return specifier.

A variable name that serves as a dummy argument can only be associated with an actual argument that is a variable, a constant, a symbolic name of a constant, a function reference, an array element, a substring, or an expression.

An array name that serves as a dummy argument can only be associated with an actual argument that is an array, array element, or array element substring of matching data type.

## 7.5. Subroutines

A subroutine is an external procedure that performs a specific task within an executable FORTRAN program. An external procedure is a distinct program unit that is defined outside of the program unit that invokes it. You can write external procedures using a programming language other than FORTRAN, such as C or assembly language. Refer to the "Interfacing FORTRAN and C" chapter in your FORTRAN User's Manual for additional system specific information.

Use the SUBROUTINE statement to define a program unit as a subroutine procedure. The SUBROUTINE statement must be the first statement in the subroutine. The SUBROUTINE statement specifies a symbolic name for the subroutine and a list of dummy arguments the subroutine requires to produce the desired result. Remember, dummy arguments reserve a place and specify a data type for the actual arguments supplied in the subroutine call. The following syntax specification shows the general format for a SUBROUTINE statement.

```
SUBROUTINE symbolic-name [(dummy [, dummy ] ... )]
```

A dummy argument in a SUBROUTINE statement can be one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

A dummy procedure name enables actual procedure names to be passed as arguments.

A subroutine can receive control of execution from the main program or from another procedure. A subroutine cannot invoke itself. Execution control transfers to a subroutine through the CALL statement. The CALL statement must specify the symbolic-name of the subroutine you want to invoke and a list of actual arguments that the subroutine requires to produce the desired result. Actual arguments specified in the CALL statement must correspond in number, order, and data type to the dummy arguments specified in the SUBROUTINE statement. Refer to the "CALL Statement" section for more information.

Actual arguments in a CALL statement can be one of the following items.

- an expression
- an array name
- an intrinsic function name
- an external procedure name
- a dummy procedure name
- an alternate return specifier using the statement label of an executable statement in the same program unit as the CALL statement

## 7.6. Functions

A function is a procedure that performs a specific task within an executable FORTRAN program. Execution control transfers to a function through a reference. You cannot use the CALL statement to invoke a function. To reference a function means to use the function name within an expression. Functions can receive control of execution from the main program or from another procedure. A function cannot reference itself.

Unlike a subroutine, a function is specifically designed to return a single value to the program unit that contains the function reference. The function assigns this return value to the function name. Therefore, the return value becomes the value of the function. When the function name appears in an expression, the value of the function is used in the evaluation of the expression. The function name determines the data type for the return value.

There are three classes of functions: external, statement, and intrinsic. specify the symbolic-name and any actual arguments the function requires to produce the desired result. Use the following general format to reference a function.

```
symbolic-name [(actual [, actual]...)]
```

Actual arguments used in the function reference must correspond in number, order, and data type with the dummy arguments in the function definition. Actual arguments in the function reference can be any one of the following items.

- an expression
- an array name
- an intrinsic function name
- an external procedure name
- a dummy procedure name

### 7.6.1. External Functions

An external function, like a subroutine, is a distinct program unit that is defined outside of the program unit that invokes it. Remember, you can write external procedures using a programming language other than FORTRAN, such as C or assembly language. Refer to the "Interfacing FORTRAN and C" chapter in your FORTRAN User's Manual for additional system specific information.

Use the FUNCTION statement to define a program unit as an external function procedure. The FUNCTION statement must be the first statement in the external function. The FUNCTION statement specifies a symbolic name for the external function, a data type for the value the function returns, and a list of dummy arguments the function requires to produce the desired result. Remember, dummy arguments reserve a place and specify a data type for the actual arguments supplied in the function reference. The following syntax specification shows the general format for a FUNCTION statement.

```
type FUNCTION symbolic-name [(dummy [, dummy ] ... )]
```

A dummy argument in a FUNCTION statement can be one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

### 7.6.2. Statement Functions

A statement function is a procedure that is completely defined in a single statement. Unlike an external function, you can only reference a statement function within the program unit that contains the statement function definition. A statement function consists of a symbolic-name to identify the function, a list of dummy arguments the function needs to produce the desired result, and an expression as shown in the following syntax specification.

```
symbolic-name ([dummy [, dummy ] ... ]) \(eq expression
```

A statement function is structured much like an assignment statement. Therefore, the data type of the expression converts to the type of the symbolic-name, if necessary, according to the rules for conversion in assignment as described in the "Assignment Statements" chapter. The symbolic-name is the statement function name.

The dummy arguments reserve a place and declare a data type for actual values that the statement function requires to produce the desired result. Actual arguments are used in the statement function reference and are substituted for the dummy arguments during the actual procedure execution. The dummy arguments in a statement function are local to the statement function. Therefore, you can use the dummy argument names to represent other entities in the same program unit. You cannot use the statement function name to represent another entity within the same program unit.

Statement functions are referenced in an expression. Actual arguments specified in a statement function reference must correspond in number, order, and data type with the dummy arguments in the statement function definition.

Other functions can be referenced within the expression of a statement function. However, the functions that you reference in a statement function expression must be defined before that statement function in the same program unit. The definition of a statement function and all references to that statement function must be in the same program unit.

## 7.7. Alternate Return Specifiers

A CALL statement transfers execution control to a subroutine. A RETURN or END statement transfers execution control from the subroutine back to the program unit that contains the CALL statement. A normal return from a subroutine is when execution control transfers to the first executable statement following the CALL statement in that calling program unit. You can, however, specify alternate return points from subroutines using alternate return specifiers. Alternate return specifiers operate through the passing of arguments between a calling program unit and a subroutine.

An alternate return specifier consists of an asterisk (*) that you specify as a dummy argument to a subroutine in a SUBROUTINE or ENTRY statement. The corresponding actual argument used in a CALL statement must be a statement label and must be preceded by either an asterisk (*) or an ampersand (&). The statement label identifies an executable statement that serves as an alternate return point for the subroutine. You can specify any number of alternate return specifiers for a subroutine.

The RETURN statement has an optional integer expression used to select one alternate return specifier from a series of specifiers. The integer expression indicates which asterisk or ampersand in the SUBROUTINE or ENTRY statement dummy argument list to use for the return. A valid integer expression must be greater than or equal to 1 and less than or equal to the number of asterisks (or ampersands) specified in the SUBROUTINE or ENTRY statement dummy argument list. Remember, each asterisk or ampersand in the dummy argument list corresponds to an actual argument supplied in the CALL statement that invokes the subroutine. The actual arguments must be statement labels that indicate the alternate return points.

For example, the following hypothetical subroutine contains three RETURN statements. The first RETURN statement, statement 110, specifies a normal return from the subroutine because there is no specified integer expression. The second and third RETURN statements, statements 120 and 130, have integer expressions indicating alternate returns.

Statement 100 is an arithmetic IF statement that determines which of the three RETURN statements will execute. The SUBROUTINE statement contains two alternate return asterisks that serve as dummy arguments to the subroutine named thrust.

```
          SUBROUTINE thrust (var1, *, *, var2)
            .
            .
            .
100       IF (dat/val) 110, 120, 130
110       RETURN
120       RETURN 1
130       RETURN 2
```

If the expression in the arithmetic IF statement evaluates to less than zero, the first RETURN statement, statement 110, executes. The first RETURN statement specifies a normal return. If the expression evaluates to zero, the second RETURN statement, statement 120, executes. Note that the second RETURN statement specifies the integer 1 indicating the first alternate return asterisk in the dummy argument list. If the expression evaluates to greater than zero, the third RETURN statement, statement 130, executes. Note that the third RETURN statement specifies the integer 2 indicating the second alternate return asterisk in the dummy argument list.

The following CALL statement calls the thrust subroutine. Note that the second and third actual arguments in the CALL statement are statement labels that correspond to the two dummy argument asterisks in the SUBROUTINE statement.

```
CALL thrust (2.86, *200, *300, 4.13)
```

If the second RETURN statement in the subroutine executes, execution control transfers to the statement identified with the label 200. This happens because the first dummy argument asterisk in the SUBROUTINE statement corresponds to the actual argument 200 in the CALL statement. If the third RETURN statement in the subroutine executes, execution control transfers to the statement identified with the label 300. This happens because the second dummy argument asterisk corresponds to the the actual argument 300. Statement 200 and 300 are alternate return points selected on the basis of the arithmetic IF statement.

Note that if the integer expression used in a RETURN statement is less than 1 or greater than the number of asterisks in the dummy argument list, a normal return from the subroutine is executed.

An ampersand (&) may be used in place of an asterisk (*) in an argument list to indicate an alternate return argument.

**Example:**

```
      CALL UPDATE (I, J, &100, K)
      ...
100   CONTINUE
```

is equivalent to

```
      CALL UPDATE (I, J, *100, K)
      ...
100   CONTINUE
```

**F77 Compatibility Notes:**

The ampersand (&) alternate return argument is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

# Chapter 8
# The FORTRAN Input/Output System

FORTRAN provides a device-independent input/output (I/O) system for transferring data. The I/O system can transfer data from one location to another within a processor's memory. It can also transfer data to and from processor memory and any external device such as a console, printer, or a storage medium such as a disk, floppy disk, or magnetic tape.

This section describes the I/O system in general, as well as the related topics of records, files, I/O units, and data transfer. The "Input/Output Statements" chapter describes the syntax of each FORTRAN I/O statement in detail.

## 8.1. Records

A record is any logically-related set of data items. There are three kinds of records:

- formatted
- unformatted
- endfile

A formatted record is a sequence of ASCII characters; an unformatted record is a sequence of items having any combination of data types. An endfile record is the last record in a file, and is written using the ENDFILE statement. Refer to the "File Positioning Statements" section for additional information.

The length of a formatted record is the number of characters it contains, and depends on the number of characters written to the record when it is created. The length of an unformatted record is specified when it is created. The length of either kind of record can be zero. The physical length of either kind of record is measured in bytes.

Formatted records can only be accessed with formatted I/O statements; unformatted records can only be accessed with unformatted I/O statements. Refer to the "Data Transfer" section for more information.

## 8.2. Files

A file is a sequence of records. The records in a file are either all formatted or all unformatted. A file cannot contain both kinds of records.

Each record in a file has a unique record-number, which is an integer that the I/O system assigns to the record when it is created.

There are two categories of files:

- external
- internal

An external file contains data that can be transferred between internal storage and any external device. Also, an external file can be permanently stored on some external storage medium, such as a floppy disk, hard disk, or magnetic tape.

An internal file is an area of internal storage. An internal file is implemented as a character variable, a character array, or an element of a character array.

The physical size of a file is the number of records in the file multiplied by the length of record, measured in bytes.

Up to 99 files may be open for I/O concurrently, subject to operating system limits.

## 8.3. I/O Units

An I/O unit is a logical or generic designation for a file. That is, at any given time an I/O unit can designate one of several different files.

Internally, the FORTRAN I/O system uses I/O units when transferring data or manipulating files, so the I/O statements described below generally refer to I/O units rather than names of files.

An I/O unit is designated by an unsigned integer in the range 0 to 99.

### 8.3.1. Connection

Connection is the property of an I/O unit that defines the relationship between the I/O unit and a file. An I/O unit is connected when it refers to a specific file; an I/O unit is disconnected when it does not refer to a specific file.

Connection is a symmetric property. That is, when an I/O unit is connected to a file, the file is also connected to the I/O unit. An I/O unit cannot be connected to more than one file at a time, nor can a file be connected to more than one I/O unit at a time.

All FORTRAN I/O statements, except OPEN and CLOSE operate on I/O units that are connected to files on a one-to-one basis. No data transfer can take place on a file unless it is connected to an I/O unit. The I/O unit/file connection must be made when the file is opened, or by preconnection.

### 8.3.2. Preconnection

An I/O unit is preconnected if, by some means external to the program, it is connected to a specific file before the program begins to run. Each FORTRAN program has three preconnected I/O units:

- Unit 0 - the default error output
- Unit 5 - the default console input
- Unit 6 - the default console output

## 8.4. Properties of Files

Each file has an associated set of properties, some of which are determined by the OPEN statement at the time of connection to an I/O unit.

### 8.4.1. Existence

Existence is a file property that describes the potential an executable program has to access the file. That is, at any given time the processor is aware of a set of files, some of which can be accessed by a program, and some of which cannot.

The files that a program can potentially access are said to exist for that program. The files that a program cannot access do not exist. For example, a file could be protected for security reasons, or a file could be in use by another program. In such a case, the file is inaccessible to the program, and therefore does not exist.

The property of existence only applies to a file in relation to a particular executable program. It does not apply to the file's physical existence within the environment of an implementation.

There are four possible combinations of connection and existence.

1. A file can exist and be connected (a disk file being written to).
2. A file can exist and not be connected (a disk file that is not yet open).
3. A file can be connected but not exist (a newly created disk file before the first record is written).
4. A file can both not exist and not be connected (a disk file that was erased).

### 8.4.2. Access Method

There are two methods for accessing a file:

- sequential access
- direct access

With sequential access, you access the records in the same order that they were created. With direct access, you can access the records in any order.

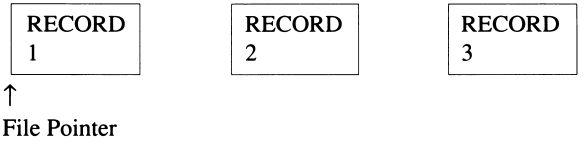
You can access an external file using either access method, but you can only access an internal file using sequential access.

If a file is connected for direct access, all the records must have the same length. A file connected for sequential access can have records with different lengths.

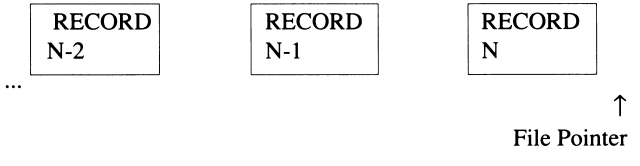
### 8.4.3. Position

When a file is connected to an I/O unit, the file has the property of position. At any given time, the file position is determined by a file pointer. The file pointer is not a physical entity; rather, it is an internal reference the I/O system uses to keep track of the file position.

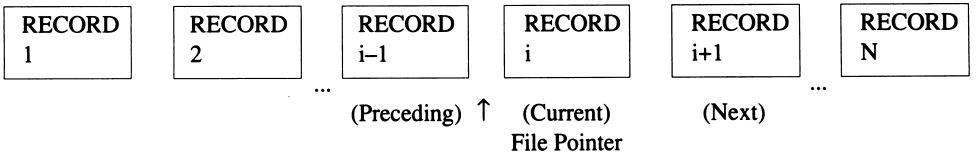
The following diagrams illustrate the concept of file position.




In the diagram above, the file is positioned at the initial point, which is just before the first record.



In the previous diagram , the file is positioned at the terminal point, which is just after the last record.




In the diagram above, the file is positioned at a specific record. This is the current record. If the file is not positioned at or within a record, there is no current record.

If a file contains N records, and the file pointer points to record i, where $1 \leq i \leq N$, then record i+1 is the next record, and record i–1 is the preceding record. If i=1, there is no preceding record. If i=N or if N=0, there is no next record.

## 8.5. Categories of I/O Statements

FORTRAN provides three categories of I/O statements:

- data transfer statements
- file positioning statements
- auxiliary statements

Data transfer statements move data between internal (processor) storage and a file. Data transfer statements can reference both internal and external files.

File positioning statements affect the position of the file pointer relative to a specific file. File positioning statements cannot reference internal files.

Auxiliary I/O statements manipulate the connection of I/O units to external devices and media, and inquire about the characteristics of a particular connection. Auxiliary statements cannot reference internal files.

The "Input/Output Statements" chapter describes the syntax of each I/O statement in detail.

### 8.5.1. Data Transfer Statements

The data transfer statements are

- READ
- WRITE
- PRINT
- ENCODE
- DECODE

The READ statement inputs data from a specific I/O unit. The WRITE statement outputs data to a specific I/O unit. The PRINT statement outputs data to the default output I/O unit.

If a READ or WRITE statement contains a format specifier it is a formatted I/O statement. Otherwise, it is an unformatted I/O statement. Refer to the "Data Transfer" section for additional information.

The ENCODE and DECODE statements transfer data between internal locations only. DECODE translates data from external character form to internal binary representation. ENCODE translates data from internal binary representation to external character form. Both ENCODE and DECODE translate the data using a format specifier.

Using formatted READ and WRITE statements with internal files achieves the same results as ENCODE and DECODE. FORTRAN supports ENCODE and DECODE only for compatibility with older FORTRAN compilers.

### 8.5.2. File Positioning Statements

The file positioning statements are

- BACKSPACE
- REWIND
- ENDFILE

The BACKSPACE statement moves the file pointer to the start of the preceding record. The REWIND statement moves the file pointer to the initial point of a file. The ENDFILE statement marks the preceding record as the last record in a file.

All three file positioning statements require that the file be connected for sequential access.

### 8.5.3. Auxiliary I/O Statements

The auxiliary I/O statements are

- OPEN
- CLOSE
- INQUIRE

The OPEN statement can

- create a file and connect it to an I/O unit.
- recreate a preconnected file.
- connect an existing file to an I/O unit.
- change the characteristics of an existing I/O unit/file connection.

The CLOSE statement disconnects a file from an I/O unit.

The INQUIRE statement returns information about the characteristics of a named file, or the connection of a file to a particular I/O unit.

## 8.6. Data Transfer

Data transfer is the process of moving data between records and individual items specified in an I/O list. An I/O list is a list of data items whose values are transferred by a data transfer statement.

The I/O system performs the following steps each time it executes a data transfer statement:

1) determines the direction of the data transfer.
2) identifies the I/O unit.
3) establishes the format (if one is specified).
4) positions the file before the transfer.
5) performs the transfer between the file and the I/O list.
6) positions the file after the transfer.
7) sets the I/O status specifier (if one is defined).

There are two categories of data transfer:

- formatted
- unformatted

Formatted data transfer requires formatted I/O statements; unformatted data transfer requires unformatted I/O statements.

## 8.7. Formatted Transfer

During formatted data transfer, the I/O system transfers data between a file and the I/O list, and performs editing on the data. The current record and (optionally) other records are read from or written to.

### 8.7.1. Editing

The editing performed during the transfer is directed by a format specification. A format is a description of the arrangement or pattern that data has when it is read or written. The same data can be read or written with different formats to suit different situations.

A format specification is a list of items separated by commas and enclosed in parentheses. The list is called a format-list, and contains items called edit descriptors. A format-list can also contain other (nested) format-lists.

There are two kinds of edit descriptors:

- repeatable
- nonrepeatable

Repeatable edit descriptors control the editing of character, logical, and numeric data. Each item in the I/O list corresponds to a repeatable edit descriptor in the format-list.

Each repeatable edit descriptor can be preceded by an integer constant called a repeat-factor, which tells the I/O system how many times to repeat the edit specified in the edit descriptor.

Nonrepeatable edit descriptors are not associated with specific data items in the I/O list. Instead, they control such things as column position, spacing, sign control, blank control, and line termination.

### 8.7.2. Format Control

The interaction between the I/O list and the format specification is a dynamic process called format control.

Format control always proceeds from left to right matching each item in the I/O list with the next repeatable edit descriptor. Format control executes nonrepeatable edit descriptors as they are encountered.

If an edit descriptor has a repeat-factor, format control processes the I/O list as if it contained the specified number of consecutive items.

If the format-list ends before reaching the end of the I/O list, format control reverts to the beginning of the last nested format-list, if there is one. If there is none, format control reverts to the beginning of the format-specification and again passes through the I/O list. Each time format control reverts, it accesses a new record.

### 8.7.3. List-directed Formatting

List-directed formatting is an alternative method of formatted data transfer. List-directed formatting is specified by using an asterisk (*) as the format specification in an I/O statement. A FORMAT statement is not required.

When the I/O system executes a list-directed READ, WRITE, or PRINT statement, it begins a new record, and formats each input or output value using the data type and field width of the corresponding I/O list item to generate an equivalent edit descriptor.

The "Format Statement and Format Specification" chapter completely describes rules for writing valid format specifications, the actions of the various edit descriptors, and list-directed formatting.

## 8.8. Unformatted Transfer

During unformatted data transfer, the I/O system transfers data between the current record of a file and the I/O list. The I/O system does not perform any editing of the data, and only one record is read from or written to.

# Chapter 9
# Structural Statements

The FORTRAN structural statements enable you to define the different kinds of program units that make up a FORTRAN program. A program unit is a logical, self-contained sequence of statements and optional comment lines that forms a discrete part of the larger program. All FORTRAN programs consist of one or more program units. There are five statements that you can use to structure a FORTRAN program.

- BLOCK DATA Statement
- ENTRY Statement
- FUNCTION Statement
- PROGRAM Statement
- SUBROUTINE Statement

## 9.1. BLOCK DATA Statement

The BLOCK DATA statement identifies a program unit as a block data subprogram. A block data subprogram is an nonexecutable program unit that enables you to specify initial values for variables and array elements in named common blocks.

The BLOCK DATA statement can specify a symbolic name for the block data subprogram as shown in the following syntax specification.

**Syntax:**

```
BLOCK DATA [symbolic-name]
```

The symbolic name is optional. If you specify a symbolic name, it must not be the same as the symbolic name of the main program, an external procedure, a common block, or another block data subprogram within the same executable program. The symbolic name must not be the same as any local name used in the block data subprogram.

A block data subprogram can contain only certain specification statements: COMMON, DIMENSION, DATA, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and type statements. You can also include comment lines. The last statement in a block data subprogram must be the END statement.

An executable FORTRAN program can contain any number of block data subprograms. Refer to the "Block Data" section for additional information.

**Examples:**

```
BLOCK DATA
BLOCK DATA initial
```

## 9.2. ENTRY Statement

Use an ENTRY statement to specify a secondary entry point in a procedure. A secondary entry point enables execution of the procedure to begin with an executable statement other than the first executable statement in the procedure. You can place an ENTRY statement anywhere after the FUNCTION statement in an external function procedure or after the SUBROUTINE statement in a subroutine procedure. A procedure can contain more than one ENTRY statement.

The ENTRY statement specifies a symbolic name to identify the entry point in the procedure and a list of dummy arguments. The dummy arguments hold a place and specify a data type for the actual arguments specified in the procedure reference.

**Syntax:**

```
ENTRY symbolic-name [(dummy[, dummy] ... )]
```

You cannot use an ENTRY statement between a block IF statement and the corresponding END IF statement, or in the range of a DO statement.

Use the CALL statement to reference a secondary entry point in a subroutine. To reference a secondary entry point in a function, use the entry point name in an expression. The dummy argument list in an ENTRY statement need not match the dummy argument list in the SUBROUTINE or FUNCTION statement at the beginning of the procedure. However, the actual arguments specified in an entry point reference must correspond in number, order, and data type with the dummy arguments specified in the corresponding ENTRY statement.

**Example:**

```
ENTRY enter2 (pressure, volume)
```

## 9.3. FUNCTION Statement

Use the FUNCTION statement to define a program unit as an external function procedure. An external function, like a subroutine, is a distinct program unit defined outside of the program unit that invokes it.

The FUNCTION statement specifies a symbolic name for the external function, a data type for the value the function returns, and a list of dummy arguments. Dummy arguments reserve a place and specify a data type for the actual arguments supplied in the function reference.

**Syntax:**

```
type FUNCTION symbolic-name [(dummy[, dummy] ... )]
```

The type specification can be any one of the standard data types described in the "Data Types" chapter. The type specification determines the data type of the value that the function returns.

A dummy argument in a FUNCTION statement can be any one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

The symbolic name of a function serves as a variable name to hold the value that the function returns. The symbolic name must be used as a variable name within the function procedure. The value of the variable name when a RETURN or END statement executes (the last value assigned to the function name) is the value of the function. An external function can use any of its dummy arguments to return values in addition to returning the value of the function.

An external function can receive control of execution from the main program or from another procedure. Execution control transfers to an external function through a reference in an expression. Refer to the "Functions" section for more information.

You can write external functions using a programming language other than FORTRAN, such as C or an assembly language. Refer to the "Interfacing FORTRAN and C" chapter in your User's Manual for more information.

**Examples:**

```
INTEGER FUNCTION reset(arg1, arg2, arg3)
DOUBLE PRECISION FUNCTION molarity(atomwt, grams)
```

## 9.4. PROGRAM Statement

Use the PROGRAM statement to define a program unit as the main program. The main program is the program unit that receives control at the beginning of execution. During execution, the main program can invoke a variety of subprograms that perform different tasks. Control returns to the main program to terminate execution unless you use a STOP statement in a subprogram.

The PROGRAM statement specifies a symbolic name for the main program as shown in the following syntax specification.

**Syntax:**

```
PROGRAM symbolic-name
```

The PROGRAM statement is not required, but if used, it must be the first statement of the main program. Refer to the "Main Program" section for additional information.

In the example below, the PROGRAM statement specifies a main program with the symbolic name "main".

**Example:**

```
PROGRAM main
```

## 9.5. SUBROUTINE Statement

Use the SUBROUTINE statement to define a program unit as a subroutine procedure. A subroutine is an external procedure. This means that a subroutine is a distinct named program unit defined outside of the program unit that invokes it.

The SUBROUTINE statement specifies a symbolic name for the subroutine and a list of dummy arguments. Dummy arguments reserve a place and specify a data type for the actual arguments supplied in the subroutine call.

**Syntax**: SUBROUTINE symbolic-name [(dummy[, dummy] ... )]

A dummy argument in a SUBROUTINE statement can be any one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

A subroutine can receive control of execution from the main program or from another procedure. Execution control transfers to a subroutine through the CALL statement. A subroutine cannot invoke itself.

You can write subroutines using a programming language other than FORTRAN, such as C or an assembly language. Refer to the "Interfacing FORTRAN and C" chapter in your User's Guide for more information.

**Example:**

```
SUBROUTINE calculations (arg1, arg2, arg3)
```

## 9.6. Recursion

Subroutines and functions in FORTRAN can be recursive. In order for recursion to be useful, there must be a way to keep local variables on the stack. By default, all local variables have static extent, which is to say that local variables have permanently allocated memory locations which retain their value after the subroutine exits. Normally, it is desirable to have each invocation of a recursive subroutine allocate new memory for its local variables. This can be done by declaring the variables AUTOMATIC.

The following example program demonstrates recursion:

```
SUBROUTINE RECURSIVE
IMPLICIT AUTOMATIC (A-Z)
INTEGER X(15)
IF (A.EQ.1) THEN
       DO I=1,15
       X(I) = I
       ENDDO
ELSE IF (X(5).NE.0) THEN
       CALL RECURSIVE(A,B,C)
ELSE
       DO I=1,15
       X(I) = X(I) - X(I-1)
       ENDDO
ENDIF
END
```

# Chapter 10
# Specification Statements

The FORTRAN specification statements enable you to define data types, establish the interpretation and use of symbolic names, and control the management of storage. Specification statements are nonexecutable statements that appear most often at the beginning of a program unit, before the executable statements. The following specification statements are available in FORTRAN:

- Type statements
- AUTOMATIC statement
- COMMON statement
- DIMENSION statement
- EQUIVALENCE statement
- EXTERNAL statement
- IMPLICIT statement
- IMPLICIT NONE statement
- INTRINSIC statement
- NAMELIST statement
- PARAMETER statement
- SAVE statement
- VIRTUAL statement
- VOLATILE statement

## 10.1. Type Statements

Type statements specify a data type explicitly for symbolic names that represent constants, variables, arrays, external functions, and statement functions. Once a symbolic name appears in a type statement, the data type of that symbolic name is defined for the entire program unit. An explicit data type specification confirms or supercedes the implicit data type convention. You can also use type statements to specify the dimensions of an array. There are six type statements in FORTRAN.

- INTEGER type statement
- REAL type statement
- DOUBLE PRECISION type statement
- COMPLEX type statement
- LOGICAL type statement
- CHARACTER type statement

Each of the data types are discussed in the following sections. Additional information on data types can be found in the "Data Types" chapter.

### 10.1.1 INTEGER Type

Use the INTEGER statement to specify an integer data type for symbolic names that represent integer constants, variables, arrays, external functions, and statement functions.

**Syntax:**

```
INTEGER[*number]  symbolic-name[, symbolic-name] ...
```

The optional asterisk and number that follows the keyword INTEGER specifies the number of bytes each integer value occupies in memory. In FORTRAN, you can specify integers that are 1, 2, or 4 bytes long. If you do not specify the asterisk and number, integers default to 4 bytes. A two byte integer can represent values that range from –32768 to 32767. A one byte integer can represent values that range from –128 to 127.

You can also use the -i2 compile time option to specify that the default size for integers (and logicals) is 2 bytes. Refer to "Compile Time Options" chapter of your system-specific User's Guide for more information on compiler option switches.

In the first example below, by default, each variable and each element in the array occupies 4 bytes of storage. However if the -i2 compile time option is used each variable and each element in the array will occupy 2 bytes.

**Examples:**

```
INTEGER var1, var2, array(10)
INTEGER*4 var1, var2, array(10)
INTEGER*1 var1, var2, array(10)
INTEGER*2 var1, var2, array(10)
```

### 10.1.2. REAL Type

Use the REAL statement to specify a real data type for symbolic names that represent real number constants, variables, arrays, external functions, and statement functions.

**Syntax:**

```
REAL[*number]  symbolic-name[, symbolic-name] ...
```

The optional asterisk (*) and number that follows the keyword REAL specifies the number of bytes each real number value occupies in memory. In FORTRAN, you can specify real values that are 4 or 8 bytes long. Specifying 8 byte real values using a REAL*8 type statement is equivalent to using the DOUBLE PRECISION type statement.

If you do not specify the asterisk and number, real numbers default to 4 bytes. The range, precision, and representation of real numbers is given in the FORTRAN User's Manual for your target system.

In the first two examples below, each variable and each element in the array occupy 4 bytes of storage. The first two examples are equivalent. The third example specifies 8 byte real values.

**Examples:**

```
REAL var1, var2, array(10)
REAL*4 var1, var2, array(10)
REAL*8 var1, var2, array(10)
```

### 10.1.3. DOUBLE PRECISION Type

Use the DOUBLE PRECISION statement to specify a double precision real data type for symbolic names that represent real number constants, variables, arrays, external functions, and statement functions. Using the DOUBLE PRECISION type statement is equivalent to specifying 8 byte real values using a REAL*8 type statement.

**Syntax:**

```
DOUBLE PRECISION  symbolic-name[, symbolic-name] ...
```

In FORTRAN, a double precision real number occupies 8 bytes of storage. The range, precision, and representation of double precision real numbers is given in the FORTRAN User's Manual for your target system.

In the following example the DOUBLE PRECISION type statement specifies two double precision variables and a ten element double precision array. Each variable and each element in the array occupies 8 bytes of storage.

**Example:**

```
DOUBLE PRECISION  var1, var2, array(10)
```

### 10.1.4. COMPLEX Type

Use the COMPLEX statement to specify a complex data type for symbolic names that represent complex constants, variables, arrays, external functions, and statement functions.

**Syntax:**

```
COMPLEX[*number]  symbolic-name[, symbolic-name] ...
```

The optional asterisk (*) and number that follows the keyword

COMPLEX specifies the number of bytes each complex value occupies in memory. In FORTRAN, you can specify complex values that are 8 or 16 bytes long. If you do not specify the asterisk and number, complex numbers default to 8 bytes.

In the first two examples below, each variable and each element in the array occupies 8 bytes of storage. The first two examples are equivalent.

**Examples:**

```
COMPLEX var1, var2, array(10)
COMPLEX*8 var1, var2, array(10)
COMPLEX*16 var1, var2, array(10)
```

Refer to the "Complex Type" section for more information on complex numbers.

### 10.1.5. LOGICAL Type

Use the LOGICAL statement to specify a logical data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions.

**Syntax:**

```
LOGICAL[*number]  symbolic-name[, symbolic-name] ...
```

The optional asterisk and number that follows the keyword LOGICAL specifies the number of bytes each logical value occupies in memory. In FORTRAN, you can specify integers that are 1, 2, or 4 bytes long. If you do not specify the asterisk and number, logicals default to four bytes.

You can also use the -i2 compile time option to specify that the default size for logicals (and integers) is 2 bytes. Refer to "Compile Time Options" chapter of your system-specific User's Guide for more information on compiler option switches.

In the first example below, by default, each variable and each element in the array occupies 4 bytes of storage. However if the -i2 compile time option is used each variable and each element in the array will occupy 2 bytes.

**Examples:**

```
LOGICAL var1, var2, array(10)
LOGICAL*4 var1, var2, array(10)
LOGICAL*1 var1, var2, array(10)
LOGICAL*2 var1, var2, array(10)
```

### 10.1.6. CHARACTER Type

Use the CHARACTER statement to specify a character data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions.

**Syntax:**

```
CHARACTER[*number]  symbolic-name[*number] [, symbolic-name[*number]]...
```

The asterisk and number that follows the keyword CHARACTER specifies the length or number of characters for each character item that you list in the statement. Alternatively, you can specify the length of each character item individually with an asterisk and number that immediately follows each symbolic name. If you do not specify the length of the character items using either method, the length of the items default to a value of one.

In certain cases, you can specify a length with an asterisk enclosed in parentheses, (*). An asterisk enclosed in parentheses indicates that a dummy argument assumes the length specification from the corresponding actual argument or that a function name obtains its length specification from the function reference. If you use an asterisk enclosed in parentheses as a length specifier for the symbolic name of a character constant, the symbolic name assumes the actual length of the constant that it represents.

Each variable and array element declared in the first three examples below has a length of 1 character. The three examples are equivalent.

In the last example below, var1 has a length of 15 characters, var2 and each element in the array have a length of 5 characters.

**Examples:**

```
CHARACTER var1, var2, array(10)
CHARACTER*1 var1, var2, array(10)
CHARACTER var1*1, var2*1, array(10)*1
CHARACTER*5 var1*15, var2, array(10)
```

## 10.2. AUTOMATIC Statement

The AUTOMATIC statement causes variables to become undefined when the RETURN or END statement of a subprogram is executed. It also allows variables to be allocated to registers for efficiency and makes re-entrant/recursive code possible. Normally, the compiler behaves as if a SAVE statement was placed at the beginning of all subprograms, causing all appropriate program entities to retain their values upon exit.

## 10.3. COMMON Statement

Use the COMMON statement to define common blocks. Common blocks are contiguous areas of storage containing data that different program units can share. When you declare common blocks of the same name in different program units, the blocks all share the same storage space when the program units are combined into an executable program.

A COMMON statement specifies symbolic names enclosed in slashes to identify each common block and lists of variable names, array names, and array declarators that represent the values in each common block that you specify.

**Syntax:**

```
COMMON  [/symbolic-name/] common-list [[,]/[symbolic-name]/ common-list]...
```

All items that you place in a common-list are contained in the common block identified by the symbolic-name that precedes the list. You must delimit all the items in a common-list with commas. The slashes serve to delimit each symbolic-name/common-list pair that you specify in the COMMON statement. You can also use commas as secondary delimiters.

If you do not specify a symbolic-name, all items in the corresponding common-list are declared to be in an unnamed (blank) common block. A program can only have one unnamed common block. If you omit the first symbolic-name in a COMMON statement that specifies more than one common block, the slashes are optional. Otherwise, slashes are required, with or without a symbolic-name, to delimit one common block specification from another.

COMMON statements are frequently used in a block data subprogram. A block data subprogram enables you to specify initial values for variables and array elements that are listed in named common blocks. Refer to the "Block Data" section for additional information on block data subprograms.

Note that the unnamed block in the last example below contains var1, var2, and var3, as well as the array names array1 and array2.

**Examples:**

```
COMMON  /atomic/var1,var2,array
COMMON  /com1/var1,var2,array1/com2/var3,var4,array2
COMMON  var1,var2/com1/var8,array3,/com2/var12,var13,//var3,array1,array2
```

## 10.4. DIMENSION Statement

Use a DIMENSION statement to declare arrays in a program unit. A DIMENSION statement can contain any number of array declarators. An array declarator specifies a symbolic name to identify the array and a number of dimension declarators. Each array declarator that you use in a DIMENSION statement specifies a different array. In the following syntax specification, the abbreviation dim stands for dimension declarator.

**Syntax:**

```
DIMENSION symbolic-name (dim[,dim]...) [symbolic-name (dim[,dim]...)] ...
```

Dimension declarators specify the number of elements in each array dimension that you declare. You set the number of elements with a lower- and upper-bound value. The lower- and upper-bound values are called dimension bounds. The form for a dimension declarator is as follows.

```
[lower-bound:]upper-bound
```

Dimension bounds can be arithmetic constant or variable expressions that evaluate to integers. The optional lower-bound value can be negative, zero, or positive. If you do not specify a lower-bound, a value of 1 is implied. The upper-bound value can be negative, zero, positive, or an asterisk indicating an assumed-size array declarator. The use of a variable expression as a dimension bound value constitutes an adjustable array declarator. Refer to the "Adjustable Array Declarators" and "Assumed-size array declarators" section for more information.

The number of dimension declarators that you specify in an array declarator determines the number of dimensions for the array. An array in FORTRAN can have a maximum of seven dimensions.

The size of an array is equal to the number of elements in the array. The number of elements is equal to the product of the dimension sizes specified in the array declarator.

**Examples:**

The following DIMENSION statement uses one array declarator to specify an array named values. The values array is two-dimensional. The first dimension has a lower bound of –12 and an upper bound of 12. The second dimension has an implied lower bound of 1 and an upper bound of 5. values consists of 125 elements.

```
DIMENSION  values (-12:12, 5)
```

The next DIMENSION statement uses three array declarators to specify arrays named arr1, arr2, and arr3. All three are one-dimensional arrays with a lower bound of 0 and an upper bound of 19. Each array consists of 20 elements.

```
DIMENSION  arr1(0:19), arr2(0:19), arr3(0:19)
```

## 10.5. EQUIVALENCE Statement

The EQUIVALENCE statement enables two or more program entities to share the same memory space. You can use the EQUIVALENCE statement to conserve memory space or to specify two or more symbolic names for the same program entity.

**Syntax:**

```
EQUIVALENCE  (item-list)[, (item-list)] ...
```

You can specify variable names, array names, array element names, or character substrings in an EQUIVALENCE statement item-list. All the program entities in one item-list share the same starting address in memory, even if the length of the items differ. Each item-list must contain at least two items.

You can specify program entities of different data types in an EQUIVALENCE statement. For example, you can specify an integer variable and a complex variable in one item-list. The result is that the integer variable shares storage with the real portion of the complex variable. No data type conversion takes place.

**Examples:**

In the following example, the EQUIVALENCE statement specifies that the double precision variable named DOUBVAR and the first two elements of an integer array named INTARR occupy the same storage units.

```
INTEGER*2 INTARR(5)
DOUBLE PRECISION DOUBVAR
EQUIVALENCE (INTARR(1),DOUBVAR)
```

In the next example, the EQUIVALENCE statement specifies that the first five characters of two character variables share the same storage units.

```
CHARACTER CODE*5, ZONE*12
EQUIVALENCE (CODE,ZONE)
```

### 10.5.1. Use of Single-Subscript in EQUIVALENCE

A single subscript may be used in an EQUIVALENCE statement to identify an element of a multi-dimensional array. The linear element number may then be used to reference the array elements.

**Example:**

The linear element numbers for array A as defined in the following statements, are shown in the following table.

```
DIMENSION B(6)
DIMENSION A(2,3)
EQUIVALENCE (A(2),B(2))
```

| Linear Element | Array A Element | Array B Element |
|---|---|---|
| 1 | A(1,1) | B(1) |
| 2 | A(2,1) | B(2) |
| 3 | A(1,2) | B(3) |
| 4 | A(2,2) | B(4) |
| 5 | A(1,3) | B(5) |
| 6 | A(2,3) | B(6) |

**F77 Compatibility Notes:**

The single-subscript identification of a multi-dimensional array element is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 10.6. EXTERNAL Statement

Use the EXTERNAL statement to specify external procedure names and dummy procedure names for use as actual arguments. If you want to use an external or dummy procedure name as an actual argument in a program unit, you must specify the name in an EXTERNAL statement.

**Syntax:**

```
EXTERNAL  symbolic-name[, symbolic-name] ...
```

A symbolic-name cannot represent an intrinsic function. Use the INTRINSIC statement to specify intrinsic functions as actual arguments.

Each symbolic name that you specify can appear only once in all the EXTERNAL statements that a program unit contains.

In the example below, the EXTERNAL statement declares three symbolic names as representing external procedures. The three external procedure names can be used as actual arguments.

**Example:**

```
EXTERNAL  alpha, bravo, delta
```

## 10.7. IMPLICIT Statement

In the absence of an explicit data type specification, symbolic names that begin with the letters I, J, K, L, M, or N have an implied integer data type. Symbolic names that begin with any other letter of the alphabet have an implied real (REAL*4) data type. Use the IMPLICIT statement to change or confirm the implicit data type convention.

**Syntax:**

```
IMPLICIT  data-type (letter-list)[, (letter-list)] ...
```

The IMPLICIT statement assigns the specified data type to symbolic names that begin with the letters you list in the statement. The data-type can be any one of the standard FORTRAN data types. A letter-list can be a list of single letters or ranges of letters. A range specification consists of the first and last letters in

the range separated with a minus sign. You must specify the range letters in alphabetical order. You cannot use the same letter, singularly or in a range specification, more than once in the IMPLICIT statements for one program unit.

You can specify a length for a CHARACTER type data as shown in the following examples. The length value must be an unsigned integer constant or an integer constant expression enclosed in parentheses. If you do not specify a length, a value of one is implied.

A program unit can contain more than one IMPLICIT statement. However, IMPLICIT statements must precede all other specification statements in a program unit except PARAMETER statements.

**Examples:**

The following IMPLICIT statements confirm the implicit data type convention.

```
IMPLICIT  INTEGER(I, J, K, L, M, N)
IMPLICIT  REAL(A-H, O-Z)
```

The next IMPLICIT statement specifies that all symbolic names beginning with the letters A, B, C, D, or E have an implied data type of INTEGER*2. Note that the implicit data type convention remains in effect for the other letters of the alphabet.

```
IMPLICIT  INTEGER*2(A, B, C, D, E)
```

The next IMPLICIT statement specify character data types. Symbolic names that begin with letters ranging from A to M and S to Z have a CHARACTER data type and a length of 10.

```
IMPLICIT  CHARACTER*10(A-M, S-Z)
```

## 10.8. IMPLICIT NONE Statement

The IMPLICIT NONE statement is used to override all implicit defaults, forcing all symbolic names to be declared explicitly. No other IMPLICIT statements are allowed in conjunction with IMPLICIT NONE. The -u compile time option performs the equivalent function. Please refer to your User's Guide for details on compiler options and defaults.

**F77 Compatibility Notes:**

The IMPLICIT NONE statement is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 10.9. INTRINSIC Statement

Use the INTRINSIC statement to specify intrinsic function names for use as actual arguments. If you want to use an intrinsic function name as an actual argument in a program unit, you must specify the name in an INTRINSIC statement.

**Syntax:**

```
INTRINSIC  symbolic-name[, symbolic-name] ...
```

The "FORTRAN Intrinsic Functions" chapter describes each of the individual FORTRAN intrinsic functions with examples.

In the example below, the INTRINSIC statement declares three symbolic names as representing intrinsic functions. The three intrinsic function names can be used as actual arguments.

**Examples:**

```
INTRINSIC  exp, tan, sqrt
```

## 10.10. NAMELIST Statement

The NAMELIST statement is used to group together a list of variables and/or arrays under a single unique groupname. This symbolic name may then be used to reference all or part of the list in an I/O statement.

Syntax: NAMELIST /group/ namelist [[,]/group/ namelist ...]

where group is a unique symbolic name and namelist consists of a list of variables and/or array names, separated by commas, that are to be referenced by group.

The order in which the list elements are specified in the NAMELIST directive defines the output order in NAMELIST-directed I/O. Elements can be of any data type, either explicit or implicit, but may not consist of individual array elements, substrings, records or dummy arguments. NAMELIST-directed input may, however, be used to assign values to substrings, array elements and/or records.

NAMELIST -directed I/O can only be used for list elements previously defined with the NAMELIST directive, but all list elements defined need not be referenced.

**Example:**

```
CHARACTER*20     NAME
NAMELIST   /PAYROLL/  NAME, EMPNUM, DATE, HOURS
NAMELIST   /VACATION/ NAME, EMPNUM, VACDAYS
```

The PAYROLL groupname in the NAMELIST above associates the variables NAME, EMPNUM, DATE and HOURS. Different data types may be grouped together under a single groupname. Note that the variables NAME and EMPNUM are associated with two different groupnames, PAYROLL and VACATION.

**F77 Compatibility Notes:**

The NAMELIST statement and it's associated specifiers are not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 10.10.1. NML Namelist Specifier

The namelist specifier is used in a READ or WRITE statement control list to indicate that namelist-directed I/O is to be performed on the specified group.

**Syntax:** [NML=]group

where group is the name of a list previously defined in a NAMELIST statement.

The NML keyword is optional if and only if the first parameter of the I/O control list is a logical unit number without the optional UNIT= keyword, and the namelist parameter directly follows the unit number in the list. A namelist specifier may not be used in a statement which contains a format specifier.

**Example:**

```
C   Valid formats for namelist-directed I/O:
C
    READ (5,PAYROLL)
    READ (UNIT=2, NML=VACATION)
C
C   Invalid formats for namelist-directed I/O:
C
    READ (UNIT=5, PAYROLL)     ! Must use NML= if UNIT= used
    READ (5,100) VACATION      ! Namelist not allowed with format specifiers
```

### 10.10.2. Namelist-Directed READ Statement

The namelist-directed READ statement reads records sequentially until the specified group name is encountered, translates namelist data into internal format according to their corresponding data types, and assigns the translated data to the specified namelist elements.

Namelist data records are enclosed within a pair of dollar sign ($) or ampersand (&) special symbols. The starting record must have either a dollar sign or ampersand in the first non-blank column of the record, followed by the namelist groupname specifier. The namelist data is terminated by a closing dollar sign or ampersand symbol, optionally followed by the keyword END. Either dollar sign or ampersand pairs may be used, however the two may not be mixed in a single data record set.

### 10.10.3. Namelist Record Format

**Syntax:** $group-name element=value [ element=value ...] $[END]

where group-name is the group defined in a NAMELIST statement, element is a NAMELIST element name and value is the assigned value for that element. Element-value pairs may be separated by commas, tabs, or spaces, and may appear on separate records.

The value assigned to an element may consist of one or more constants. Multiple occurrences of a single constant value may be represented in the form n*c, where n is the number of occurrences of the constant value c. Multiple null values may be specified in the form n*, where n is an integer number indicating the number of nulls to be supplied.

Input records for namelist-directed READs are subject to the following rules:

- The namelist group-name may not contain spaces or tabs, and must be specified in the first record in entirety.
- The value assigned to a NAMELIST element may not contain spaces or tabs, other than within parentheses of a subscript or substring.
- Element-value pairs may not span records, both the NAMELIST element and it's associated value must be contained in a single record.
- Constants used in value assignments must be specified in standard Fortran form. For example, a complex constant takes the form of a real or integer number pair, separated by a comma and enclosed in parenthesis. Only leading and/or trailing spaces are allowed, spaces may not appear within a numeric constant.

- Logical constants may be assigned true or false values by using one of the following special symbols:
  TRUE .TRUE., T, .T, t or .t
  FALSE .FALSE., F, .F, f or .f
- Character constants must be enclosed in apostrophes. An apostrophe within a character constant may be specified by two consecutive apostrophes.
- Hollerith, octal and hexadecimal constants are not allowed in NAMELIST records.
- A sequence of constants may be separated by spaces, tabs or commas. Two consecutive commas specifies a null value, indicating that the corresponding array value should remain unchanged. A null value may be used to represent an entire complex constant, but not for only one of the values in a complex pair.
- Element-value pairs may be separated by spaces, tabs or commas. Consecutive commas, which would imply a null assignment for an unspecified variable, are not allowed. Any number of spaces or tabs may be placed on either side of the equals (=) sign in an element-value pair assignment.

### 10.10.4. Assigning Values to NAMELIST Elements

Values are assigned to NAMELIST elements in the form element=value, where element is the variable name that appears in the NAMELIST statement, and value is a constant, or series of constants. Symbols defined in a PARAMETER statement may not be used as constants in an element-value assignment.

Constants may be integer, real, logical, complex or character values. If the data type of the constant specified does not match the data type of the corresponding NAMELIST element, conversion is performed following standard rules for arithmetic assignments. Conversion between numeric and character data types is not permitted.

If a list of values is to be assigned to an array, the first value specified in the list will be stored in the first array element, and so forth. Not all array elements need be assigned, however the total number of values specified may not exceed the array length. Consecutive commas within a value list indicate null assignments. If an array element, rather than array name is specified, the corresponding values are assigned beginning with that element.

Values need not be assigned to every element in a NAMELIST group. If the value is to remain unchanged, the element name may simply be omitted from the record.

**Example:**

Input file records:

```
$VACATION NAME='Smith', RATE = 1.2 ACCRUED = 20.5 $END
&VACATION
NAME='Jones'
RATE=1.2
ACCRUED=40.5
&
```

Program segment:

```
C     Monthly vacation update program
C
      NAMELIST /VACATION/ NAME, RATE, ACCRUED
      CHARACTER*15 NAME
      REAL*4         RATE, ACCRUED, WEEKLY, DEDUCT
      INTEGER        EMP_NO
      ...
      READ (5,VACATION)
      ...
      ACCRUED = ACCRUED + RATE
      WRITE (3,VACATION)
      ...
```

The program in the example above reads vacation file records from logical unit 5, calculates the vacation accrued at the monthly rate, and writes the updated records to logical unit 3. Note that input element-value pairs may be specified on one or more records.

### 10.10.5. Prompting for NAMELIST Values

A program which is executing a namelist-directed READ from the keyboard may prompt the user for namelist elements. In this instance, the user simply enters the value for the requested element, followed by a carriage return. To initiate the prompting option, the user enters an initiating special symbol (dollar sign or ampersand), followed by the NAMELIST group name, one or more spaces and a question mark. The entire string is then terminated by a carriage return.

Using the VACATION record defined in the previous example, a user might enter

```
$VACATION ?
```

to cause the program to prompt for NAME, RATE and ACCRUED input.

### 10.10.6. Namelist-Directed WRITE Statements

The namelist-directed WRITE statement takes the internal data for the namelist elements, translates it from internal format to the appropriate data types and writes the translated data to external records of a sequential file.

Every NAMELIST element is written out, along with its associated value, one element-value pair per record. The format of the output data in a namelist-directed WRITE is compatible with the format required by a namelist-directed READ or ACCEPT statement. The NAMELIST elements are written out in the order that they occur in the NAMELIST statement.

**Example:**

The following records were written out using the example program and input records from the namelist-directed READ section, above.

```
$VACATION
NAME    = 'Smith          ',
RATE    =   1.200000    ,
ACCRUED =   21.70000
$END
$VACATION
NAME    = 'Jones          ',
RATE    =   1.200000    ,
ACCRUED =   41.70000
$END
```

## 10.11. PARAMETER Statement

The PARAMETER statement is used to assign a symbolic name to a constant, compile-time constant expression or another symbolic name. A compile-time constant expression is one that is evaluated at compilation time, rather than calculated during program execution. Compile-time constants may be logical, character or arithmetic expressions.

**Syntax:**

```
PARAMETER  (symbol-name = expression [,symbol-name = expression]...)
```

-or-

```
PARAMETER  symbol-name = expression
```

where symbol-name is assigned the value of expression.

In the first form, one or more PARAMETER pairs may be assigned within a single PARAMETER statement, each delimited by commas and the entire statement enclosed in parenthesis. This form requires that the data type of the symbol be defined, either implicitly or explicitly, before the constant is referenced in the PARAMETER statement.

The second form of the PARAMETER statement is used when the symbolic name does not have a specific type that has been defined in a type declaration. The type of the symbolic name is taken from the type of the expression.

**Compile Time Expressions**

In the PARAMETER statement, expression may be a logical, character or arithmetic expression, subject to the following rules:

A logical expression is a compile-time constant expression if:

- Each operand is either: a constant; the symbolic name of a constant; one of the functions IAND, IEOR, IOR, ISHFT, LGE, LGT, LLE, LLT, with constant arguments; or another compile-time constant expression.
- Each operand is of a logical or integer data type.
- Each operator is a Boolean or a relational operator.

A character expression is a compile-time constant expression if:

- Each operand is either: a constant; the symbolic name of a constant; the function CHAR with a constant argument; or another compile-time constant expression.
- Each operand is of character data type.
- Each operator is the concatenation operator //.

An arithmetic expression is a compile-time constant expression if:

- Each operand is either: a constant; the symbolic name of a constant; one of the functions MIN, MAX, ABS, MOD, ICHAR, NINT, DIM, DPROD, CMPLX, CONJG, or AIMAG, with constant arguments; or another compile-time constant expression.
- Each operand is of an integer, real, or complex data type.
- Each operator is a +, -, *, /, or ** operator. (The exponentiation operator is evaluated at compile time only if the exponent is of integer data type.)

**Symbolic Names in Constant Expressions**

A symbolic name equated to a constant will assume a data type based on explicit type declarations preceding the PARAMETER statement, or by using the default conventions for determining implicit data types.

Symbolic constants are subject to the following rules:

- If the symbolic name is used as the length specifier in a CHARACTER statement, it must be enclosed in parentheses.
- The symbolic name of a constant cannot appear as part of another constant, except that it may appear as either the real or the imaginary part of a complex constant.
- A constant may be defined in a PARAMETER statement only once within a program or subroutine.

**Examples:**

In the first example, the first form of the PARAMETER statement is used to define a variety of constants. Note that this form requires that the data type be defined, either implicitly or explicitly, before the constant is referenced in a PARAMETER statement.

```
C     Note that ONE and ZERO are implicitly REAL*4
C     This form requires all data types to be pre-defined
      REAL*8    E, ESQ
      COMPLEX*8 I
      PARAMETER  (NAMELENGTH=10)
      CHARACTER *(NAMELENGTH) LASTNAME
      PARAMETER  (LASTNAME="KARAMAZOV ")
      PARAMETER  (ONE=1.0, ZERO=0.0)
      PARAMETER  ( I=(ZERO,ONE) )
      PARAMETER  ( E=2.7182818284D0 )
      PARAMETER  ( ESQ= E * E )
```

In the second example, the alternate form of the PARAMETER statement is used.

```
C     Alternate form of PARAMETER statement
C
C     Data types must NOT be pre-defined
      PARAMETER  LASTNAME="KARAMAZOV "
      PARAMETER  ONE=1.0, ZERO=0.0
      PARAMETER  I=(ZERO,ONE)
      PARAMETER  E=2.7182818284D0
      PARAMETER  (ESQ = E * E)
```

**F77 Compatibility Notes:**

The second form of the PARAMETER statement, used when the symbolic name does not have a specific type defined in a type declaration, is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 10.12. SAVE Statement

Execution of a RETURN or END statement in a procedure causes all program entities in that procedure to become undefined except for the following:

- entities in a blank (unnamed) common block
- entities that are initially defined but do not become redefined or undefined in the procedure
- entities in a named common block that appear in the procedure and in at least one other program unit that references the procedure

Use the SAVE statement to retain the definition status of a program entity following the execution of a RETURN or END statement.

**Syntax:**

```
SAVE  [symbolic-name [, symbolic-name] ... ]
```

A symbolic-name in a SAVE statement can represent a variable, an array, or a named common block. The name of a common block must be enclosed in slashes. A SAVE statement that does not contain explicit name specifications implies the saving of all appropriate program entities in the program unit that contains that SAVE statement.

Entities specified in SAVE statements for one program unit do not become undefined when a RETURN or END statement executes in that program unit. However, if the entities are in a common block, they might become undefined in another program unit.

You cannot use the names of procedures, entities within a common block, or dummy arguments in a SAVE statement.

**Examples:**

The following SAVE statement specifies that three program entities retain their definition status following the execution of a RETURN or END statement in the program unit. Note that the name of a common block

is enclosed in slashes.

```
SAVE  var1, array1, /block/
```

The next SAVE statement specifies the saving of all appropriate program entities in the program unit that contains the SAVE statement.

```
SAVE
```

## 10.13. VIRTUAL Statement

The VIRTUAL statement is equivalent to the DIMENSION statement, and has been included for compatibility with PDP-11 Fortran.

**F77 Compatibility Notes:**

The VIRTUAL statement is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 10.14. VOLATILE Statement

The VOLATILE statement is used to specify variables, arrays and common blocks that will not be subject to certain compiler optimizations. This allows storage to be retained for these items, if they are declared but not referenced in the program body.

**F77 Compatibility Notes:**

The VOLATILE statement is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

# Chapter 11
# Records and Structures

Records are data structures which are like arrays except that they enable you to group together disparate but related items. For example, you could group times and temperatures, or names and account balances, and so forth. Unlike arrays, the disparate items may even have different data types. The format or structure of a record is defined via a STRUCTURE statement.

Note that "record" in this context bears no relation to "record" in the context of external files which the user may read or write.

**F77 Compatibility Notes:**

Records and structures, including the RECORD, UNION and STRUCTURE statements, are not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 11.1. STRUCTURE Statement

A structure defines the form of a record, as a blueprint defines the form of a building. Just as several buildings may be built with different materials from the same blueprint, so several different records may be defined with the same structure but with different names and contents. Unfortunately, the term "structure" has long been ambiguous; in the computer field as in architecture, it has been used to refer both to the abstract shape (blueprint/template) and also to a specific entity based on that template. It should be stressed that the former meaning is the one used here.

**Syntax:**

```
STRUCTURE  /structure-name/
      structure definition statements consisting of:
            [ data type declarations ]
            [ PARAMETER statements ]
            [ sub-structure and mapped common statements ]
            [ union declarations ]
END STRUCTURE
```

where *structure-name* is a symbolic name that will be used to reference the structure.

One or more *structure definition statements* may be specified within the STRUCTURE block, and may appear in any order.

Data type declaration statements are allowed in STRUCTURE statements, subject to the following rules:

- All field names must be explicitly typed. Implicit typing, or typing via an IMPLICIT statement has no effect within a STRUCTURE.
- Any valid Fortran data type may be used.
- Array dimensions, if any, must be specified within the type statement. No DIMENSION statements are allowed.

- Adjustable size arrays and passed length CHARACTER declarations are not allowed.
- Field names within a single structure level must be unique, however names used in a substructure may duplicate those specified in an outer structural level.
- Initialization may not be done in a structure declaration.
- Record variables cannot be declared in a structure statement.
- Structure declarations may not be nested.

**Example:**

```
STRUCTURE  /ADDR_STRUCT/
      CHARACTER*30     STREET_ADDR
      CHARACTER*15     CITY
      CHARACTER*2      STATE
      INTEGER     ZIP_CODE
END STRUCTURE

STRUCTURE  /STUDENT_FILE/
      CHARACTER*25     NAME
      CHARACTER*15     MAJOR
      RECORD      /ADDR_STRUCT/    ADDRESS
END STRUCTURE
```

The following examples show invalid structure usage:

```
C     Illegal initialization within a structure declaration
C
      STRUCTURE /A1/
            INTEGER*2  I /100/
      END STRUCTURE
      RECORD      /A1/A1_A
...
C     Declaring record variables in a structure is not allowed
C
      STRUCTURE /A1/A1_A
            INTEGER*2  I
      END STRUCTURE
...
C     Structure declarations may not be nested
C
      STRUCTURE /A1/
            REAL*4      R4
            STRUCTURE /A2/A2_A
                  BYTE B
            END STRUCTURE
      END STRUCTURE
...
```

## 11.2. UNION Declarations

A union declaration allows you to define more than one set of fields within a STRUCTURE that can be used to reference a single data area. Unlike an EQUIVALENCE statement, union declarations allow a single data space to be used alternately. When fields of one map declaration are referenced, the other map declaration values become undefined.

**Syntax:**

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

where *field-declaration* is any valid STRUCTURE field definition statement.

**Example:**

The following example defines the structure DATA_BUFFER, which can alternately contain payroll data or vacation information. A structure of this form might be used to alternately read data from an employee payroll file, or from the vacation file.

```
STRUCTURE  /DATA_BUFFER/
      CHARACTER*20     LAST_NAME
      CHARACTER*15     FIRST_NAME
      INTEGER     EMPL_NBR
      UNION
            MAP
                  INTEGER     HOURLY_WAGE
                  INTEGER     YTD_EARNINGS
            END MAP
            MAP
                  INTEGER     VAC_RATE
                  INTEGER     VAC_DAYS
            END MAP
      END UNION
END STRUCTURE
```

## 11.3. RECORD Statement

The RECORD statement is used to associate a symbolic *record-name* with a template defined by a previous STRUCTURE statement. The resulting data item is declared as *aggregate*, rather than *scalar* data.

**Syntax:** RECORD /struct-name/record-list [,/struct-name/record-list...]

where struct-name has been previously defined in a STRUCTURE statement, and record-list consists of one or more variables and/or array names, separated by commas.

**Example:**

The following example defines two records using the structure definition STUDENT_FILE defined in the STRUCTURE example.

```
RECORD  /STUDENT_FILE/ UNDERGRADS(100), GRADUATES(100)
```

Using the RECORDs and STRUCTUREs defined above, the data for the 5th graduate would be referenced as follows:

```
GRADUATES(5).NAME      student's name
GRADUATES(5).MAJOR     student's major
GRADUATES(5).ADDRESS   student's address
```

or simply as

```
GRADUATES(5).
```

## 11.4. Using RECORDS and STRUCTURES

### 11.4.1. Record and Field References

Records consist of one or more fields, which are either ordinary variables or array elements or substructures themselves. Fields may be referenced either individually, or as an aggregate entity involving the entire record.

Records may be referenced by aggregate fields and/or scalar fields. An aggregate field reference refers to a single record or substructure. A scalar field reference refers to an individual variable or array.

**Syntax:** record-name[.aggr-name[.aggr-name ...][.scalar-field]

where *record-name* is the name used in a RECORD statement to identify a record; *aggr-name* is the name of a field that is a substructure (i.e., a record or a nested structure declaration) within the record structure specified by the record name; and *scalar-field* is the name of a typed data item defined within the structure declaration.

Note that since periods are used to delimit fields in record references, you should not define field names which, when set off with periods, signify relational operators (e.g., .EQ., .XOR.), logical constants (.TRUE., .FALSE.), or logical operators (.AND., .NOT., .OR.).

### 11.4.2. Aggregate Assignment Statement

Aggregate record assignments may be made when the aggregates to the left and right of the equals sign have the same structure. Using the example above, one could write

```
GRADUATE(5).NAME      = UNDERGRAD(47).NAME
GRADUATE(5).MAJOR     = UNDERGRAD(47).MAJOR
GRADUATE(5).ADDRESS   = UNDERGRAD(47).ADDRESS
```

or simply

```
GRADUATE(5)           = UNDERGRAD(47)
```

### 11.4.3. Scalar Field References

A scalar field reference refers to a single, typed data item and may be treated like an ordinary reference to a FORTRAN variable or array element. The type conversion rules for these references are the same as the rules for variables and array elements. Scalar field references may be used in statement functions and executable statements. However, they may not be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements.

### 11.4.4. Aggregate Field References in I/O Statements

Aggregate field references may be used in unformatted I/O statements (one I/O record is written no matter how many aggregate and array name references there are in the I/O list); but they may not be used in formatted or NAMELIST I/O statements.

# Chapter 12
# DATA Statement

The DATA statement assigns initial values to variables, arrays, array elements, and substrings. An initial value is a value assigned to a program entity at the beginning of program execution. An entity that is not assigned an initial value at the beginning of execution is recognized as undefined.

DATA statements are nonexecutable. You can use a DATA statement anywhere in a program unit after any specification statements that the program unit contains.

**Syntax:**

```
DATA  name-list /constant-list/ [[,] name-list /constant-list/] ...
```

A name-list consists of one or more variable names, array names, array element names, substring names, and implied DO lists. Use a substring name in a DATA statement to initialize a portion of a character string. Use an implied-DO loop in a DATA statement to initialize a portion of an array. You must separate the names in the name-list with commas.

You cannot use the names of functions, entities in blank common, or dummy arguments in a DATA statement name-list. You can use the names of entities in named common blocks in a DATA statement name-list if the DATA statement is in a block data subprogram.

A constant-list consists of constants, symbolic names for constants, and constants preceded by a factor. A constant preceded by a factor specifies multiple, successive appearances of the constant in a constant-list. Use the following form to write a constant preceded by a factor.

```
factor * constant
```

The factor must be a nonzero, unsigned integer constant or the symbolic name of such a constant. The constant that follows the asterisk can be zero, a signed or unsigned constant, or a symbolic name. The following examples show two equivalent constant-lists.

```
/3*100/
/100, 100, 100/
```

Note that you must delimit each constant-list with slashes and you must separate the constants in the constant-list with commas. You can optionally delimit each name-list/constant-list pair with commas.

The DATA statement assigns the constant values in each constant-list to the entities specified in the preceding name-list. The DATA statement assigns the values consecutively, one by one, as they appear in each list. The number of constants in a constant-list must correspond exactly to the number of entities specified in the preceding name-list. If you specify an unsubscripted array name in a name-list, you must specify a constant for each element of the array in the constant-list.

The DATA statement in the following example declares initial values for two variables, an array, and a substring.

**Example:**

```
INTEGER array1(3)
REAL var1
LOGICAL var2
CHARACTER subst*5
DATA var1, var2 /1.02E3,.TRUE./ array1,subst /3*100,'total'/
```

## 12.1. Type Conversion in DATA Statements

An entity in a name-list and the corresponding constant in a constant-list must both have either numeric or character data types. When the name-list entity and the corresponding constant have numeric data types that differ, the data type of the constant converts to the type of the name-list entity. Conversion takes place according to the FORTRAN rules for arithmetic conversion described in the "Assignment Statements" chapter.

When the length of a character entity in a name-list is greater than the corresponding character constant, the DATA statement initializes the extra characters in the entity with blanks. When the length of an entity in a name-list is less than the corresponding character constant, the DATA statement ignores the extra characters in the entity.

**Examples:**

In the following example, the real constant assigned to the integer variable, intvar, is converted to the integer 3. The integer constant 128 assigned to the real variable, realvar, is converted to the real number 128.0.

```
INTEGER intvar
REAL realvar
DATA intvar, realvar /3.14159,128/
```

In the next example, the DATA statement ignores the last character in the character constant Challenger because the character variable substr1 has a length of only 9 characters. The DATA statement initializes the last four characters in subst2 with blanks because the corresponding character constant, orbit, only has five characters.

```
CHARACTER*9 subst1, subst2
DATA subst1, subst2 /'Challenger','orbit'/
```

## 12.2. Implied-DO in DATA Statements

You can use an implied-DO loop in a DATA statement to initialize a portion of an array. The implied-DO loop repeats the initialization process in a DATA statement for the array elements that you specify.

**Syntax:**

```
(do-list, variable = initial,limit [,increment])
```

The do-list specifies the array elements that you want to initialize. The variable, referred to as the implied-DO variable, assumes each iteration value in the range specified using the initial and limit values. The

variable must be an integer variable. The iteration count specified with the initial and limit values must proceed in a positive direction. The optional increment value specifies an iteration step. If you omit the increment value, a value of 1 is implied.

**Examples:**

The DATA statement in the following example uses an implied-DO loop to initialize elements 10 through 20 of a 50 element integer array. The statement initializes a total of 11 elements with the constant 100.

```
INTEGER array(50)
DATA (array(i), i = 10, 20)/11*100/
```

The DATA statement in the next example uses an implied-DO loop to initialize every other element from (1,1) through (100,199) in a 20,000 element real array. Note the use of the increment value, 2, for iteration control over the second array dimension. The statement initializes a total of 10,000 elements with the real value 3.14159.

```
REAL array(100,200)
DATA ((array(k,m), k=1, 100), m=1, 200,2)/10000*3.14159/
```

# Chapter 13
# Assignment Statements

Assignment statements assign values to variables, arrays, array elements, and substrings. The assignment statement evaluates an expression and assigns the result of the evaluation to the entity. Once an assignment statement assigns a value to an entity, that entity is recognized as defined. There are four kinds of assignment statements in FORTRAN:

- arithmetic
- logical
- character
- ASSIGN

## 13.1. Arithmetic Assignment Statements

Use an arithmetic assignment statement to assign the value of an arithmetic expression to an arithmetic variable or array element. Place the arithmetic expression on the right side of an equal sign and the name of the arithmetic entity on the left as shown in the following syntax specification.

**Syntax:**

```
symbolic-name = arithmetic-expression
```

Upon execution, an arithmetic assignment statement first evaluates the arithmetic-expression according to the rules for expression evaluation described in Section 5. Then, the statement assigns the result of the evaluation to the entity that the symbolic-name identifies.

If the entity on the left of the equal sign has the same data type as the expression on the right, the statement assigns the value directly. If the data types differ, the value of the expression converts to the data type of the entity on the left before the assignment takes place. Assignment statements use certain FORTRAN intrinsic functions to perform the conversion of data types. The table below lists the generic intrinsic function names used for conversion in arithmetic assignment statements. Refer to the "FORTRAN Intrinsic Functions" chapter for additional information.

**Functions for Arithmetic Type Conversion**

| Variable or Array Element Data Type | Generic Intrinsic Function Name |
|---|---|
| integer | INT(expression) |
| real | REAL(expression) |
| double precision | DBLE(expression) |
| complex | CMPLX(expression) |

**Examples:**

The arithmetic assignment statement in the following example assigns the real constant 6.02E23 to the real variable avogadro.

```
REAL  avogadro
avogadro = 6.02E23
```

The next arithmetic assignment statement evaluates an expression, converts the result of the expression to an integer, and assigns the integer value to the first element in the integer array element ionic(25).

```
INTEGER  ionic(25)
ionic(1) = 28.43/14.32**(-3)
```

## 13.2. Logical Assignment Statements

Use a logical assignment statement to assign the value of a logical expression to a logical variable or array element. Place the logical expression on the right side of an equal sign and the name of the logical entity on the left, as shown in the following syntax specification.

**Syntax:**

```
symbolic-name = logical-expression
```

Upon execution, a logical assignment statement first evaluates the logical-expression. Then, the statement assigns the result of the evaluation to the entity that the symbolic-name identifies. Note that a logical expression must evaluate to a logical value, either true or false. Refer to the "Logical Expressions" section for additional information.

**Examples:**

The following logical assignment statement assigns the logical constant .false. to the logical variable switch.

```
switch = .false.
```

The next logical assignment statement first evaluates a logical expression, then assigns the result to the logical variable prnout.

```
prnout = var1/var7 .GT. 128 .AND. var2/var8 .LT. 128
```

## 13.3. Character Assignment Statements

Use a character assignment statement to assign the value of a character expression to a character variable, array element, or substring. Place the character expression on the right side of an equal sign and the name of the character entity on the left as shown in the following syntax specification.

**Syntax:**

```
symbolic-name = character-expression
```

If the length of the character-expression is less than the length of the character entity, the statement pads the expression on the right with blank characters.

If the length of the character-expression is greater than the length of the character entity, the statement truncates the expression from the right.

Assigning a value to a character substring does not affect any characters in the character variable or array element that are outside the substring reference. Any character positions in a character entity that are outside the substring reference remain unchanged whether or not the positions were defined or undefined.

**Examples:**

The character assignment statement in the following example assigns the character constant Sirius to the character variable starname.

```
CHARACTER  starname*12
starname = 'Sirius'
```

The character assignment statement in the next example evaluates a character expression, then assigns the result to the character array element compound(5).

```
CHARACTER*8  compound(25)
compound(5) = 'Na' // 'Cl'
```

The character assignment statement in the last example assigns the character constant mix67 to the substring var1(2:6). Note that character positions in the character variable var1 that are outside of the substring reference remain unchanged after the assignment takes place.

```
CHARACTER  var1*7
var1(2:6) = 'mix67'
```

## 13.4. ASSIGN Statement

Use the ASSIGN statement to assign a statement label value to an integer variable. This enables the variable name to be used as a transfer specification in an assigned GOTO statement, or as a format specifier in a formatted I/O statement.

**Syntax:**

```
ASSIGN  statement-label  TO  symbolic-name
```

The statement-label must identify an executable statement or a FORMAT statement. The executable statement or the FORMAT statement must be in the same program unit as the ASSIGN statement.

The symbolic-name must represent an integer variable. Once the variable becomes defined for reference as a statement label, it becomes undefined for use as an integer variable.

The ASSIGN statement must execute before any statements containing a reference to the assigned variable name. You cannot specify arithmetic operations involving a variable that represents a statement label.

**Examples:**

The following ASSIGN statement assigns the statement label 300 to the integer variable trans. The statement defines trans as a statement label variable.

```
INTEGER  trans
ASSIGN  300  TO  trans
```

You can redefine trans in another assignment statement. The following statement returns trans to its status as an integer variable. After this statement executes, you can no longer use trans in an assigned GOTO statement.

```
trans = 2149
```

# Chapter 14
# Control Statements

Normally, statements in a program execute sequentially, in the order that you write them. Control statements specify changes to the sequential flow of statement execution. You can use a control statement to transfer the flow of execution to a point within the same program unit or to a point in a different program unit.

Some control statements change the flow of execution depending on a condition that is determined at that point in the flow of execution. Other control statements transfer the flow of execution every time that particular control statement executes, regardless of any condition.

The following control statements are supported by FORTRAN. Note that many of these are variations of the GOTO and IF statements.

- Unconditional GOTO Statement
- Computed GOTO Statement
- Assigned GOTO Statement
- Arithmetic IF Statement
- Logical IF Statement
- Block IF Statement
- ELSE IF Statement
- ELSE Statement
- END DO Statement
- END IF Statement
- DO Statement
- DO WHILE Statement
- CONTINUE Statement
- STOP Statement
- PAUSE Statement
- END Statement
- CALL Statement
- RETURN Statement

## 14.1. Arithmetic IF Statement

Use an arithmetic IF statement to transfer control to one of three executable statements that you specify. The executable statements must be in same program unit as the arithmetic IF statement. The value of an arithmetic expression determines which of the three statements receives control.

**Syntax:**

```
IF (arithmetic-expression) 1st-label, 2nd-label, 3rd-label
```

The value of the arithmetic-expression determines which of the three statements receives control. If the arithmetic-expression evaluates to a number less than 0, control transfers to the executable statement that the 1st-label identifies. If the arithmetic-expression evaluates to a number equal to 0, control transfers to the executable statement that the 2nd-label identifies. If the arithmetic-expression evaluates to a number greater than 0, control transfers to the executable statement that the 3rd-label identifies.

All three labels are required, but they do not have to identify three different statements.

**Examples:**

The following arithmetic IF statement transfers control to an executable statement with the statement label 7514 if the integer variable num is greater than to 0. If num is equal to zero, control transfers to statement 3500. Control transfers to statement 2000 if num is less than 0.

```
IF (num)  2000, 3500, 7514
```

The following arithmetic IF statement transfers control to statement 120 if the two variables in the expression have different values. Control transfers to statement 150 if the two variables have the same value.

```
IF (total-value)  120, 150, 120
```

## 14.2. Assigned GOTO Statement

Use an assigned GOTO statement to transfer control to an executable statement identified with a variable. The value assigned to the variable must represent the statement label of the executable statement that is to receive control. The executable statement that the variable identifies must be in the same program unit as the assigned GOTO statement. You assign a statement label to an integer variable using the ASSIGN statement. Refer to the "ASSIGN Statement" section for more information.

**Syntax:**

```
GOTO  variable-name[[,] (statement-label-list)]
```

The variable-name must have an integer data type and must have an assigned statement label value before the GOTO statement executes. If a program unit contains more than one ASSIGN statement for the variable-name in an assigned GOTO statement, the most recently executed ASSIGN statement determines the value of the variable. You can transfer control to different executable statements using multiple ASSIGN statements. An assigned GOTO statement and any related ASSIGN statements must be in the same program unit.

The optional statement-label-list contains all the statement labels to which the assigned GOTO statement can transfer control. If you specify the list, your program can compare an assigned value with the values in the list. If the program does not find the assigned value in the list, the program can interrupt execution and issue an error message. If you specify the list, you must include all valid statement labels.

**Examples:**

The assigned GOTO statement in the following example transfers control to an executable statement with the statement label 810.

```
ASSIGN  810  TO  inert
GOTO  inert
```

The assigned GOTO statement in the next example transfers control to an executable statement with the statement label 464. Note the use of the statement-label-list. The program could be designed to report an error if the program cannot find the assigned value in the list. The assigned value 464 is in the list. Therefore, no error would be detected.

```
ASSIGN 464 TO kalend
GOTO  kalend (312, 1012, 464, 2000)
```

## 14.3. Block IF Statement

A block IF statement indicates the beginning of a block IF construct. A block in a block IF construct is a sequence of FORTRAN statements designed to execute under specified conditions. The decision to execute a statement block in a block IF construct depends on a logical expression.

A block IF construct utilizes the ELSE, ELSE IF, and END IF statements in addition to the block IF statement to control the flow of execution.

**Syntax:**

```
IF (logical-expression) THEN
   .
   .
   statement-block
   .
   .
END IF
```

If the value of the logical-expression is true, control transfers to the statement-block. A statement-block can be empty. If the value of the logical-expression is false, control transfers to the first executable statement following the END IF.

Note that each block IF statement that you specify must have a corresponding END IF statement. The END IF statement identifies the end of a block IF construct.

**Example:**

The following example shows a block IF construct. The block IF statement that identifies the beginning of the construct contains an expression that evaluates to true. Therefore, the specified statement-block executes.

```
smallnum = 100/2
largenum = 100*2
IF (smallnum .LT.  largenum) THEN
   pi = 3.14159
   radius = smallnum
   area = pi * radius ** 2
END IF
```

## 14.4. CALL Statement

Use the CALL statement to transfer execution control to a subroutine. The CALL statement specifies the symbolic name of the subroutine that you want to invoke and a list of actual arguments to pass to the subroutine.

**Syntax:**

```
CALL  symbolic-name [(actual[, actual] ... )]
```

Actual arguments specified in the CALL statement must correspond in number, order, and data type to the dummy arguments specified in the subroutine. Actual arguments can be any one of the following items.

- an expression
- an array name
- an intrinsic function
- an external procedure name
- a dummy procedure name
- an alternate return specifier using the statement label of an executable statement in the same program unit as the CALL statement

In the following example, the CALL statement calls a subroutine named graph and passes three actual arguments.

**Example:**

```
CALL graph (vertical, horizontal, number)
```

## 14.5. Computed GOTO Statement

Use a computed GOTO statement to transfer execution control to one of a specified list of executable statements. The executable statements that you specify must be in same program unit as the computed GOTO statement. The value of an arithmetic expression determines which statement in the specified list receives control.

**Syntax:**

```
GOTO  (statement-label-list)[,] arithmetic-expression
```

The statement-label-list contains one or more statement labels that identify executable statements. Separate the labels in the list with commas. You can refer to the statement-label-list as the transfer list.

A computed GOTO statement converts the value of the arithmetic-expression to an integer, if necessary. The value of the arithmetic-expression specifies a number that corresponds to the position of a particular statement label in the statement-label-list.

If the value of the arithmetic-expression is less than 1 or greater than the number of statement labels in the statement-label-list, execution control transfers to the first executable statement that follows the computed GOTO statement in the program.

**Examples:**

The following computed GOTO statement selects one of four statement labels for the transfer of execution control depending on the value of the variable num.

```
GOTO (54, 166, 418, 500), num
```

The next computed GOTO statement selects one of six statement labels for the transfer of execution control depending on the value of the three variables: velocity, t1, and t2.

```
GOTO (250, 500, 1000, 2000, 2500, 3000) velocity/t1-t2
```

## 14.6. CONTINUE Statement

The CONTINUE statement simply transfers execution control to the next executable statement in the program. Use the CONTINUE statement as the terminal statement for a DO loop that would otherwise end incorrectly with a control statement, such as an arithmetic IF, ELSE IF, or RETURN statement.

**Syntax:**

```
CONTINUE
```

The following example shows a DO loop that would end incorrectly with an arithmetic IF statement. The CONTINUE statement enables the DO loop to end properly.

**Example:**

```
      DO 211 var1 = 1, 5
              .
              .
              .
      IF (tax/2 + 4.25) 300, 400, 500
211   CONTINUE
```

## 14.7. DO Statement

Use the DO statement to specify a block of statements for loop processing. Loop processing is the repetitious execution of a statement block a specified number of times.

The DO statement indicates the beginning of a DO statement block. You must specify a terminal statement using a statement label in the DO statement to indicate the end of the block. All statements between the DO statement and the terminal statement, inclusive, define the range of a DO loop.

A variable and three expressions in the DO statement control the iteration count for a DO loop. The iteration count is the number of times a DO statement block is designed to execute or loop.

**Syntax:**

```
DO statement-label [,] variable = 1st-exp, 2nd-exp, [3rd-exp]
```

The statement-label identifies the terminal statement. The terminal statement cannot be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement.

The variable can be an integer, real, or double precision variable. You can refer to this variable as the control variable or the DO variable. The value of the variable at any given time during execution of the DO loop depends on the values of the three expressions.

1st-exp, 2nd-exp, and 3rd-exp can be integer, real, or double precision expressions. 1st-exp is the initial value of the DO variable. 2nd-exp is the limit value of the DO variable. The optional 3rd-exp is an increment value that specifies how much the DO variable is incremented after each execution of the DO loop. If you omit 3rd-exp, an increment value of 1 is implied.

The DO statement evaluates the expressions first and converts the values of the expressions to the same data type as the DO variable, if necessary. The value of the 1st-exp is assigned to the DO variable. Then, the iteration count is calculated using the intrinsic functions MAX and INT, as shown below.

```
MAX(INT((2nd-exp - 1st-exp + 3rd-exp) / 3rd-exp), 0)
```

If the iteration count is greater than zero, the statements within the range of the DO loop execute. Following the first execution of the DO loop, the value of the DO variable is incremented according to the specified increment value. The iteration count is computed again and tested to see if it is greater than zero. If the iteration count is greater than zero, the statements within the range of the DO loop execute again. If the iteration count is negative or equal to zero, execution of the DO loop stops. Execution of the program continues with the first executable statement that follows the terminal statement of the DO loop.

You can nest DO loops; however, the range of a nested (inner) DO loop must lie completely within the range of the host (outer) DO loop. Nested DO loops can use the same terminal statement.

If you use a DO statement within the statement block of a block IF, ELSEIF, or ELSE statement, the range of the corresponding DO loop must lie completely within that statement block. If you use a block IF statement within the range of a DO loop, the corresponding END IF statement must lie within the range of the DO loop.

**Examples:**

The following DO statement specifies 10 iterations of a DO loop. Note that an increment value is not specified. Therefore, an increment value of 1 is implied. The statement identified with the statement label 130 is the terminal statement for the loop.

```
DO 130 var = 1, 10
```

The next DO statement specifies 50 iterations of a DO loop. Note the use of a specified increment value.

```
DO 2100 var1 = 1, 100, 2
```

The next DO statement specifies 8 iterations of a DO loop. Note the use of the negative increment value.

```
DO 623 var2 = 16, 1, -2
```

The last DO statement demonstrates the use of more complicated expressions.

```
DO 599 var3 = value1*2+1, value2-var4/2, incr/2
```

### 14.7.1. Extended Range DO Loops

The range of a DO loop is extended if it contains a statement, *s1*, that transfers control outside of the loop, and, after executing one or more statements, control is returned to the loop via statement *s1*. The range of the DO loop is extended to include all executable statements outside of the initial loop between *s1* and *s2*.

Extended range DO loops are subject to the following rules:

- Transfers into the range of a DO statement may only be made from within the extended range of that DO statement.
- No statement in the extended range of a DO statement may alter the control variable of the DO statement.

**Examples:**

```
C     Valid extended DO loop
C
      DO 10, i=1,10
      ...
      GO TO 200
10    CONTINUE
      ...
200   j = j + 1
      ...
      GO TO 10
      ...
```

**F77 Compatibility Notes:**

Extended range DO loops are not supported in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 14.8. DO WHILE Statement

The DO WHILE statement is similar to the DO statement, except that DO WHILE will execute a loop while a logical expression remains true, rather than for a fixed number of iterations.

**Syntax:**

```
DO [statement-label[,]] WHILE (expression)
```

where *statement-label* identifies the terminating statement and *expression* is a logical expression delimited by parentheses. If the optional *statement-label* is not supplied, you must use END DO as the terminating statement for the loop. As with the DO statement, *statement-label* must be a valid DO loop terminator, and may not be a GOTO, IF, RETURN, STOP, END or DO statement.

Before each execution of a DO WHILE loop, *expression* is evaluated. If the logical expression is true, the statements within the loop are executed. If the logical expression is false, control is transferred to the first executable statement following *statement-label*.

**F77 Compatibility Notes:**

The DO WHILE statement is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 14.9. END Statement

Use the END statement to indicate the end of a program unit. The END statement must be the last statement in every program unit.

**Syntax:**

```
END
```

When execution control reaches the END statement in a main program, program execution terminates. When execution control reaches the END statement in a subprogram, control returns to the main program. An END statement in a subprogram works like a RETURN statement.

## 14.10. END DO Statement

The END DO statement is used to terminate a DO or DO WHILE loop. END DO is an alternative terminator if a terminating statement label was not supplied with a DO or DO WHILE statement. The END DO statement may also be used in conjunction with a terminating statement label specification, just as a CONTINUE statement is used.

**Syntax:**

```
END DO
```

**Example:**

```
      DO WHILE (i .LT.  10)
      ...
      k = i + 1
      END DO
-or-

      DO j=1,10
      ...
10    END DO
```

**F77 Compatibility Notes:**

The END DO statement is not supported in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 14.11. END IF Statement

An END IF statement indicates the end of a block IF construct.

**Syntax:**

```
   .
   .
END IF
```

Each block IF statement that you use to specify a block IF construct must have a corresponding END IF statement to indicate the end of the construct.

**Examples:**

```
IF (smallnum .GT.  largenum) THEN
     STOP
ELSE IF (smallnum .LT.  largenum) THEN
     total = largenum - smallnum
ELSE IF (smallnum .EQ.  largenum) THEN
     ASSIGN 533 TO equality
     GOTO equality (133, 333, 533)
END IF
```

## 14.12. ELSE Statement

An ELSE statement specifies an additional statement block for conditional execution within a block IF construct. Unlike block IF and ELSE IF statements, an ELSE statement does not contain a logical expression to determine whether or not control transfers to the specified statement-block. A statement-block that corresponds to an ELSE statement executes only if no preceding statement-block in the block IF construct executes.

**Syntax:**

```
   .
   .
ELSE
   .
   .
   statement-block
   .
   .
```

An ELSE statement and its corresponding statement-block must follow a block IF statement and its corresponding statement-block or an ELSE IF statement and its corresponding statement-block. No other ELSE, or ELSE IF statements can follow an ELSE statement in a block IF construct. An END IF statement can follow an ELSE statement.

In the following example, the statement-block that corresponds to the ELSE statement in the block IF construct executes because the logical expression in the opening block IF statement evaluates to false.

**Example:**

```
largenum = 712
smallnum = 4
IF (largenum .LT.  smallnum) THEN
      factor = smallnum/2 - 2.612
      unit5 = val1 + val2 * factor
ELSE
      ASSIGN 2001 TO standard
      GOTO standard (2001)
END IF
```

## 14.13. ELSE IF Statement

An ELSE IF statement specifies an additional statement block for conditional execution within a block IF construct. Like a block IF statement that indicates the beginning of a block IF construct, an ELSE IF statement contains a logical expression to determine whether or not control transfers to the specified statement-block. A block IF construct can contain any number of ELSE IF statements.

An ELSE IF statement and its corresponding statement-block must follow a block IF statement and its corresponding statement-block.

**Syntax:**

```
   .
   .
ELSE IF (logical-expression) THEN
   .
   .
   statement-block
   .
   .
```

If the value of the logical-expression is true, control transfers to the statement-block. A statement-block can be empty. If the value of the logical-expression is false, control transfers to the next ELSE IF or ELSE statement, or the first executable statement following the END IF statement in that construct.

In the following example, there are two ELSE IF statements in the block IF construct. The logical-expression in the opening block IF statement is false. Therefore, control passes to the first ELSE IF statement. The logical-expression in the first ELSE IF statement is also false. Therefore, control passes to the second ELSE IF statement. The logical-expression in the second ELSE IF statement is true. Therefore, the statement block specified after the second ELSE IF statement executes.

**Example:**

```
smallnum = 56000
largenum = 56000
IF (smallnum .GT.  largenum) THEN
      STOP
```

```
ELSE IF (smallnum .LT.  largenum) THEN
     total = largenum - smallnum
ELSE IF (smallnum .EQ.  largenum) THEN
     ASSIGN 533 TO equality
     GOTO equality (133, 333, 533)
END IF
```

## 14.14. Logical IF Statement

A logical IF statement conditionally executes a single FORTRAN statement that you specify literally within the logical IF statement. The decision to execute the statement is based on the value of a logical expression.

**Syntax:**

```
IF (logical-expression) statement
```

Note that the statement is a complete, literal statement specification and not a statement label. If the value of the logical-expression is true, control transfers to the statement. If the value of the logical-expression is false, control transfers to the first executable statement following the logical IF statement in the program.

**Examples:**

The value of the logical-expression in the following logical IF statement is true. Therefore, the STOP statement specified in the logical IF statement executes.

```
largenum = 427
smallnum = 8.602
IF (largenum .GT.  smallnum) STOP
```

The value of the logical-expression in the next logical IF statement is false. Therefore, the RETURN statement specified in the logical IF statement does not execute.

```
largenum = 427
smallnum = 8.602
IF (smallnum .GT.  largenum) RETURN
```

## 14.15. PAUSE Statement

Use the PAUSE statement to temporarily suspend program execution. You can specify an optional message that displays on the console screen just before the program pauses.

**Syntax:**

```
PAUSE [display-message]
```

The optional display-message can be a character constant or a digit string of five digits or less. The message can serve as a prompt for user input from the console. If you specify a display-message, the PAUSE statement displays the message on the console screen, suspends program execution, and waits for user response from the console.

In the following example, the first PAUSE statement simply suspends program execution and waits for a response from the console. The second PAUSE statement displays the message TYPE ANY KEY TO CONTINUE before suspending execution.

**Examples:**

```
PAUSE
PAUSE 'TYPE ANY KEY TO CONTINUE'
```

## 14.16. RETURN Statement

Use a RETURN statement to terminate execution of a procedure and to transfer execution control back to the program unit that references the procedure. Procedures can contain any number of individual routines with multiple entry and return points. The RETURN statement serves as an intermediate termination point in a procedure that contains more than one individual routine. A procedure can contain any number of RETURN statements.

There are two forms of the RETURN statement. The first form is for use in a function procedure. The second form is for use in a subroutine procedure.

**Syntax:**

```
RETURN
RETURN  [integer-expression]
```

Execution of a RETURN statement in a function procedure returns execution control to the program unit that referenced the function. The value of the function must be defined before the RETURN statement executes.

Execution of a RETURN statement in a subroutine procedure returns execution control to the program unit that called the subroutine. Use the optional integer expression to select an alternate return specifier. The integer expression indicates which alternate return asterisk in the SUBROUTINE or ENTRY statement dummy argument list to use for the return. A valid integer expression must be greater than or equal to 1 and less than or equal to the number of asterisks specified in the SUBROUTINE or ENTRY statement dummy argument list. Each asterisk in the dummy argument list corresponds to an actual argument supplied in the CALL statement that invokes the subroutine. Actual arguments are statement numbers that indicate the alternate return points. Refer to the "Alternate Return Specifiers" section for more information.

Execution of a RETURN statement in a procedure causes all program entities in that procedure to become undefined except for the following:

- entities in a blank (unnamed) common block
- entities that are initially defined but do not become redefined or undefined in the procedure
- entities in a named common block that appear in the procedure and in at least one other program unit that references the procedure

You can use the SAVE statement to retain the definition status of any program entity following the execution of a RETURN or END statement.

The RETURN statement is not required to terminate a procedure. The END statement declares the physical end of a procedure subprogram. An END statement used in a procedure has the same effect as a RETURN statement.

## 14.17. STOP Statement

Use the STOP statement to terminate program execution.

**Syntax:**

```
STOP [display-message]
```

The optional display-message can be a character constant or a digit string of five digits or less. If you specify a display-message, the STOP statement displays the message on the console screen, terminates program execution, and returns control to the operating system.

The first STOP statement in the example below terminates program execution and returns control to the operating system. The second STOP statement example displays the message END OF PROGRAM before terminating execution.

**Example:**

```
STOP
STOP 'END OF PROGRAM'
```

## 14.18. Unconditional GOTO Statement

Use an unconditional GOTO statement to transfer execution control to an executable statement. The executable statement must be in the same program unit as the unconditional GOTO statement. An unconditional GOTO statement transfers control to the same statement each time it executes.

**Syntax:**

```
GOTO  statement-label
```

The unconditional GOTO statement transfers control to the statement identified by the statement-label. The statement-label must identify an executable statement that is in the same program unit as the GOTO statement.

In the example below, the unconditional GOTO statement transfers control to an executable statement with the statement label 2189. Every time the statement executes, it sends control to the same executable statement.

**Example:**

```
GOTO  2189
```

# Chapter 15
# Input/Output Statements

FORTRAN I/O statements transfer data from one storage location to another within a processor. They also transfer data between a processor and any external device such as a console, printer, or a storage medium like a disk, floppy disk, or magnetic tape.

FORTRAN provides three categories of I/O statements:

- data transfer statements
- file positioning statements
- auxiliary I/O statements

Data transfer statements move data between internal (processor) storage and some external medium. The data transfer statements are

- ACCEPT
- DECODE
- ENCODE
- PRINT
- READ
- TYPE
- WRITE

File positioning statements manipulate the position of the internal file pointer relative to a specific file. The file positioning statements are

- BACKSPACE
- ENDFILE
- REWIND

Auxiliary I/O statements manipulate the connection of I/O units to external media, and inquire about the characteristics of a particular connection. The auxiliary I/O statements are

- CLOSE
- INQUIRE
- OPEN

This chapter describes the syntax of each I/O statement in detail. The "FORTRAN Input/Output System" chapter describes records, files, I/O units, and the I/O system in general. For reference in this section, the statement descriptions are listed alphabetically.

## 15.1. ACCEPT Statement

The ACCEPT statement transfers data from standard input to the variable or list of variables specified.

**Syntax:**

```
ACCEPT format-spec [,iolist]
-or- ACCEPT * [,iolist]
-or- ACCEPT group
```

where format-spec is a numeric format specifier; * implies list-directed input; group is a NAMELIST group specifier; and iolist is a list of elements, separated by commas, to be input. All input is from the implicit unit, standard input.

**Example:**

```
      CHARACTER*10 NAMEA, NAMEB, NAMEC
      NAMELIST     /MYGROUP/ A,B,C
      ACCEPT 1000, NAMEA, NAMEB
      ACCEPT *, NAMEC
      ACCEPT MYGROUP
      ...
1000  FORMAT (2A10)
```

**F77 Compatibility Notes:**

The ACCEPT statement is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 15.2. BACKSPACE Statement

The BACKSPACE statement moves the file pointer to the beginning of the preceding record.

**Syntax:**

```
BACKSPACE Unit-number
BACKSPACE(argument-list)
```

In the first form, Unit-number is an integer expression whose value is in the range 0 to 99. In the second form, argument-list is a list of specifiers that control the positioning process.

With either form, if the file has no preceding record, the I/O system ignores the BACKSPACE statement. If the preceding record is an endfile record, the BACKSPACE statement moves the file pointer to the beginning of the endfile record.

### The Argument-list

The argument list of the BACKSPACE statement consists of a comma separated list of one or more of the I/O specifiers listed below.

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer in the range 0 to 99 that specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or integer array element. The I/O system assigns IO-status the following values based on the outcome of the positioning:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the BACKSPACE statement. If an error occurs while processing the BACKSPACE statement, the following actions occur:

1. The BACKSPACE operation is terminated.
2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, REWIND, INQUIRE, or BACKSPACE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

### Restrictions

- The BACKSPACE statement cannot reference an internal file.
- The file must be connected for sequential access.
- In the form BACKSPACE(argument-list), you must include an I/O unit specifier.
- You cannot use the BACKSPACE statement with a record that was written using list-directed formatting.

### Examples

```
BACKSPACE 2
BACKSPACE(2,IOSTAT=errorflag,ERR=999)
```

## 15.3. CLOSE Statement

The CLOSE statement disconnects a file from an I/O unit. After the I/O unit is disconnected, a subsequent OPEN statement can connect it to the same file or a different file in the same program. Also, if the CLOSE statement disconnects a file, a subsequent OPEN statement can connect it to the same I/O unit or a different I/O unit, if the file still exists.

**Syntax:**

```
CLOSE ([UNIT=]Unit-number, [Close-list])
```

where Unit-number is an integer value between 0 and 99 that specifies an external I/O unit and Close-list is a comma separated list of one or more specifiers that control the close operation.

The keyword UNIT = is optional if and only if Unit-number is the first item specified in the list.

**The Close-list**

The following table summarizes the CLOSE keywords accepted by Fortran. A comment of 'Ignored' indicates that the keyword or value is accepted by the compiler, but is ignored. Information on keywords accepted but ignored by the compiler is included for VAX/VMS compatibility considerations.

Detailed descriptions for each supported keyword follow the table, and are presented alphabetically.

| Keyword | Value | Default/Comment |
|---|---|---|
| DISPOSE | | Equivalent to STATUS |
| DISP | | Equivalent to STATUS |
| STATUS | 'KEEP' | 'KEEP' |
| | 'SAVE' | |
| | 'DELETE' | |
| | 'PRINT' | Ignored |
| | 'PRINT/DELETE' | Equivalent to 'DELETE' |
| | "SUBMIT' | Ignored |
| | 'SUBMIT/DELETE' | Equivalent to 'DELETE' |
| ERR | error-label | No default |
| IOSTAT | INTEGER | No default |
| UNIT | INTEGER | Values 0-99 only |

**File Disposition Specifier**

DISPOSE = Disposition-type

The DISPOSE (DISP) keyword is used to specify the disposition of a file when the associated logical unit is closed. The keywords DISPOSE and DISP are equivalent when used in a CLOSE-list.

Disposition-type may be one of the following values:

'KEEP' or 'SAVE'
'DELETE' or 'PRINT/DELETE' or 'SUBMIT/DELETE'

If you omit DISPOSE, the default is 'KEEP', which will retain the file after closure. Disposition-type 'KEEP' and 'SAVE' are equivalent.

If you specify DISPOSE = 'DELETE', the file is deleted after the unit is closed. Note that 'DELETE', 'PRINT/DELETE' and 'SUBMIT/DELETE' are equivalent.

Disposition exceptions occur if an attempt is made to save a scratch file, or to delete a read-only file. In these circumstances the conflicting Disposition-type is ignored.

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program I/O unit as the CLOSE statement. If the I/O system encounters an error while processing the CLOSE statement, the following actions occur:

1. The CLOSE operation is terminated.
2. The file position becomes undefined, unless the error is an end-of-file condition. Then, the file pointer points just past the endfile record, and the only valid statements that you can execute are BACKSPACE, REWIND, or INQUIRE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer or an element of an integer array. The I/O system assigns IO-status one of the following values based on the outcome of the close process:

```
IO-status =  0 if no error has occurred
IO-status ≥ 1 if an error has occurred
IO-status = -1 if an end-of-file condition has occurred
```

Status Specifier

STATUS = Status

Status is a character string expression, partially duplicating the DISPOSE= specifier, whose value must be one of the following:

'KEEP' or 'DELETE'

If you omit STATUS, the I/O system supplies the default STATUS = 'KEEP' unless the file already has a status of SCRATCH, in which case the default is STATUS = 'DELETE'.

**Restrictions**

- The CLOSE statement need not occur in the same program I/O unit as its corresponding OPEN statement.
- If the specified file does not exist, the I/O system ignores the CLOSE statement.
- If STATUS = 'KEEP' the file continues to exist after the I/O system executes the CLOSE statement.
- If STATUS = 'DELETE', the file does not exist after the I/O system executes the CLOSE statement.
- When a program terminates normally, the I/O system automatically closes all connected I/O units, and deletes all files with STATUS = 'SCRATCH'.

**Example:**

```
CLOSE(3)
CLOSE(3,IOSTAT=errorflag,ERR=999,STATUS='KEEP')
```

**F77 Compatibility Notes:**

The following CLOSE keywords are part of the VMS extensions, and therefore are not available when using F77 compatibility mode.

```
DISPOSE    DISP
```

Please refer to your User's Guide for details on compiler options and default modes.

## 15.4. DECODE Statement

The DECODE statement transfers data from external character form to internal binary representation, using a format specification. DECODE is functionally equivalent to using a READ statement with formatted records on an internal file connected for sequential access. However, when an array name is specified for the internal file, the entire array is considered a single record with the length of the entire array, rather than being interpreted as a series of records one for each array element.

**Syntax:**

```
DECODE(control-list)[transfer-list]
```

where control-list is a comma separated list of specifiers that control the transfer process, and transfer-list is the list of variables that receive the data after translation to internal binary representation.

**The Control-list**

Characters Specifier

Num-chars

Num-chars is an integer expression that designates the number of characters to translate to internal binary representation. Num-chars must be the first specifier in the control-list.

Format Specifier

Format

Format is a format identifier that controls the editing of the data during the transfer. Refer to "The FORMAT Statement and Format Specifications" chapter for more information. Format must be the second specifier in the control-list.

Location Specifier

Location

Location is the name of a variable, an array, or an array element that contains the characters to translate to internal binary representation. Location must be the third specifier in the control-list.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or an element of an integer array. The I/O system assigns IO-status one of the following values based on the outcome of the data transfer:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred
IO-status = –1 if an end-of-string condition has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the DECODE statement. If the I/O system encounters an error while processing the DECODE statement, the following actions occur:

1. The DECODE operation is terminated.
2. The I/O system sets the IO-status (if one is specified).
3. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

**Restrictions**

- If Location is an array, DECODE processes the elements in normal column-major order.
- The process of format control is the same as for a formatted READ statement.
- The number of characters that DECODE can process depends on the data type of location. A character variable or a character array element can contain the same number of characters as its declared length, but a character array can contain a number of characters equal to the length of each element times the number of elements. For example, each element of an INTEGER*4 array can contain four characters, so the total number of characters DECODE can process is four times the number of array elements.

**Examples**

```
DECODE (3,Z,100) var1, var2
DECODE (ichar, '3I4', block1) ivar1, ivar2, ivar3
```

## 15.5. ENCODE Statement

The ENCODE statement transfers data from internal binary representation to external character form, using a format specification. ENCODE is functionally equivalent to using a WRITE statement with formatted records on an internal file connected for sequential access, unless an array name is specified for the internal file. In that case, the entire array is treated as a single record with the length of the entire array, rather than being interpreted as a series of records, one for each array element.

**Syntax:**

```
ENCODE(control-list)[transfer-list]
```

where control-list is a comma separated list of specifiers that control the transfer process as described below, and transfer-list is the list of variables to translate to an internal binary representation.

**The Control-list**

Characters Specifier

Num-chars

Num-chars is an integer expression that designates the number of characters to translate to external form. Num-chars must be the first specifier in the control-list.

Format Specifier

Format

Format is a format identifier that controls the editing of the data during the transfer. Refer to the "Format Statement and Format Specifications" chapter for more information. Format must be the second specifier in the control-list.

Location Specifier

Location

Location is the name of a variable, an array, or an array element that contains the characters after translation to external form. Location must be the third specifier in the control-list.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or an element of an integer array. The I/O system assigns IO-status one of the following values based on the outcome of the data transfer:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the ENCODE statement. If the I/O system encounters an error while processing the ENCODE statement, the following actions occur:

1. The ENCODE operation is terminated.
2. The I/O system sets the IO-status (if one is specified).
3. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

**Restrictions**

- If Location is an array, ENCODE processes the elements in normal column-major order.
- The process of format control is the same as for a formatted WRITE statement.
- The number of characters that ENCODE can process depends on the data type of location. A character variable or a character array element can contain the same number of characters as its declared length, but a character array can contain a number of characters equal to the length of each element times the number of elements. For example, each element of an INTEGER*4 array can contain four characters, so the total number of characters ENCODE can process is four times the number of array elements.

**Examples**

```
ENCODE (3,Z,arr1) var1
ENCODE (ichar,'(2I5,F7.2)',arr2) ivar1, ivar2, var3
```

## 15.6. ENDFILE Statement

The ENDFILE statement causes the I/O system to write an endfile record as the next record of the file.

**Syntax:**

```
ENDFILE Unit-number
ENDFILE(argument-list)
```

In the first form, Unit-number is an integer expression whose value is in the range 0 to 99. In the second form, argument-list is a comma separated list of specifiers that control the positioning process.

After executing the ENDFILE statement, the I/O system cannot execute any further data transfer statements until it executes a BACKSPACE or REWIND statement.

**The Argument-list**

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer in the range 0 to 99 and specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or integer array element. The I/O system assigns IO-status the following values based on the outcome of the positioning:

IO-status = 0 if no error has occurred

IO-status ≥ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the ENDFILE statement. If an error occurs while processing the ENDFILE statement, the following actions occur:

1. The ENDFILE operation is terminated.
2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, REWIND, BACKSPACE, or INQUIRE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

**Restrictions**

- The ENDFILE statement cannot reference an internal file.
- In the form, ENDFILE(argument-list), you must include an I/O unit specifier.
- The file must be connected for sequential access.

**Examples**

```
ENDFILE 4
ENDFILE(4,IOSTAT=errorflag,ERR=999)
```

## 15.7. INQUIRE Statement

The INQUIRE statement returns information about the characteristics of a named file, or the connection of a file to a particular I/O unit.

**Syntax:**

```
INQUIRE (FILE=file-name, inquire-list)
INQUIRE ([UNIT=]unit-number, inquire-list)
```

The first form is called inquire-by-file, and requires a FILE= file-name keyword qualifier. This qualifier MUST be specified, but may appear at any point in the inquire_list. The file file-name need not exist, or be connected.

The second form is called inquire-by-unit, and requires that the first parameter be an integer unit number, or that the UNIT= unit-number specifier appear somewhere in the inquire-list.

unit-number must be an integer value between 0 and 99, specifying the external I/O unit and may be preceded by the UNIT= keyword.

### Inquire-list Specifiers

Inquire list specifiers may be either upper-case or lower-case. Multiple specifiers in an INQUIRE statement must be separated by commas.

The following table summarizes the INQUIRE keywords accepted by Fortran. A comment of 'Ignored' indicates that the keyword or value is accepted by the compiler, but is ignored. Information on keywords accepted but ignored by the compiler is included for VAX/VMS compatibility considerations.

Detailed descriptions for each supported keyword follow the table, and are presented alphabetically.

| Keyword | Value | Comment |
|---|---|---|
| ACCESS | 'SEQUENTIAL' | |
| | 'DIRECT' | |
| | 'UNKNOWN' | Returned if no connection exists |
| BLANK | 'NULL' | Null blank control used |
| | 'ZERO' | Zero blank control used |
| | 'UNKNOWN' | No connection or not formatted I/O |
| CARRIAGECONTROL | 'FORTRAN' | File has Fortran carriage control |
| | 'LIST' | File has implied carriage control |
| | 'NONE' | File has no carriage control |
| | 'UNKNOWN' | No other value applies |
| DEFAULTFILE | n/a | Ignored |
| DIRECT | 'YES' | Direct access allowed |
| | 'NO' | Direct access not allowed |
| | 'UNKNOWN' | Cannot determine access |
| ERR | error-label | Control transfers to error-label |
| EXIST | .TRUE. | File or unit exists |
| | .FALSE. | File or unit doesn't exist |
| FILE | filename | Required unless UNIT specified |
| FORM | 'FORMATTED' | Opened for formatted I/O |
| | 'UNFORMATTED' | Opened for unformatted I/O |
| | 'UNKNOWN' | No connection |
| FORMATTED | 'YES' | Formatted form allowed |
| | 'NO' | Formatted form not allowed |
| | 'UNKNOWN' | Formatted form undetermined |
| IOSTAT | INTEGER | INQUIRE execution return code |
| NAME | filename | Full filename |
| NAMED | .TRUE. | File has filename |
| | .FALSE. | File does not have filename |
| NEXTREC | INTEGER | Next record number to be accessed |
| NUMBER | INTEGER | Logical unit number |
| OPENED | .TRUE. | File or unit is open |
| | .FALSE | File or unit not open |
| ORGANIZATION | 'SEQUENTIAL' | Sequential file |
| | 'RELATIVE' | Relative file |
| | 'UNKNOWN' | File organization undetermined |
| RECL | INTEGER | Maximum record length |
| RECORDTYPE | n/a | Ignored |
| SEQUENTIAL | 'YES' | Sequential access allowed |
| | 'NO' | No sequential access allowed |
| | 'UNKNOWN' | Access undetermined |
| UNFORMATTED | 'YES' | Unformatted form allowed |
| | 'NO' | Unformatted form not allowed |
| | 'UNKNOWN' | Form undetermined |
| UNIT | unit-number | Alternate form to FILE |

Access Method Specifier

ACCESS = Access-method

Access-method is a character variable or an element of a character array that the I/O system assigns the character value 'SEQUENTIAL' if the file being inquired about is connected for sequential access, or the character value 'DIRECT' if the file is connected for direct access. The character value 'UNKNOWN' is returned if the file is not connected

Blank Specifier

BLANK = Blank-type

Blank-type is a character variable or an element of a character array that the I/O system assigns the character value 'NULL' if the file is connected for formatted I/O and is using null blank control, or the character value 'ZERO' if the file is connected for formatted I/O and is using zero blank control. Refer to the "OPEN Statement" section for more information.

If the file is not connected, or it is not connected for formatted I/O, Blank-type is assigned the character value 'UNKNOWN'.

Carriage Control Specifier

CARRIAGECONTROL = Control-type

Control-type is a character variable that the I/O system assigns based on the file's carriage control attributes. The value 'FORTRAN' is assigned if the file uses Fortran carriage control, 'LIST' is returned if the file has implied carriage control, and 'NONE' is returned if the file has no implicit carriage control. The value 'UNKNOWN' is returned if the carriage control attributes of a file cannot be determined by the I/O system.

Direct Specifier

DIRECT = Direct-allowed

Direct-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if direct access is allowed for the file being inquired about, or the character value 'NO' if direct access is not allowed for the file.

If the I/O system cannot determine whether direct access is allowed for the file, it assigns the value 'UNKNOWN'.

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the INQUIRE statement. If the I/O system encounters an error while processing the INQUIRE statement, the following actions occur:

1. The INQUIRE operation is terminated.
2. The file position becomes undefined, unless the error is an end-of-file condition. Then, the file pointer points just past the endfile record, and the only valid statements that you can execute are CLOSE, BACKSPACE, or REWIND.
3. The I/O system sets the IO-status (if one is specified); all other specifiers become undefined.
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

Existence Specifier

EXIST = Exist-status

Exist-status is a logical variable or an element of a logical array.

When executing an inquire-by-file statement, the I/O system assigns Exist-status the logical value .TRUE. if File-name exists; otherwise, it assigns the logical value .FALSE..

When executing an inquire-by-unit statement, the I/O system assigns Exist-status the logical value .TRUE. if Unit-number exists; otherwise, it assigns the logical value .FALSE..

File Name Specifier

FILE = File-name

File-name is the name of the file being inquired about. This specifier is required whenever the first form of the INQUIRE statement is used. The file File-name need not exist, or be connected.

I/O Type Specifier

FORM = IO-type

IO-type is a character variable or an element of a character array that the I/O system assigns the character value 'FORMATTED' if the file being inquired about is connected for formatted I/O, or 'UNFORMATTED' if the file is connected for unformatted I/O.

If the file being inquired about is not connected, IO-type is assigned the value 'UNKNOWN'.

Formatted Specifier

FORMATTED = Formatted-allowed

Formatted-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if the file being inquired about can contain formatted records, or the character value 'NO' if the file cannot contain formatted records.

If the I/O system cannot determine whether the file can contain formatted records, it assigns the character value 'UNKNOWN'.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer or an element of an integer array. The I/O system assigns one of the following values based on the outcome of the inquiry.

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred
IO-status = –1 if an end-of-file condition has occurred

Current Name Specifier

NAME = Current-file-name

Current-file-name is a character variable or an element of a character array that the I/O system assigns the current name of the file being inquired about.

If the file has no name, Current-file-name remains unchanged.

Named Specifier

NAMED = Named-status

Named-status is a logical variable or an element of a logical array. When the I/O system executes an inquire-by-unit statement, it assigns Named-status the logical value .TRUE. if the file connected to the Unit-number has a name; otherwise, it assigns the value .FALSE..

Next Record Specifier

NEXTREC = Next-record-number

Next-record-number is an integer variable or an element of an integer array that the I/O system assigns the value n+1, where n is the number of the last record the I/O system has read from or written to in the file being inquired about.

If the file is connected, but the I/O system has not yet read or written any records, Next-record-number is assigned the value 1.

If the file is not connected for direct access, or the file position is undefined because of a previous error, Next-record-number is set to zero (0).

Current Unit Specifier

NUMBER = Current-unit-number

Current-unit-number is an integer or an element of an integer array to which the I/O system assigns the number of the I/O unit currently connected to File-name. If no I/O unit is connected to File-name, Current-unit-number remains unchanged.

Open Specifier

OPENED = Open-status

Open-status is a logical variable or an element of a logical array.

When executing an inquire-by-file statement, the I/O system assigns Open-status the logical value .TRUE. if File-name is connected to a I/O unit; otherwise, it assigns the value .FALSE..

When executing an inquire-by-unit statement, the I/O system assigns Open-status the logical value .TRUE. if Unit-number is connected to a file; otherwise, it assigns the value .FALSE..

File Organization Specifier

ORGANIZATION = Organization-type

Organization-type is a character variable or array element that the I/O system assigns a character value of 'SEQUENTIAL' or 'RELATIVE' depending on file organization.

If the file organization cannot be determined, the value 'UNKNOWN' is assigned.

Current Record-Length Specifier

RECL = Current-record-length

Current-record-length is an integer variable or an element of an integer array that the I/O system assigns the current value for the record length in the file being inquired about.

If the file is connected for formatted I/O, Current-record-length is the record length in bytes.

If the file is connected for unformatted I/O, Current-record-length is measured in 32-bit words, unless F77 compatibility mode is chosen, in which case Current-record-length is measured in bytes.

If the file is not connected, or the file is not connected for direct access, Current-record-length is set to zero (0).

Sequential Specifier

SEQUENTIAL = Sequential-allowed

Sequential-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if sequential access is allowed for the file, or the character value 'NO' if the sequential access is not allowed for the file.

If the I/O system cannot determine whether sequential access is allowed for the file, it assigns the character value 'UNKNOWN'.

Unformatted Specifier

UNFORMATTED = Unformatted-allowed

Unformatted-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if the file being inquired about can contain unformatted records, or the character value 'NO' if the file cannot contain unformatted records.

If the I/O system cannot determine whether the file can contain unformatted records, it assigns the character value 'UNKNOWN'.

Unit Number Specifier

[UNIT =] Unit-number

Unit-number is an integer value from 0 through 99 that specifies the external I/O unit. Unit-number is required whenever the second form of the INQUIRE statement is used. The UNIT= keyword is optional if the Unit-number is the first element of the inquire-list.

**Restrictions**

- The I/O system can execute the INQUIRE statement before, while, or after a file is connected to a I/O unit.
- All the values assigned to the inquire-list specifiers are those that are current when the I/O system executes the INQUIRE statement.
- Any variable or array element that becomes defined or undefined by being used as a specifier in an INQUIRE statement cannot be referenced by another specifier in the same INQUIRE statement.
- When the I/O system executes an inquire-by-file statement, the specifiers:
  - Named-status
  - Current-file-name
  - Sequential-allowed
  - Direct-allowed
  - Formatted-allowed
  - Unformatted-allowed

  are assigned values only if File-name is a valid filename for the implementation and exists; otherwise, they all remain unchanged.

  Current-unit-number is assigned a value if and only if Open-status is TRUE.

  Also, if Open-status is assigned TRUE, then the specifiers:
  - Access-method
  - IO-type
  - Current-record-length
  - Next-record-number
  - Blank-type can also become defined.
- When the I/O system executes an inquire-by-unit statement, the specifiers
  - Current-unit-number
  - Named-status
  - Current-file-name
  - Access-method
  - Sequential-allowed
  - Direct-allowed
  - IO-type
  - Formatted-allowed
  - Unformatted-allowed
  - Current-record-length

- Next-record-number
- Blank-type

are assigned values only if Unit-number exists and is connected to a file; otherwise they all remain unchanged.

- When the I/O system executes the INQUIRE statement, the specifiers Exist-status and Open-status are always assigned a value unless an error condition occurs.

**Example:**

```
INQUIRE (3, NAME=FNAME, OPENED=ESTAT)
INQUIRE (FILE='TMP.DAT', NUMBER=FILEUNIT)
```

The first INQUIRE example, above, returns the name of the file currently associated with unit 3 (if any) and sets the variable ESTAT to a logical TRUE or FALSE depending on file open status.

The second INQUIRE example returns the unit number, if any, associated with the file TMP.DAT into the variable FILEUNIT.

**F77 Compatibility Notes:**

The RECL keyword specifier Current-record-length is measured in bytes for unformatted I/O in F77 compatibility mode, and in 32-bit words otherwise.

The following INQUIRE keywords are part of the VMS extensions, and therefore are not available when using F77 compatibility mode.

```
ACCESS='UNKNOWN' CARRIAGECONTROL DEFAULTFILE
FORM='UNKNOWN'   ORGANIZATION    RECORDTYPE
```

Please refer to your User's Guide for details on compiler options and default modes.

## 15.8. OPEN Statement

The OPEN statement can

- create a file and connect it to a I/O unit.
- recreate a preconnected file.
- connect an existing file to a I/O unit.
- change the characteristics of an existing I/O unit/file connection.

**Syntax:**

```
OPEN(open-list)
```

where open-list is a comma separated list of specifiers that control the opening process.

**The Open-list**

The following table summarizes the OPEN keywords accepted by Fortran. A comment of 'Ignored' indicates that the keyword or value is accepted by the compiler, but is ignored. Information on keywords accepted but ignored by the compiler is included for VAX/VMS compatibility considerations.

Detailed descriptions for each supported keyword follow the table, and are presented alphabetically.

| Keyword | Value | Default/Comment |
|---|---|---|
| ACCESS | 'SEQUENTIAL' | 'SEQUENTIAL' |
| | 'DIRECT' | |
| | 'APPEND' | |
| ASSOCIATEVARIABLE | INTEGER*4 | Next record |
| BLANK | 'NULL' | 'NULL' |
| | 'ZERO' | |
| BLOCKSIZE | n/a | Ignored |
| BUFFERCOUNT | n/a | Ignored |
| CARRIAGECONTROL | 'FORTRAN' | 'FORTRAN' for formatted files |
| | 'LIST' | |
| | 'NONE' | 'NONE' for unformatted files |
| DEFAULTFILE | n/a | Ignored |
| DISP | | Equivalent to DISPOSE |
| DISPOSE | 'KEEP' | 'KEEP' |
| | 'SAVE' | |
| | 'DELETE' | |
| | 'PRINT' | Ignored |
| | 'PRINT/DELETE' | Equivalent to 'DELETE' |
| | 'SUBMIT' | Ignored |
| | 'SUBMIT/DELETE' | Equivalent to 'DELETE' |
| ERR | error-label | Value required (no default) |
| EXTENDSIZE | n/a | Ignored |
| FORM | 'FORMATTED' | 'FORMATTED' for direct access |
| | 'UNFORMATTED' | 'UNFORMATTED' for sequential access |
| FILE | scalar/array | fort.n or FORnnn.DAT (n=I/O unit) |
| INITIALSIZE | n/a | Ignored |
| IOSTAT | INTEGER | Value required (no default) |
| MAXREC | n/a | Ignored |
| NOSPANBLOCKS | n/a | Ignored |
| ORGANIZATION | 'SEQUENTIAL' | 'SEQUENTIAL' |
| | 'RELATIVE' | |
| READONLY | n/a | Ignored |
| RECL | INTEGER | RECORDSIZE is equivalent keyword |
| RECORDTYPE | 'FIXED' | Ignored |
| | 'VARIABLE' | Ignored |
| | 'SEGMENTED' | Ignored |
| | 'STREAM' | Ignored |
| | 'STREAM_CR' | Ignored |
| | 'STREAM_LF' | Ignored |
| SHARED | n/a | Ignored |
| STATUS | 'UNKNOWN' | 'UNKNOWN' |
| | 'NEW' | |
| | 'OLD' | |
| | 'SCRATCH' | |
| TYPE | | Equivalent to STATUS |
| UNIT | INTEGER | Values 0-99 only |
| USEROPEN | routine-name | Requires user-supplied OPEN runtime |

Access Method Specifier

ACCESS = Access-method

Access-method is a character string expression whose value must be either:

'SEQUENTIAL'
'DIRECT' or
'APPEND'

If you omit ACCESS, the I/O system supplies the default ACCESS = 'SEQUENTIAL'.

If you specify ACCESS = 'DIRECT', you must also specify the record length (see below).

If you specify ACCESS = 'APPEND', the file is accessed sequentially, after the last record in the file.

If the file already exists, Access-method must match the file's characteristics. If the file does not already exist, the OPEN statement creates it with the given Access-method.

Direct Access Record Specifier

ASSOCIATEVARIABLE = Record-number

Record-number is an INTEGER*4 variable whose value reflects the record number of the next sequential record in a direct access file. This keyword is valid for direct access files only and is otherwise ignored.

Blank Specifier

BLANK = Blank-type

Blank-type is a character string expression whose value is either:

'NULL' or 'ZERO'

If you omit BLANK, the I/O system supplies the default BLANK = 'NULL'.

If you specify BLANK = 'NULL', the I/O system ignores any blank (20h) characters in numeric, formatted input fields, with the exception that a field of all blanks has the value zero (30h).

If you specify BLANK = 'ZERO', the I/O system treats all blanks, except leading blanks as zeros.

Carriage Control Specifier

CARRIAGECONTROL = Control-type

Control-type is a character string expression whose value is either

'FORTRAN'
'LIST' or
'NONE'

If you omit CARRIAGECONTROL, the I/O system supplies the default CARRIAGECONTROL = 'FORTRAN' if formatted files are used. Otherwise, the default CARRIAGECONTROL = 'NONE' is supplied.

If you specify CARRIAGECONTROL = 'FORTRAN', normal Fortran interpretation of the initial character is used when printing a file.

If CARRIAGECONTROL = 'LIST' is used, the file is printed with single spaces between records.

If CARRIAGECONTROL = 'NONE' is specified, no implied carriage control is used when the file is printed.

File Disposition Specifier

DISPOSE = Disposition-type or DISP = Disposition-type

The DISPOSE (or DISP) keyword is used to specify the disposition of a file when the associated logical unit is closed.

Disposition-type may be one of the following values:

'KEEP' or 'SAVE'

'DELETE' or 'PRINT/DELETE' or 'SUBMIT/DELETE'

If you omit DISPOSE, the default is 'KEEP', which will retain the file

after closure. Disposition-type 'KEEP' and 'SAVE' are equivalent.

If you specify DISPOSE = 'DELETE', the file is deleted after the unit is closed. Note that 'DELETE', 'PRINT/DELETE' and 'SUBMIT/DELETE' are equivalent.

Disposition exceptions occur if an attempt is made to save a scratch file, or to delete a read-only file. In these circumstances the conflicting Disposition-type is ignored.

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the OPEN statement. If the I/O system encounters an error while processing the OPEN statement, the following actions occur:

1. The OPEN operation is terminated.
2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

File Specifier

FILE = File-name

File-name is the name of the file to be connected to the specified I/O unit. File-name must be the name of a file in the operating system's file structure.

If you omit the FILE specifier and the I/O unit is not preconnected, the I/O system connects the I/O unit to a file (named fort. on on Unix systems, and FORnnn.DAT on VMS systems), where n is the I/O unit number.

Record Type Specifier

FORM = Record-type

Record-type is a character string expression whose value must be either:

'FORMATTED' or 'UNFORMATTED'

If you omit FORM, the I/O system supplies the default FORM = 'FORMATTED' when you connect the file for direct access, or FORM = 'UNFORMATTED' when you connect the file for sequential access.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer or an element of an integer array. The I/O system assigns IO-status one of the following values based on the outcome of the open process:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred

File Organization Specifier

ORGANIZATION = File-type

File-type is a character string expression whose value must be either:

'SEQUENTIAL' or 'RELATIVE'

If ORGANIZATION is omitted, the I/O system uses the organization of the existing file. If the file does not already exist, the default ORGANIZATION = 'SEQUENTIAL' is supplied.

If ORGANIZATION is specified for an existing file, it must have the same value as that file.

Record Length Specifier

RECL = Record-length

Record-length is a positive integer expression whose value is the length of each record in the file. The file must be connected for direct access.

If the records are formatted, Record-length is the number of bytes. If the records are unformatted, Record-length is the number of bytes in F77 compatibility mode, and 32-bit words otherwise.

Status Specifier

STATUS = Status or TYPE = Status

Status is a character string expression whose value must be one of the following:

'NEW' or 'OLD' or 'SCRATCH' or 'UNKNOWN'

The keywords STATUS and TYPE are equivalent.

If you omit STATUS, the I/O system supplies the default STATUS = 'UNKNOWN'. NEW and OLD are only valid if you also use a FILE specifier, or the file is preconnected.

If you specify STATUS = 'SCRATCH', the I/O system connects the specified I/O unit to a temporary file. The connection lasts until you execute a CLOSE statement (see below), or the program terminates.

If you specify STATUS = 'UNKNOWN' and the file exists, the I/O system connects the file. If you specify STATUS = 'UNKNOWN' and the file does not exist, the I/O system first creates the file and then connects it.

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer value between 0 and 99 that specifies an external I/O unit. The keyword UNIT = is optional.

User-Written Function Specifier

USEROPEN = Routine-name

Routine-name is the symbolic name of a user-supplied replacement for the runtime OPEN routine.

The Fortran compiler accepts the USEROPEN keyword and the runtime library supports it, however its use implies the existence of a user-supplied replacement for the runtime OPEN routine. The Fortran routine OPEN routine has its own calling conventions and is not intended to be a replacement for any other such runtime routine. User replacement of this routine is a significant undertaking, at minimum requiring licensed access to the library source.

**Restrictions**

- The open-list must contain a Unit-number; all other specifiers are optional, but there can be only one of each.
- If you omit UNIT = , then the Unit-number must be the first specifier in the open-list.
- If the file is connected for direct access, the open-list must contain a Record-length.
- If the file is connected for sequential access, the open-list cannot contain a Record-length.
- If STATUS = 'SCRATCH', the open-list cannot contain a File-name.
- If the file is already connected to a I/O unit, the only specifier that can be changed with the OPEN statement is Blank-type. The file position is not affected.
- The specifier BLANK = is valid only with formatted records.

**Examples**

```
OPEN(3)
OPEN(UNIT=3,STATUS='SCRATCH')
OPEN(8,FILE='overdues.dat',IOSTAT=errorflag,ERR=999)
```

**F77 Compatibility Notes:**

The RECL keyword specifier, Record-length, is measured in bytes under F77 compatibility mode, and in 32-bit words otherwise.

The following OPEN keywords are part of the VMS extensions, and therefore are not available when using F77 compatibility mode.

| | | | |
|---|---|---|---|
| ACCESS='APPEND' | ASSOCIATEVARIABLE | BLOCKSIZE | BUFFERCOUNT |
| CARRIAGECONTROL | DEFAULTFILE | DISPOSE | DISP |
| EXTENDSIZE | NAME | INITIALSIZE | MAXREC |
| NOSPANBLOCKS | ORGANIZATION | READONLY | RECORDSIZE |
| RECORDTYPE | SHARED | TYPE | USEROPEN |

Please refer to your User's Guide for details on compiler options and default modes.

## 15.9. PRINT Statement

The PRINT statement transfers formatted data to the default output I/O unit. The PRINT statement is functionally equivalent to using a WRITE statement with

formatted records.

**Syntax:**

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

where format is a format specification and output-list is the list of data items to be written.

When executing the PRINT statement, the I/O system does not print the first character in a formatted record. Instead, the I/O system uses this character to determine the vertical spacing to use before printing the remainder of the record. The PRINT statement then prints any remaining characters in the record on one line beginning at the lefthand margin.

The table below shows the amount vertical spacing determined by the first character.

**Vertical Spacing in the PRINT Statement**

| Character | Hex Value | Vertical Space Output Before Printing Formatted Record |
|---|---|---|
| Blank | 20h | Advance one line |
| 0 | 30h | Advance two lines |
| 1 | 31h | Advance to first line of next page |
| + | 2Bh | No advance |

### The Output-list

The output-list can contain any of the following:

- a variable name
- an array name
- an array element name
- an expression containing any of the types CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL

If you use an array name in the output-list, the I/O system writes the entire array in the normal column-major order.

### Implied-DO List

The PRINT statement can write a range of subscripted array elements from the output-list using an implied-DO list of the form:

```
(input-list,var = e1,e2[,e3])
```

where var and e1,e2, and e3 are the same as described in the explanation of the DO statement in the "Control Statements" chapter. The output-list can contain other (nested) implied-DO lists.

### Restrictions

- The data must be formatted.

### Examples

```
PRINT 35, name,score
PRINT *, employee
```

## 15.10. READ Statement

The READ statement transfers data from a specific I/O unit to internal (processor) storage. You can use the READ statement to reference both internal and external files. The READ statement can transfer both formatted and unformatted records.

### Syntax:

```
READ(control-list)[input-list]
READ Format[,input-list]
```

In the first form, the control-list is a list of comma separated specifiers that control the transfer process. In the second form, format is a format specification. Refer to the "Format Statement and Format Specifications" chapter for more information on format specification. In both forms, input-list is the list of variables that receive the input data.

**The Control-list**

Unit Specifier

[UNIT =] Unit-id

Unit-id can be an unsigned integer in the range 0 to 99 that identifies an external I/O unit, an asterisk (*) specifying the default input I/O unit, or an internal file. The keyword UNIT = is optional.

Format Specifier

[FMT =] Format

Format is a format identifier that controls the editing of the data during the transfer (see "The Format Statement and Format Specification" chapter). The keyword FMT = is optional.

Record Specifier

REC = Record-number

Record-number is a nonzero integer expression that specifies the number of the record to read.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or an element of an integer array. The I/O system assigns IO-status one of the following values based on the outcome of the data transfer:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred
IO-status = –1 if an end-of-file condition has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the READ statement. If the I/O system encounters an error while processing the READ statement, the following actions occur:

1. The READ operation is terminated.
2. The file position becomes undefined, unless the error is an end-of-file condition. Then, the file pointer points just past the endfile record, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

End Specifier

END = End-label

End-label is the label of an executable statement in the same program unit as the I/O statement. When the I/O system encounters an end-of-file condition while processing the READ statement, the following actions occur:

1. The READ operation is terminated.
2. The file pointer points just past the endfile record, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND or INQUIRE.
3. The I/O system sets the IO-status = –1 (if one is specified).
4. The flow of control resumes at the statement labeled with End-label. If there is no END specifier, a run-time error occurs.

**The Input-list**

The input-list can contain any of the following:

- a variable name
- an array name
- an array element name

If you use an array name in the input-list, the I/O system reads the entire array in the normal column-major order.

**Implied-DO List**

The READ statement can read a range of subscripted array elements from the input-list using an implied-DO list of the form:

```
(input-list,var = e1,e2[,e3])
```

where var and e1, e2, and e3 are the same as described in the explanation of the DO statement in "Control Statements" chapter. The input-list can contain other (nested) implied-DO lists.

**Restrictions**

- The control-list must contain an I/O unit specifier, but can contain only one of each of the other specifiers.
- If you do not use UNIT = , then the Unit-id must be the first specifier in the control-list.
- If you do not use FMT = , then you cannot use UNIT = and the Format must be the second specifier in the control-list.
- In the form READ Format[,input-list], the I/O unit is the default input unit.
- If Unit-id designates an internal file, the control-list must contain a format specification other than an asterisk (*), but cannot contain a record number.
- If the Format is an asterisk (*), specifying list-directed formatting, the control-list cannot contain a record number specifier.
- If the file is connected for direct access, the control-list must contain a record number.
- The END specifier is only valid if the file is connected for sequential access.

**Examples**

```
READ(10,200) name,score
READ(10,200,IOSTAT=errorflag,ERR=350,END=999) name,score
READ(3,REC=105,IOSTAT=errorflag,ERR=200,END=999) altitude
READ(3,REC=(2j + k),IOSTAT=errorflag,ERR=200,END=999) altitude
READ(5,*) employee
```

**Relative Record Specifications**

The 'n form of relative record specification is supported in VMS compatibility mode only.

**Example:**

```
READ (10'7)
```

will read record 7 from a direct access file on logical unit 10.

## 15.11. REWIND Statement

The REWIND statement moves the file pointer to the initial point of the file.

**Syntax:**

```
REWIND Unit-number
REWIND(argument-list)
```

In the first form, Unit-number is an integer expression whose value is in the range 0 to 99. In the second form, argument-list is a comma separated list of specifiers that control the positioning process.

With both forms, if the file pointer is already positioned at the initial point, the I/O system ignores the REWIND statement.

**The Argument-list**

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer in the range 0 to 99 that specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or integer array element. The I/O system assigns IO-status the following values based on the outcome of the positioning:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the REWIND statement. If an error occurs while processing the REWIND statement, the following actions occur:

1. The REWIND operation is terminated.
2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

**Restrictions**

- The REWIND statement cannot reference an internal file.
- In the form REWIND(argument-list), you must include an I/O unit specifier.
- The file must be connected for sequential access.

**Examples**

```
REWIND 4
REWIND(4,IOSTAT=errorflag,ERR=999)
```

## 15.12. TYPE Statement

The TYPE statement is the same as the formatted sequential WRITE statement, except that it cannot write to any unit other than standard output. That is, TYPE bears the same relationship to WRITE as ACCEPT does to READ.

**Syntax:** `TYPE f[,iolist]`

or

`TYPE *[,iolist]`

or

`TYPE n`

where *f* is the nonkeyword form of a format specifier; * implies list-directed output; *n* is the nonkeyword form of a namelist specifier; and iolist is a list of elements to be output.

```
      CHARACTER*12     NAME(5)
      INTEGER          BUF1, BUF2
      NAMELIST /MYLIST/ A,B,C,D
      TYPE *, BUF1, BUF2
      TYPE 1000, I, J, K
      TYPE MYLIST
1000  FORMAT (3I5)
```

Here, the first TYPE statement writes character data from array NAMES; the second one writes character data from array MORENAMES; the third writes the values of E, F, G, and H. All output is to the implicit unit (standard output).

**F77 Compatibility Notes:**

The TYPE statement is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 15.13. WRITE Statement

The WRITE statement transfers data from internal (processor) storage to a specific I/O unit. You can use the WRITE statement to reference both internal and external files. The WRITE statement can transfer both formatted and unformatted records.

**Syntax:**

```
WRITE(control-list)[output-list]
```

where control-list is a comma separated list of specifiers that control the transfer process, and output-list is the list of variables to be written.

**The Control-list**

Unit Specifier

[UNIT =] Unit-id

Unit-id can be an unsigned integer in the range 0 to 99 that designates an external I/O unit, an asterisk (*) specifying the default output I/O unit, or an internal file. The keyword UNIT = is optional.

Format Specifier

[FMT =] Format

Format is a format identifier that controls the editing of the data during the transfer. Refer to the "Format Statement and Format Specifications" chapter for more information. The keyword FMT = is optional.

Record Specifier

REC = Record-number

Record-number is a nonzero integer expression that specifies the number of the record to write.

I/O-status Specifier

IOSTAT = IO-status

IO-status is an integer variable or an element of an integer array.

The I/O system assigns IO-status one of the following values based on the outcome of the data transfer:

IO-status = 0 if no error has occurred
IO-status ≥ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the WRITE statement. If the I/O system encounters an error while processing the WRITE statement, the following actions occur:

1. The WRITE operation is terminated.
2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE or INQUIRE.
3. The I/O system sets the IO-status (if one is specified).
4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

**The Output-list**

The output-list can contain any of the following:

- a variable name
- an array name
- an array element name
- an expression containing any of the types CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL

If you use an array name in the output-list, the I/O system writes the entire array in the normal column-major order.

**Implied-DO List**

The WRITE statement can write a range of subscripted array elements from the output-list using an implied-DO list of the form:

```
(output-list,var = e1,e2[,e3])
```

where var and e1,e2, and e3 are the same as described in the explanation of the DO statement in the "Control Statements" chapter. The output-list can contain other (nested) implied-DO lists.

### Restrictions

- The control-list must contain an I/O unit specifier, but can contain only one of each of the other specifiers.
- If you do not use UNIT = , then the Unit-id must be the first specifier in the control-list.
- If you do not use FMT = , then you cannot use UNIT = and the Format must be the second specifier in the control-list.
- If Unit-id designates an internal file, the control-list must contain a format specification other than an asterisk (*), but cannot contain a record number.
- If the Format is an asterisk (*) specifying list-directed formatting, the control-list cannot contain a record number specifier.
- If the file is connected for direct access, the control-list must contain a record number.

### Examples

```
WRITE(10,200) name,score
WRITE(10,200,IOSTAT=errorflag,ERR=350) name,score
WRITE(3,REC=105,IOSTAT=errorflag,ERR=200) altitude
WRITE(3,REC=(2j + k),IOSTAT=errorflag,ERR=200) altitude
WRITE(6,*) employee
```

# Chapter 16
# The FORMAT Statement and Format Specification

This section describes the various methods of format specification, including the FORMAT statement and list-directed I/O. It also describes how to use edit descriptors when performing formatted data transfer.

## 16.1. Specifying Formats

There are three ways to specify a format:

- explicitly, in a FORMAT statement
- implicitly, as a character variable, element of a character array, or any character expression that evaluates to a valid format specification
- implicitly, as list-directed formatting (see the "List-directed Formatting" section below)

### 16.1.1 The FORMAT Statement

The FORMAT statement is a labeled statement that defines a formatspecification. It must appear in the same program unit where it is referenced.

Syntax:

```
statement-label FORMAT format-specification
```

where format-specification is a valid form of format specification as described below.

### 16.1.2. Character Format Specification

A character format specification must have the form of a valid format specification starting at the leftmost character position. It must be enclosed in parentheses, and any characters following the right parenthesis do not affect the format specification.

If a format specification is given as a character array name and the specification's length exceeds the length of the first array element, the specification becomes the concatenation of all the array elements in normal column-major order.

If the format specification is an array element, the specification's length cannot exceed the length of the array element.

## 16.2. General Form For Format Specification

A format-specification has the general form

```
([format-list])
```

The format-list is a list of items enclosed in parentheses. The items in the format-list can be any of the following:

```
[r] repeatable-edit-descriptor
nonrepeatable-edit-descriptor
[r] format-list
```

where repeatable-edit-descriptor and nonrepeatable-edit-descriptor are special character strings that describe the kind of editing being performed (see the descriptions below). "r" is a positive integer constant called the repeat-factor. If omitted, the I/O system supplies the default value r = 1.

The format-list can be empty only if the corresponding I/O list is also empty. If format-list contains another (nested) format-list, the nested list cannot be empty.

If the I/O list contains at least one item, the format-list must contain at least one repeatable edit descriptor.

## 16.3. Format Control

The interaction between the I/O list and the format specification is a dynamic process called format control.

Format control always proceeds from left to right, matching each item in the I/O list with the next repeatable edit descriptor. There is no match between I/O list items and nonrepeatable edit descriptors. Format control executes nonrepeatable edit descriptors as they are encountered.

If an edit descriptor has a repeat-factor "r", format control processes the I/O list as if it contained "r" consecutive items.

If the format-list ends before reaching the end of the I/O list, format control reverts to the beginning of the last nested format-list, if there is one. If there is none, format control reverts to the beginning of the format-specification and again passes through the I/O list. Each time format control reverts, it accesses a new record.

## 16.4. Using Repeatable Edit Descriptors

Repeatable edit descriptors control the editing of character, logical, and numeric data. Each item in the I/O list corresponds to a repeatable edit descriptor in the format-list.

Each repeatable edit descriptor can be preceded by a repeat-factor, which determines how many times to repeat the edit specified by the edit descriptor.

Each repeatable edit descriptor consists of a single letter together with a number. The letter indicates the type of data to edit, and the number indicates the size of the data field.

**Repeatable Edit Descriptors**

| Syntax | Type of Descriptor |
|---|---|
| A[w] | Alphanumeric Descriptor |
| Dw.d | Floating-point Descriptor |
| Ew.d[Ee] | Floating-point Descriptor |
| Fw.d | Floating-point Descriptor |
| Gw.d[Ee] | Floating-point Descriptor |
| Iw | Integer Descriptor |
| Iw.m | Integer Descriptor |
| Lw | Logical Descriptor |
| Ow[.m] | Octal Descriptor |
| Zw[.m] | Hexadecimal Descriptor |

In the table above, A, D, E, F, G, I, L, O and Z indicate the type of data to edit; "w" is a positive integer constant called the field width, and indicates the number of characters in the field; "d" is an unsigned integer constant indicating the number of digits following the decimal point; "e" is a positive integer constant indicating the number of digits in the exponent.

The following subsections describe how to use each repeatable edit descriptor to edit alphanumeric, numeric, and logical data.

## 16.5. Alphanumeric Editing

The A edit descriptor is used to edit character or Hollerith data. Storage is allocated based on the maximum number of characters that may be stored in that data type.

**Syntax:** A[*w*]

where w is an optional integer value greater than 0 specifying field width.

If *w* is present, the field width is *w* characters. If *w* is not present, the field width defaults to the length of the data item in the I/O list, or in the case of non-character data types, to the maximum storage length for that type. Note that COMPLEX and DOUBLE COMPLEX values are stored as real number pairs and will, therefore, require two format descriptors.

**Maximum Storage (in characters) for Various Data Types**

| Data Type | Storage | Data Type | Storage |
|---|---|---|---|
| BYTE | 1 | | |
| CHARACTER*$n$ | $n$ | | |
| INTEGER*2 | 2 | INTEGER*4 | 4 |
| LOGICAL*1 | 1 | LOGICAL*2 | 2 |
| LOGICAL*4 | 4 | | |
| REAL | 4 | DOUBLE PRECISION | 8 |
| COMPLEX | 8 | DOUBLE COMPLEX | 16 |

The following rules apply to A[*w*], where *len* is the actual length of the I/O list element:

On input,

- If $w \geq len$ , the I/O system transfers the rightmost *len* characters. If $w < len$ , the I/O system transfers w characters and they are left-justified, with ( $len - w$ ) trailing blanks.

On output,

- If $w > len$ , the I/O system transfers ( $w - len$ ) blanks, followed by *len* characters. If $w \leq len$ , the I/O system transfers the leftmost w characters.

**Example:**

**Input Processing**

| Data Type | Format | I/O List Item | Input Result |
|---|---|---|---|
| CHARACTER*8 | A3 | Math | Mat␣␣␣␣␣ |
| CHARACTER*8 | A7 | Math | Math␣␣␣␣␣ |
| INTEGER*2 | A5 | Math | Ma |
| REAL | A3 | Math | Mat␣ |

**Output Processing**

| Data Type | Format | I/O List Item | Output Result |
|---|---|---|---|
| CHARACTER*8 | A3 | Math | Mat |
| CHARACTER*8 | A7 | Math | ␣␣␣Math |

## 16.6. Numeric Editing

The D, E, F, G, I, O and Z edit descriptors edit numeric data. D, E, F, and G edit any floating-point type such as REAL*4, REAL*8, COMPLEX*8, or COMPLEX*16. I edits INTEGER data. E and G produce floating-point numbers in scientific notation (on output only). O and Z are used for octal and hexadecimal numbers.

The following rules apply to all the numeric edit descriptors:

On input,

- the I/O system ignores leading blanks, except that a field of all blanks is treated as a zero. Treatment of other blanks is determined by the BLANK specifier in the OPEN statement, and the settings of the nonrepeatable edit descriptors BN and BZ.
- a decimal point in the input field overrides the placement of the decimal point specified by a D, E, F, or G edit descriptor. Also, the input field can contain more digits than the processor needs to approximate the value.

On output,

- all negative values are prefixed with a minus sign. A positive value or zero can have a plus sign as controlled by the S, SS, and SP nonrepeatable edit descriptors.
- the I/O system right justifies all numeric values, and when necessary, pads with blanks on the left.
- if the characters in the output field exceed the field width w, the I/O system produces a field of w asterisks (*).

### 16.6.1. Floating-point Editing, D and E

The Dw.d and Ew.d[Ee] edit descriptors describe a field whose width is "w" positions with a fractional part containing "d" digits (unless the scale factor "k" > 1), and an exponent of "e" digits. When using the Dw.d and Ew.d[Ee] edit descriptors, the matching I/O list item must be a floating-point type.

The following rules apply to Dw.d and Ew.d[Ee]:

On input,

- the input field is identical to that of the Fw.d edit descriptor; "e" has no effect.

On output,

- if the scale factor "k" = 0, the output field has the form [+–][0].x1x2x3...xd exp where x1x2x3...xd are the "d" most significant digits of the value after rounding, and "exp" is a decimal exponent. The form of "exp" depends on its absolute value as shown in the following table. Note that Dw.d and Ew.d are not valid if |exp| > 999.

**D and E Editing - Exponent Forms**

| Edit Descriptor | Absolute Value of exp | Exponent Form |
|---|---|---|
| Ew.d | \|exp\|≤99 | E+z1z2 or 0+z1z2 |
| | 99<\|exp\|≤999 | +z1z2z3 |
| Ew.d[Ee] | \|exp\|≤(10**e)–1 | E+z1z2z3...ze |
| Dw.d | \|exp\|≤99 | D+z1z2 or E+z1z2 or +0z1z2 |
| | 99<\|exp\|≤999 | +z1z2z3 |
| | | (z is a digit) |

- The scale factor "k" also controls decimal normalization (see the kP nonrepeatable edit descriptor) according to the following rules:
- If –d < k < 0, then the output field has |k| leading zeros and d – |k| significant digits following the decimal point.
- If 0 < k < d+2, then the output field has k significant digits to the left of the decimal point and d – k + 1 significant digits to the right of the decimal point.
- No other values of k are valid.

**Examples of Dw.d and Ew.d[Ee] Editing**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| D9.2 | 999999.99 | 9.9999999D+05 |
| D14.4 | 20583.4077D+03 | 2.05834077D+07 |
| E9.2 | 3.31587E2 | 3.31587E2 |
| E10.3 | ␣␣81.081E3 | 8.1081E4 |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| D15.3 | 0.0181 | ␣␣␣␣␣␣0.181D–01 |
| D8.1 | 0 | ␣0.0D+00 |
| E10.2 | 1216641.731 | ␣␣0.12E+07 |
| E12.4 | 1216641.731 | ␣␣0.1217E+07 |

### 16.6.2. Floating-point Editing, F

The Fw.d edit descriptor describes a field whose width is w positions, with a fractional part containing d digits. When using Fw.d, the I/O list item must be a floating-point type.

The following rules apply to Fw.d:

On input,

- the field can contain an optional sign, followed by digits that can optionally contain a decimal point. If there is no decimal point, the I/O system interprets the rightmost "d" digits in the field as the fractional part. The input field can contain more digits than are needed by the processor to approximate the value.
- the input field can be followed by an exponent expressed as
  - a signed integer constant
  - the character D or E followed by zero or more blanks, followed by an optionally signed integer constant.

On output,

- the output field consists of any necessary blanks followed by digits containing a decimal point that represent the internal value rounded to "d" fractional digits and modified by the established scale factor (see the kP nonrepeatable edit descriptor).
- if the value is negative, the output field is prefixed with a minus sign. If the value is positive, the output field can have an optional plus sign.
- if the absolute value of the internal data is less than one, the output field can have an optional zero immediately to the left of the decimal point.

**Examples of Fw.d Editing**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| F7.2 | 6671878 | 66718.78 |
| F9.5 | –10.24E+2 | –1024.0 |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| F9.4 | 2.71828 | ␣␣b2.7183 |
| F8.3 | –98.87314 | ␣–98.873 |

### 16.6.3. Floating-point Editing, G

The Gw.d[Ee] edit descriptors describe a field whose width is w positions with a fractional part containing "d" digits (unless the scale factor k > 1), and an exponent of "e" digits. When using the Gw.d[Ee] edit descriptor, the I/O list item must be a floating-point type.

The following rules apply to Gw.d[Ee]:

On input,

- the input field is identical to that of the Fw.d edit descriptor; "e" has no effect.

On output,

- the output field depends on the M, the magnitude of the I/O list item.
- If M < 0.1 or M \(>= 10**d, then the output field is identical to that produced by Gw.d[Ee] using the current scale factor k.
- If 0.1 ≤ M < 10**d, then M is inside the range that permits Fw.d editing. In this case, the I/O system ignores the current scale factor k, and M produces an equivalent conversion as shown in the following table.

**Gw.d[Ee] Conversion when 0.1 ≤ M < 10**d**

| Value of M | Equivalent Conversion |
|---|---|
| 0.1≤M< 1 | F(w–n).d,n(␣) |
| 1≤M<10 | F(w–n).(d–1),n(␣) |
| 10**(d–2)≤M<10**(d–1) | F(w–n).1,n(␣) |
| 10**(d–1)≤M<10**d | F(w–n).0,n(␣) |

n(b) = 4 blank spaces for Gw.d
n(b) = e+2 blank spaces for Gw.d[Ee]

**Examples of Gw.d[Ee] Editing**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| G7.2 | 6671878 | 66718.78 |
| G9.5 | –10.24E+2 | –1024.0 |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| G12.5 | 864.50695 | ␣␣864.51␣␣␣ |
| G12.5 | –1910.66666 | ␣–1910.7␣␣␣ |

### 16.6.4. Complex Editing

A complex data value is represented as a pair of values: a real part and an imaginary part. Editing of complex data is accomplished by the successive interpretation of two D, E, F, or G edit descriptors. The first descriptor describes the real part, and the second descriptor describes the imaginary part.

The two edit descriptors can be different, and nonrepeatable edit descriptors can appear between any two successive D, E, F, or G edit descriptors.

**Examples of Complex Editing**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| 2F9.3 | 28829809856777.765 | 288298.098, 56777.765 |
| D9.2,E9.2 | 999999.993.31587E2 | 9.9999999D+05, 3.31587E2 |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| D8.1,D8.3 | 0.0, 3.14159 | ␣0.0D+00.314D+01 |
| 2E9.2 | 283.2394, 0.129312 | ␣0.28E+03␣0.13E+00 |

### 16.6.5. Integer Editing

The Iw and Iw.m edit descriptors describe a field whose width is "w" positions. When using Iw or Iw.m, the I/O list must be of type INTEGER, and consist of at least one digit.

The following rules apply to Iw and Iw.m:

On input,

- the input field can be an optionally signed integer constant. Leading blanks are treated as described above.

On output,

- for Iw, the output field consists of an integer constant. If the value is positive, it can be prefixed with an optional plus sign. If the value is negative, it is prefixed by a minus sign. Leading blanks are treated as described above.

- for Iw.m, the output field is identical to that produces by Iw, except that it must have at least "m" digits, and if necessary, have leading zeros. The value of "m" cannot exceed "w". If m = 0 and the internal value of the I/O list item is zero, the output field consists of all blanks regardless of any sign control in effect.

**Examples of Iw and Iw.m Editing**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| I5 | +128 | 128 |
| I5 | –999 | –999 |
| I5 | ␣␣␣␣0 | 0 |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| I4 | +128 | ␣128 |
| I4 | –999 | –999 |
| I5.3 | 1 | ␣␣001 |
| I5.3 | –1 | ␣–001 |

### 16.6.6. Octal Editing

The O edit descriptor inputs or outputs data in octal format (base 8). It can be used with any data type.

**Syntax:** O*w*[.*m*]

where *w* specifies the number of octal digits to be transferred, and *m* optionally specifies the minimum number of output digits, zero-padded as necessary.

If *w* is not specified, the default field width for that data type will be used, as shown in the table below.

| Data Type | Default | Data Type | Default |
|---|---|---|---|
| BYTE | 7 | | |
| INTEGER*2 | 7 | LOGICAL*2 | 7 |
| INTEGER*4 | 12 | LOGICAL*4 | 12 |
| REAL*4 | 12 | REAL*8 | 23 |

The following rules apply to the octal edit descriptor:

On input,

- *w* octal digits are transferred from the external field to the target I/O list element. The external field may contain only valid unsigned octal numbers (digits 0-7).
- a blank external field will default to all zeroes.
- invalid characters in the external field will result in an error.
- the *m* optional parameter is not valid on input, and is ignored.

On output,

- the first *w* digits of the unsigned octal representation of the corresponding I/O list element are transferred to the external field.
- if the resulting output is less than *w* digits, the external field is padded with spaces on the left.
- if the resulting output is larger than *w*, the external field will be filled with *w* asterisks (*), indicating an overflow result.
- if *m* is specified and the resulting output is less than *m* digits in length, the external field will be right-justified, zero-filled up to a total of *m* digits. Any remaining locations are blank-filled up to the total field length, *w*.

**Example:**

| | Output Processing | |
|---|---|---|
| **Format** | **Value (decimal)** | **Result** |
| O5 | 511 | ␣␣777 |
| O2 | 511 | ** |
| O7 | -1 | ␣177777 (assuming 2-byte integer) |
| O5.3 | 17 | ␣␣021 |
| O5.5 | 17 | 00021 |
| O5.0 | 0 | ␣␣␣␣␣ |

**F77 Compatibility Notes:**

The Ow[.m] edit descriptor is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 16.6.7 Hexadecimal Editing

The Z edit descriptor inputs or outputs data in hexadecimal format (base 16), and can be used with any data type. Valid hexadecimal digits are 0-9, a-f and A-F inclusive. Upper and lower case letters are equivalent.

**Syntax:** Z*w*[.*m*]

where *w* specifies the number of hexadecimal digits to be transferred, and *m* optionally specifies the minimum number of output digits, zero-padded as necessary.

If *w* is not specified, the default field width for that data type will be used, as shown in the table below.

| Data Type | Default | Data Type | Default |
|---|---|---|---|
| BYTE | 7 | | |
| INTEGER*2 | 7 | LOGICAL*2 | 7 |
| INTEGER*4 | 12 | LOGICAL*4 | 12 |
| REAL*4 | 12 | REAL*8 | 23 |

The following rules apply to the hexadecimal edit descriptor:

On input,

- *w* hexadecimal digits are transferred from the external field to the target I/O list element. The external field may contain only valid unsigned hexadecimal digits (numbers 0-9 and letters A-F, a-f).
- a blank external field will default to all zeroes.
- invalid characters in the external field will result in an error.
- the *m* optional parameter is not valid on input, and is ignored.

On output,

- the first *w* digits of the unsigned hexadecimal representation of the corresponding I/O list element are transferred to the external field.
- if the resulting output is less than *w* digits, the external field is padded with spaces on the left.
- if the resulting output is larger than *w*, the external field will be filled with *w* asterisks (*), indicating an overflow result.
- if *m* is specified and the resulting output is less than *m* digits in length, the external field will be right-justified, zero-filled up to a total of *m* digits. Any remaining locations are blank-filled up to the total field length, *w*.

**Example:**

| | **Input Processing** | |
|---|---|---|
| **Format** | **Input** | **Result** |
| Z2 | 0FFF | 0F |
| Z4 | 0FFF | 0FFF |
| Z6 | 0FFF | 000FFF |

| | **Output Processing** | |
|---|---|---|
| **Format** | **Internal value (decimal)** | **Result** |
| Z2 | 4095 | ** (overflow) |
| Z6 | 4095 | ␣␣␣FFF |
| Z6.4 | 4095 | ␣␣0FFF |

**F77 Compatibility Notes:**

The Zw[.m] edit descriptor is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 16.7. Logical Editing

The Lw edit descriptor describes a field whose width is w positions. When using Lw, the I/O list item must be of type LOGICAL.

The following rules apply to Lw:

On input,

- the field can contain optionally contain leading blanks and a decimal point, followed by the character T for true or F for false. The logical constants .TRUE. and .FALSE. are also acceptable as input.

On output,

- the output field contains (w–1) leading blanks, followed by the character T if the value is true, or F if it is false.

**Examples of Lw Editing**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| L1 | T | .TRUE. |
| L3 | ␣␣T | .TRUE. |
| L7 | .FALSE. | .FALSE. |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| L4 | .TRUE. | ␣␣␣T |
| L1 | .FALSE. | F |

## 16.8. Using Nonrepeatable Edit Descriptors

Nonrepeatable edit descriptors are not associated with specific data items in the I/O list. Instead, they control such things as column position, spacing, sign control, blank control, and line termination.

**Nonrepeatable Edit Descriptors**

| Syntax | Type of Descriptor |
|---|---|
| 'c1,c2,...cn' | Apostrophe (Literal-string) Descriptor |
| nHc1,c2,...cn | Hollerith-string Descriptor |
| BN | Blank-control Descriptor |
| BZ | Blank-control Descriptor |
| kP | Scale-factor Descriptor |
| S | Sign-control Descriptor |
| SP | Sign-control Descriptor |
| SS | Sign-control Descriptor |
| Tc | Position Descriptor |
| TLc | Position Descriptor |
| TRc | Position Descriptor |
| nX | Position Descriptor |
| / | Line-termination Descriptor |
| : | Conditional line-termination Descriptor |
| Q | Read remainder of record |
| &$ | Carriage Control Editing |

In this table, "c" is any printable ASCII character, "n" is a positive integer constant, and "k" is an optionally signed integer constant that represents a scale factor.

The following subsections describe how to use each nonrepeatable edit descriptor.

### 16.8.1. Apostrophe Descriptor '

The I/O system transfers literal character strings when they are enclosed in single apostrophes. This is called apostrophe editing, and is valid only on output. The field width is length of the character string. A single apostrophe inside the string must be written as two consecutive apostrophes.

Example of Apostrophe Editing

| Format | Output Result |
|---|---|
| 'Today''sbdatebis:' | Today'sbdatebis: |

### 16.8.2. Hollerith Descriptor

The nH edit descriptor is an alternative method for transferring literal character strings. nH causes the I/O system to transfer "n" characters following the H. Like apostrophe editing, it is valid only on output.

If the character string contains a single apostrophe, it is counted as one character when specifying "n".

Examples of Hollerith Editing

| Format | I/O List Item | Output Result |
|---|---|---|
| 3H | ABC | ABC |
| 4H | IT'S | IT'S |

### 16.8.3. Q Editing

The Q edit descriptor is used to read the remaining number of characters from the input record. It can be used to validate an input record length, or to clear out the input stream after all required data has been processed. The Q descriptor has no parameters associated with it, and is specified in a FORMAT statement by the single letter 'Q'.

The Q descriptor returns the number of characters read from the input stream. If Q is the first descriptor in a FORMAT statement, the actual input record length with be returned. The Q descriptor can be used only with INTEGER or LOGICAL data type list elements.

**Example:**

```
      READ (4,100) BUF
      IF (BUF.GT.80) THEN
            STOP 'Record too long'
      ELSE
            BACKSPACE 4
            READ (4,300)STRINGA,STRINGB,STRINGC
      END IF
100   FORMAT(Q)
200   FORMAT(40A1,20A1,20A1)
      ...
```

Note that the BACKSPACE is necessary because the first READ statement advanced the record pointer.

**F77 Compatibility Notes:**

The Q edit descriptor is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 16.8.4. Carriage Control Editing

The dollar sign ($) edit descriptor is only used in output formatting. It is used to change the default action specified by the first character of the record.

If the first character is a space, the $ causes the carriage return to be suppressed. This is useful for terminal I/O activity where you wish to display a user prompt, and accept input from the same line. Another use would be when multiple format statements are used, but the desired result is a single line of output.

If the first character in the record is a plus sign (+), the $ descriptor causes the output to begin at the end of the previous line, effectively appending the new record to the previous one.

The $ descriptor has no effect if the first character of the record is either 0 or 1.

**Example:**

```
      TYPE 1000
1000  FORMAT('␣Job Number?␣'$)
      ACCEPT 1010, JOBNUM
1010  FORMAT(I4)
Executing these statements will display:
Job Number?
```

The user's response (for example, 1234) can then be accepted from the same line, as shown below.

```
Job Number? 1234
```

**F77 Compatibility Notes:**

The dollar-sign ($) edit descriptor is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 16.8.5. Blank-control Descriptors BN and BZ

BN and BZ control the interpretation of blanks (other than leading blanks) on input only with D, E, F, G, and I editing; they have no effect on output.

Prior to executing any formatted I/O statement, the BLANK specifier currently in effect for the unit, determines the interpretation of blanks.

When the I/O system encounters BN in a format specification, it ignores all blank characters in any succeeding input fields. Ignoring blanks has the same effect as removing blanks, right-justifying the field, and replacing the blanks as leading blanks.

When the I/O system encounters BZ in a format specification, it treats all blank characters in succeeding input fields as zeroes.

Once specified, BN and BZ remain in effect until changed explicitly, or the I/O statement finishes executing.

Examples of BN and BZ Editing

| Format | I/O List Item | Input Result |
|---|---|---|
| BN | 12␢34␢ | 1234 |
| BZ | 12␢34␢ | 120340 |

### 16.8.6. Scale-factor Descriptor kP

The kP edit descriptor establishes a scale factor when using D, E, F, or G editing.

Prior to executing an I/O statement, the scale factor is zero. Once set with the kP descriptor, the value of k remains in effect until changed with another another kP descriptor, or the I/O statement finishes executing.

The scale factor k produces the following effects:

On input,

- "k" has no effect with D, E, F, and G editing if there is an exponent in the field.
- with D, E, F, and G editing, the externally represented number equals the internal representation multiplied by 10**k.

On output,

- with D and E editing, the mantissa is multiplied by 10**k, which moves the decimal point "k" positions to the right (or left, if negative), and the exponent is reduced by "k".
- with F editing, the externally represented number equals the internal representation multiplied by 10**k.
- with G editing, "k" is ignored unless the output value is outside the range of F editing. If E editing is required, "k" has the same effect as described for E editing.

***Note:*** When kP immediately follows a D, E, F, or G edit descriptor, a comma is not required between items.

Table 14-15 illustrates the effect of a scale factor when used with floating-point edit descriptors.

**Examples of Floating-point Editing with a Scale Factor**

**On Input**

| Format | I/O List Item | Input Result |
|---|---|---|
| 2PF10.4 | ␢␢125.63␢␢ | 1.2563 |

**On Output**

| Format | I/O List Item | Output Result |
|---|---|---|
| 2PF10.2 | 104.12345 | 10412.345 |
| 2PD15.3 | 0.0181 | 18.10D–03 |

### 16.8.7. Sign-control Descriptors S, SP, and SS

The S, SS, and SP edit descriptors control the optional plus sign character in numeric output fields. S, SS, and SP affect the D, E, F, G and I edit descriptors on output only; they have no effect on input.

When executing any formatted I/O statement, the I/O system normally has the option to produce a plus sign in numeric output fields. An SP edit descriptor directs the I/O system to always produce a plus sign in any subsequent position that normally contains an optional plus sign.

An SS edit descriptor directs the I/O system to always suppress a plus sign in any subsequent position that normally contains an optional plus sign.

An S edit descriptor restores to the I/O system the option of producing a plus sign in numeric output fields.

Examples of Editing with Sign-control Descriptors

| Format | I/O List Item | Output Result |
|---|---|---|
| SS | 5 | 5 |
| SP | 5 | +5 |

### 16.8.8. Position Descriptors Tc, TLc, TRc, and nX

The Tc, TLc, TRc, and nX edit descriptors determine the position at which the I/O system transfers the next character to or from a record.

Tc specifies that transfer of the next character to or from a record occurs at the cth position.

TRc specifies that transfer of the next character to or from a record occurs at c positions to the right of the current position.

TLc specifies that transfer of the next character to or from a record occurs at c positions to the left of the current position. If the current position is less than or equal to c, TLc transfers the next character at position one of the current record.

nX specifies that transfer of the next character to or from a record occurs at n positions to the right of the current record.

The following rules apply to the position descriptors:

On input,

- T can specify a position in either direction from the current position. This allows the I/O system to process part of a record more than once, possibly with different editing.
- nX can specify a position beyond the last position in a record if no characters are transferred from such a position.

On output,

- when the I/O system transfers characters to positions at or following the position specified by Tc, TRc, TLc, or nX, any positions that are skipped and not previously filled are padded with blanks.
- a Tc, TRc, TLc or nX edit descriptor cannot replace an existing character within a record, but they can affect position such that subsequent editing causes a replacement.

### 16.8.9. Line-termination Descriptor /

The line-termination descriptor / indicates the end of data transfer on the current record.

The following rules apply to the line-termination descriptor:

On input,

- if the file is connected for sequential access, the I/O system skips the rest of the current record, and positions the file at the initial point of the next record, which then becomes the current record.
- if the file is positioned at the initial point of a record, the I/O system skips the entire record.

On output,

- if the file is connected for sequential access, the I/O system creates a new record, which then becomes the current and last record in the file.
- if the file is connected for direct access, the I/O system increments the current record number by one, and positions the file at the initial point of the new record, which then becomes the current record.
- the I/O system can output an empty record. If the file is connected for direct access or is an internal file, the record contains blanks.

*Note:* A comma is not required before or after the / and any I/O list items.

The following example illustrates the line-termination descriptor.

| Format | Output Result |
|---|---|
| (1X,'ABC'//1X,'DEF') | ␣ABC<br>␣<br>␣<br>␣DEF |

### 16.8.10. Conditional Line-termination Descriptor, :

The conditional line-termination descriptor ":" terminates format control if there are no more items in the I/O list. If there are remaining items in the I/O list, the I/O system ignores the ":" descriptor.

The following example illustrates the conditional line-termination descriptor.

```
      Print Statement  Output Result
      PRINT 10,5
10    FORMAT (IX,'I=',I2,'␣J=',I2)    I=␣5␣J=

      PRINT 20,6
20    FORMAT (IX,'I=',I2,:'␣J=',I2)   I=␣6
```

## 16.9. List-directed Formatting

List-directed formatting is specified by an asterisk (*) as the format specification in an I/O statement. An explicit FORMAT statement is not required.

A list-directed file is an external file containing list-directed records. A list-directed record is a sequence of characters that are either values or value separators.

Each value can be

- a constant
- a null value
- one of the forms r*c or r*

where r is an positive integer constant repeat-factor, and c is a character value. r*c is equivalent to r successive occurrences of c; r* is equivalent to r successive null values. Neither form can contain any embedded blanks, other than those within c.

A value separator can be

- a comma, optionally preceded or followed by one or more contiguous blanks
- a slash (/), optionally preceded or followed by one or more contiguous blanks
- one or more contiguous blanks between two constants, or following the last constant

The following rules apply to list-directed formatting:

- The I/O system treats blanks as separators, so embedded blanks are allowed only inside character strings.
- The I/O system treats the end of a record as a blank, except inside a character string.
- The end of a record following any separator with or without any intervening blanks does not imply a null value.
- There are two ways to specify a null value:
- with the r* form
- by having no characters precede the first value separator, or appear between the successive value separators.

### 16.9.1. List-directed Input

When the I/O system executes a list-directed READ statement, it begins a new record, and formats each input value using the data type and field width of the corresponding I/O list item to generate an equivalent edit descriptor.

The following rules apply to list-directed input:

- When the I/O system encounters a null value while executing a list-directed input statement, the null value does not affect the corresponding I/O list item. The item retains its value, or if it is undefined, it remains undefined.
- A null value cannot appear as the real or imaginary part of a complex constant, but a single null value can represent a whole complex constant.

- When the I/O system encounters a slash (/) as a value separator while executing a list-directed input statement, it stops executing the statement at that point, and treats any further items in the I/O list as null values.
- If the I/O list item is of type CHARACTER*n, the input value must be a nonempty character string enclosed in single apostrophes. Commas, blanks, and slashes (/) are all valid inside character-string constants. An apostrophe inside a character string must be written as two consecutive apostrophes.

A character-string constant can continue from the end of one record to the beginning of the next for as many records as are needed. The end of a record does not cause a blank character to appear in the constant.

If w is the field width and the input value is CHARACTER*n, the I/O system transfers characters as follows:

- if $w \leq n$, the leftmost w characters are transferred
- if $w > n$, the leftmost n characters are transferred and the remaining $w - n$ positions are padded with blanks

This is the same effect as assigning the I/O list item in an ordinary assignment statement.

- If the I/O list is of type COMPLEX*8 or COMPLEX*16, the input value consists of a pair of numeric input fields separated by a comma, and enclosed in parentheses. The first field contains the real part of the complex constant, and the second field contains the imaginary part. Each field can be preceded or followed by one or more contiguous blanks.
- If the I/O list item is of type LOGICAL, the input value cannot contain any commas or slashes (/) embedded among the optional characters after the T or F.
- If the I/O list item is of type REAL*4 or REAL*8, the input value has the form of a numeric input field suitable for F editing. That is, it has no fractional digits unless a decimal point appears in the field.

The table below summarizes the correspondence between the data type of the I/O list item and the equivalent edit descriptors.

**Equivalence of Edit Descriptors Using List-directed Input**

| Data Type of Input Item | Equivalent Edit Descriptor | |
|---|---|---|
| CHARACTER*n | Aw | if w≤n |
| | An,(w–n)X | if w > n |
| COMPLEX*8 or COMPLEX*16 | (Fw.0,Fw.0) | |
| LOGICAL | Lw | |
| REAL*4 or REAL*8 | Fw.0 | |

### 16.9.2. List-directed Output

When the I/O system executes a list-directed WRITE or PRINT statement, it begins a new record, and formats each output value using the data type and field width of the corresponding I/O list item to generate an equivalent edit descriptor.

The following rules apply to list-directed output:

- Each output record begins with a blank to provide carriage control when printing.
- Output values are separated by one or more blanks.

- The I/O system treats blanks as separators, so embedded blanks are allowed only inside character strings.
- The I/O system treats the end of a record as a blank, except inside a character string constant is not enclosed in single apostrophes, or preceded or followed by a value separator. The I/O system can insert a blank for carriage control if a record begins with the continuation of a character constant from the preceding record.
- When the I/O system outputs a COMPLEX*n constant, the constant is enclosed in parentheses with a comma separating the real and imaginary parts, which are edited according the rules for REAL*k values where k = n/2.
- The I/O system outputs an INTEGER*n value using the Iw edit descriptor.
- The I/O system outputs a LOGICAL constant using T for the value true or F for the value false.

The I/O system outputs a REAL*n constant, the constant is represented using G format.

The table below summarizes the correspondence between the data type of the I/O list item and the equivalent edit descriptors.

**Equivalence of Edit Descriptors Using List-directed Output**

| Data Type | Output Format |
|---|---|
| LOGICAL | I5 |
| INTEGER*2 | I7 |
| INTEGER*4 | I12 |
| REAL*4 | 1PG15.7 |
| REAL*8 | 1PG25.16 |
| COMPLEX*8 | 1X,'(',1PG14.7,',', 1PG14.7,')' |
| COMPLEX*16 | 1X,'(',1PG25.16,',',1PG25.16,',)' |
| CHARACTER*n | 1X, An |

# Chapter 17
# System Subroutines, Built-ins and Intrinsic Functions

## 17.1. System Routines

FORTRAN supplies routines which can be called in the same way as a user-written routine. A summary of the routines is shown in the table below, preceding the detailed descriptions for each routine. In this section, all integer values must be INTEGER*4.

| Routine | Description |
|---|---|
| DATE | Returns system date as a character string |
| ERRSNS | Returns information on most recent runtime error |
| EXIT | Terminates program, closes files and exits to operating system |
| IDATE | Returns integer values for month, day and year |
| MVBITS | Copies a bit pattern from one location to another |
| RAN | Returns a pseudo-random number between 0.0 and 1.0 inclusive |
| SECNDS | Returns difference between supplied value and current system time |
| TIME | Returns system time as a character string |

**F77 Compatibility Notes:**

All of the system subroutines listed in this section are available only in VMS compatibility mode, with the exception of MVBITS, which is also available when DOD MIL compatibility mode is selected. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.1. DATE Subroutine

The DATE subroutine takes no arguments, and returns the current system date as a 9-character string value.

**Syntax:** CALL DATE (datebuf)

where *datebuf* is a 9-byte variable where the system date will be stored. The *datebuf* variable may be a character string, substring, array or array element.

The date returned is in the form dd-mmm-yy, where *dd* is the 2-digit day, *mmm* is a 3-letter month abbreviation, and *yy* is the last two digits of the current year. For example, the string "03-JUN-88" would be returned if the current system date was June 3rd, 1988.

**F77 Compatibility Notes:**

The DATE subroutine is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.2. IDATE Subroutine

The IDATE subroutine takes no arguments, and returns the current system date as three integer values, representing month, day, and year.

**Syntax:** CALL IDATE (imon,iday,iyear)

where *imon* will contain the month number, *iday* will contain the day, and *iyear* will contain the integer value for the last two digits of the current system year.

For example, a system date of March 27th, 1990 would return imon=3, iday=27 and iyear=90.

**F77 Compatibility Notes:**

The IDATE subroutine is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.3. ERRSNS Subroutine

The ERRSNS subroutine takes no arguments, and returns information about the most recent run-time error detected.

**Syntax:** CALL ERRSNS (ierrnum,iosts,iostv,iunit,icondval)

where *ierrnum* will contain the most recent FORTRAN error number, or zero (0) if no error has occurred since the previous call, or start of execution and *iunit* will contain the unit number on which the last error occurred. The parameters iosts and *iostv* are unchanged, and included only for compatibility with VAX/VMS Fortran. The parameter *icondval* is set to zero.

Refer to your FORTRAN User's Manual for a list of Fortran error numbers and their descriptions.

**F77 Compatibility Notes:**

The ERRSNS subroutine is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.4. EXIT Subroutine

The EXIT subroutine takes an optional status return code argument, and is called to perform a "clean" exit, closing all open files, terminating program execution, and returning control to the operating system.

**Syntax:** CALL EXIT [(istatus)]

where *istatus* is an optional integer argument used to return status code information upon program termination.

Refer to your User's Guide or Operating System Manual for return status code conventions.

**F77 Compatibility Notes:**

The EXIT subroutine is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.5. MVBITS Subroutine

The MVBITS subroutine copies data from one integer variable to another, allowing the user to specify starting bit positions and number of bits to copy.

**Syntax:** CALL MVBITS (src, sstart, len, dst, dstart)

where *src* is a variable that will be the source of the bit field to be copied, *sstart* is the starting bit position within *src* and *len* is the total number of bits to be copied to the destination variable. *dst* is a variable where the designated bits of *src* will be copied to, starting at bit location *dstart*.

All MVBITS parameters are of data type INTEGER*4. Bit locations are determined from right (bit 0) to left (bit 31). The values for (*sstart+len*) and (*dstart+len*) must be less than 32.

**Example:**

```
      INTEGER*2 isrc,idst
C     isrc initialized to 111111111111111
C     idst initialized to 000000000000000
      isrc = '77777'O
      idst = '00000'O
C     copy last 4 bits from isrc to idst
      CALL MVBITS (isrc,3,4,idst,3)
C     idst now contains 000000000001111
```

**F77 Compatibility Notes:**

The MVBITS subroutine is only available in VMS compatibility mode, or when DOD MIL compatibility mode is selected. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.6. RAN Function

The RAN function takes a single seed argument and returns a pseudo-random number between 0.0 and 1.0 inclusive. Successive calls to RAN will produce a uniformly distributed set of numbers.

**Syntax:** ranval = RAN (iseed)

where *iseed* is an INTEGER*4 variable used as an initial seed value for the random number generator. RAN will alter the value of *iseed*, using the formula

```
iseed = 69069 * iseed + 1 (mod 2**32)
```

so that subsequent calls to RAN will return a different *ranval* number. As a general rule, different *iseed* values should be used for each execution so that distinct sets of random values are obtained. RAN calculates a value for *ranval* by taking the high order 24 bits of *iseed*, and converting that to a floating-point number.

**F77 Compatibility Notes:**

The RAN function is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.7. SECNDS Function

The SECNDS function accepts a single REAL*4 argument, and returns the difference between the supplied value and the current system time (represented as seconds since midnight). SECNDS is useful for calculating elapsed time during program execution. SECNDS is accurate to the resolution of the system clock.

**Syntax**: delta = SECNDS (t)

where *delta* is a REAL*4 variable that will contain the difference between the current system time, and the user-supplied value, *t*. Successive calls to SECNDS will return the elapsed time, in seconds, since the previous call, as illustrated in the example, below.

**Example:**

```
      REAL*4 a,b
      a = SECNDS(0.0)
      TYPE 10
10    FORMAT(" Enter carriage return",$)
      ACCEPT 15
15    FORMAT(Q)
      b = SECNDS(a)
      TYPE 20,b
20    FORMAT(" That took ",F4.2," seconds")
      END
```

**F77 Compatibility Notes:**

The SECNDS function is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.1.8. TIME Subroutine

The TIME subroutine takes no arguments, and returns the current system time in hours, minutes and seconds.

**Syntax:** CALL TIME (timebuf)

where *timebuf* is an 8-byte variable that will receive the system time, in 24-hour format as an ASCII string.

The form of *timebuf* is hh:mm:ss, where *hh* is a 2-digit hour, *mm* is a 2-digit minute and *ss* a 2-digit value for seconds. For example, a call to TIME one minute and 30 seconds after noon will return "12:01:30"

**F77 Compatibility Notes:**

The TIME subroutine is not available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

## 17.2. Built-In Functions

The following built-in functions are provided to facilitate communications between FORTRAN and non-FORTRAN subprograms. A summary of the built-in functions is shown in the table below.

| Function | Description |
|---|---|
| %VAL | Pass arguments by value |
| %REF | Pass arguments by reference |
| %DESCR | Equivalent to %REF |
| %LOC | Return internal address of storage element |

The first three functions, %VAL, %REF and %DESCR are used to pass arguments in forms other than standard FORTRAN. The fourth function, %LOC, is provided to obtain the internal storage address of an element.

The functions %VAL, %REF and %DESCR must only be used within an actual CALL statement or function argument list.

**F77 Compatibility Notes:**

None of the built-in functions described below are available in F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.2.1. %VAL Built-in Function

The %VAL function is used to pass arguments by value.

**Syntax:** %VAL (arg)

where *arg* is to be passed as a 32-bit value. If *arg* is shorter than 32 bits, the value is sign-extended to the full 32-bits. Refer to the ZEXT function if zero extension is required.

Only INTEGER, REAL*4 or LOGICAL data types may be passed by value. Array names and procedure values must be passed by reference.

### 17.2.2. %REF Built-in Function

The %REF function is used to pass arguments by reference.

**Syntax:** %REF (arg)

where *arg* is to be passed by reference. This is the default method for argument passing, and can be used with any data type.

### 17.2.3. %DESCR Built-in Function

The %DESCR function is currently equivalent to the %REF function, described above. Argument passing by descriptor may be provided in a future release.

### 17.2.4. %LOC Built-in Function

The %LOC function returns the internal address of the storage element *arg* as an INTEGER*4 value.

Syntax: %LOC (arg)

where arg is an array name, memory reference, aggregate reference or external procedure name.

## 17.3. Fortran Intrinsic Functions

This section presents the Fortran intrinsic functions in alphabetical order according to the generic function name. Each function description contains a brief identification of function usage, a syntax specification, a description of the function, and a listing of specific names.

You can reference an intrinsic function with the generic name or you can use a specific function name. The specific name list identifies the data type for the arguments and the type of return value. Specific names reference a particular part of a function to perform a more specific operation. A dash in the specific name column indicates that you must use the generic name for the corresponding specific operation.

The Fortran intrinsic functions are documented assuming VMS compatibility mode, which is the default mode on most systems. Some function names differ when F77 compatibility mode is used. If a function is different, or unavailable in F77 compatibility mode, this information will be contained in a notes section following the function description. Please refer to your User's Guide for details on compiler options and default modes on your system.

**Summary of FORTRAN Intrinsic Functions**

| Function | Description |
|---|---|
| ABS | Absolute value |
| ACOS | Trigonometric arccosine (radians) |
| ACOSD | Trigonometric arccosine (degrees) |
| AIMAG | Imaginary part of complex value |
| AINT | Truncate real to integer value |
| AMAX0 | Select largest value |
| AMIN0 | Select smallest value |
| ANINT | Nearest whole number |
| ASIN | Trigonometric arcsine (radians) |
| ASIND | Trigonometric arcsine (degrees) |
| ATAN | Trigonometric arctangent (radians) |
| ATAND | Trigonometric arctangent (degrees) |
| ATAN2 | Trigonometric arctangent of quotient (radians) |
| ATAN2D | Trigonometric arctangent of quotient (degrees) |
| BTEST | Test bit |
| CHAR | Convert integer to ASCII character equivalent |
| CMPLX | Convert integer or real to COMPLEX |
| CONJG | Conjugate of complex argument |
| COS | Trigonometric cosine (radians) |
| COSD | Trigonometric cosine (degrees) |
| COSH | Trigonometric hyperbolic cosine |
| DBLE | Numeric data type conversion |
| DCMPLX | Convert integer or real to COMPLEX*16 |
| DFLOAT | Convert integer to REAL*8 |

**Summary of FORTRAN Intrinsic Functions**

| Function | Description |
|---|---|
| DIM | Positive difference |
| DPROD | Double precision product |
| EXP | Exponential |
| FLOAT | Convert integer to REAL*4 |
| IABS | Return absolute value of integer |
| IAND | Logical AND operation |
| IBCLR | Clear specified bit |
| IBITS | Extract bit field |
| IBSET | Set specified bit |
| ICHAR | Convert ASCII to corresponding integer value |
| IDIM | Positive difference between integers |
| IDINT | Convert real number to integer |
| IDNINT | Nearest integer |
| IEOR | Exclusive OR operation |
| IFIX | Convert real number to integer |
| INDEX | Index of substring |
| INT | Convert real or complex to integer |
| IOR | Inclusive OR operation |
| ISHFT | Shift bits logically |
| ISHFTC | Shift bits logically with wrap |
| LEN | Character string length |
| LGE | Compare strings (greater than or equal to) |
| LGT | Compare strings (greater than) |
| LLE | Compare strings (less than or equal to) |
| LLT | Compare strings (less than) |
| LOG | Natural logarithm |
| LOG10 | Common logarithm |
| MAX | Select largest number |
| MAX0 | Select largest integer number |
| MAX1 | Select largest real and convert to integer |
| MIN | Select smallest number |
| MIN0 | Select smallest integer number |
| MIN1 | Select smallest real and convert to integer |
| MOD | Remainder |
| NINT | Nearest integer |
| NOT | Bit complement operation |
| REAL | Convert to REAL*4 (DREAL converts to REAL*8) |
| SIGN | Transfer sign |
| SIN | Trigonometric sine (radians) |
| SIND | Trigonometric sine (degrees) |
| SINH | Trigonometric hyperbolic sine |
| SQRT | Square root |
| TAN | Trigonometric tangent (radians) |
| TAND | Trigonometric tangent (degrees) |
| TANH | Trigonometric hyperbolic tangent |
| ZEXT | Zero-extend |

### 17.3.1. ABS Function

**Usage:** absolute value

**Syntax:** x = ABS(number)

The ABS function returns the absolute value of an integer, real, or complex number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIABS | INTEGER*2 | INTEGER*2 |
| JIABS | INTEGER*4 | INTEGER*4 |
| ABS | REAL*4 | REAL*4 |
| DABS | REAL*8 | REAL*8 |
| CABS | COMPLEX | REAL*16 |
| CDABS | COMPLEX*16 | REAL*8 |

**See Also:** IABS, INT

**F77 Compatibility Notes:**

The CDABS function must be replaced by ZABS when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.2. ACOS Function

**Usage:** trigonometric arccosine

**Syntax:** x = ACOS(number)

The ACOS function returns the trigonometric arccosine of a real number in radians.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ACOS | REAL*4 | REAL*4 |
| DACOS | REAL*8 | REAL*8 |

**See Also:** ACOSD

### 17.3.3. ACOSD Function

**Usage:** trigonometric arccosine

**Syntax:** x = ACOSD(number)

The ACOSD function returns the trigonometric arccosine of a real number in degrees.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ACOSD | REAL*4 | REAL*4 |
| DACOSD | REAL*8 | REAL*8 |

**See Also:** ACOS

### 17.3.4. AIMAG Function

**Usage:** imaginary part of a complex value

**Syntax:** x = AIMAG(complex-number)

The AIMAG function returns the imaginary part of a complex-number with a real data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| AIMAG | COMPLEX | REAL*4 |
| DIMAG | COMPLEX*16 | REAL*8 |

### 17.3.5. AINT Function

**Usage:** truncation

**Syntax:** x = AINT(number)

The AINT function truncates a real number to an integer but maintains the original data type specification. The AINT function does not convert a number to the integer data type. If the number you want to truncate is an integer, the AINT function simply returns that integer.

If the number you want to truncate is a real number with an absolute value less than 1, the AINT function returns 0. If the number you want to truncate is a real number with an absolute value greater than 1, the AINT function returns the largest integer that does not exceed the value of the original number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| AINT | REAL*4 | REAL*4 |
| DINT | REAL*8 | REAL*8 |

**See Also:** IDINT, INT

### 17.3.6. AMAX0 Function

**Usage:** selecting the largest value

**Syntax:** x = AMAX0(number,number[,number...])

The AMAX0 function returns the largest value from a list of integer numbers and converts the data type to real. All arguments specified must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| AIMAX0 | INTEGER*2 | REAL*4 |
| AJMAX0 | INTEGER*4 | REAL*4 |

**See Also:** MAX, MAX0, MAX1

### 17.3.7. AMIN0 Function

**Usage:** selecting the smallest value

**Syntax:** x = AMIN0(number,number[,number...])

The specific function AMIN0 determines the smallest value from a list of integers and converts the data type to real. All arguments specified must be of the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| AIMIN0 | INTEGER*2 | REAL*4 |
| AJMIN0 | INTEGER*4 | REAL*4 |

**See Also:** MAX, MIN, MIN0, MIN1

### 17.3.8. ANINT Function

**Usage:** nearest whole number

**Syntax:** x = ANINT(number)

The ANINT function translates a real number to the nearest whole number value and maintains the original data type. If the number you want to translate is an integer, the ANINT function simply returns that integer.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ANINT | REAL*4 | REAL*4 |
| DNINT | REAL*8 | REAL*8 |

**See Also:** AINT, IDNINT, NINT

### 17.3.9. ASIN Function

**Usage:** trigonometric arcsine

**Syntax:** x = ASIN(number)

The ASIN function returns the trigonometric arcsine of a real number in radians.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ASIN | REAL*4 | REAL*4 |

**See Also:** ASIND

### 17.3.10. ASIND Function

**Usage:** trigonometric arcsine

**Syntax:** x = ASIND(number)

The ASIND function returns the trigonometric arcsine of a real number, expressed in degrees.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ASIND | REAL*4 | REAL*4 |
| DASIND | REAL*8 | REAL*8 |

**See Also:** ASIN

### 17.3.11. ATAN Function

**Usage:** trigonometric arctangent

**Syntax:** x = ATAN(number)

The ATAN function returns the trigonometric arctangent of a real number in radians.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ATAN | REAL*4 | REAL*4 |
| DATAN | REAL*8 | REAL*8 |

**See Also:** ATAND

### 17.3.12. ATAND Function

**Usage:** trigonometric arctangent

**Syntax:** x = ATAND(number)

The ATAND function returns the trigonometric arctangent of a real number, expressed in degrees.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ATAND | REAL*4 | REAL*4 |
| DATAND | REAL*8 | REAL*8 |

**See Also:** ATAN, ATAN2, ATAN2D

### 17.3.13. ATAN2 Function

**Usage:** trigonometric arctangent of a quotient

**Syntax:** x = ATAN2(number,number)

The ATAN2 function returns the trigonometric arctangent of a quotient in radians. The first number argument is the dividend. The second number argument is the divisor.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ATAN2 | REAL*4 | REAL*4 |
| DATAN2 | REAL*8 | REAL*8 |

**See Also:** ATAN, ATAND, ATAN2D

### 17.3.14. ATAN2D Function

**Usage:** trigonometric arctangent of a quotient

**Syntax:** x = ATAN2D(number,number)

The ATAN2D function returns the trigonometric arctangent of a quotient in degrees. The first number argument is the dividend. The second number argument is the divisor.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ATAN2D | REAL*4 | REAL*4 |
| DATAN2D | REAL*8 | REAL*8 |

**See Also:** ATAN, ATAND, ATAN2

### 17.3.15. BTEST Function

**Usage:** test bit and return TRUE or FALSE

**Syntax:** x = BTEST(buf,ibit)

The BTEST function tests the specified bit location in the supplied bit pattern, and returns a logical TRUE if the bit is set (one), and FALSE if the bit is clear (zero).

Bit number ibit is checked in the integer bit pattern buf. If this bit is set to 1, a logical TRUE is returned, otherwise, FALSE is returned.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| BITEST | INTEGER*2 | LOGICAL*2 |
| BJTEST | INTEGER*4 | LOGICAL*4 |

**See Also:** IBSET, IBCLR

**F77 Compatibility Notes:**

The specific functions BITEST and BJTEST are only available in VMS compatibility mode, or when DOD MIL compatibility mode has been selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

### 17.3.16. CHAR Function

**Usage:** character data type conversion

**Syntax:** x = CHAR(integer)

The CHAR function converts an integer value to the corresponding ASCII character representation. The integer argument for CHAR must range from 0 to 127.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| – | LOGICAL*1 | CHARACTER |
| – | INTEGER*2 | CHARACTER |
| – | INTEGER*4 | CHARACTER |

**See Also:** ICHAR

### 17.3.17. CMPLX Function

**Usage:** numeric data type conversion

**Syntax:** x = CMPLX(number[,number])

The CMPLX function converts an integer or real number to a complex number. If you use CMPLX with one argument, the function uses the argument for the real portion of the complex value. The imaginary portion becomes 0.

If you use CMPLX with two arguments, the function uses the first argument for the real portion of the complex value and the second argument for the imaginary part. Both arguments must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| – | INTEGER | COMPLEX |
| – | REAL*4 | COMPLEX |
| – | REAL*8 | COMPLEX |
| – | COMPLEX | COMPLEX |
| – | COMPLEX*16 | COMPLEX |

**See Also:** DCMPLX, REAL

### 17.3.18. CONJG Function

**Usage:** conjugate of a complex argument

**Syntax:** x = CONJG(complex-number)

The CONJG function returns the conjugate of a complex-number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| CONJG | COMPLEX | COMPLEX |
| DCONJG | COMPLEX*16 | COMPLEX*16 |

### 17.3.19. COS Function

**Usage:** trigonometric cosine

**Syntax:** x = COS(number)

The COS function returns the trigonometric cosine of a real or complex number in radians.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| COS | REAL*4 | REAL*4 |
| DCOS | REAL*8 | REAL*8 |
| CCOS | COMPLEX | COMPLEX |
| CDCOS | COMPLEX*16 | COMPLEX*16 |

**See Also:** COSD, COSH

**F77 Compatibility Notes:**

The CDCOS function must be replaced by ZCOS when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.20. COSD Function

**Usage:** trigonometric cosine

**Syntax:** x = COSD(number)

The COSD function returns the trigonometric cosine of a real or complex number expressed in degrees.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| COSD | REAL*4 | REAL*4 |
| DCOSD | REAL*8 | REAL*8 |

**See Also:** COS, COSH

### 17.3.21. COSH Function

**Usage:** trigonometric hyperbolic cosine

**Syntax:** x = COSH(number)

The COSH function returns the trigonometric hyperbolic cosine of a real number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| COSH | REAL*4 | REAL*4 |
| DCOSH | REAL*8 | REAL*8 |

### 17.3.22. DBLE Function

**Usage:** numeric data type conversion

**Syntax:** x = DBLE(number)

The DBLE function converts an integer, real, or complex number to a double precision real number. If the number you want to convert is already a double precision real number, the DBLE function simply returns that double precision number.

For a complex number, the DBLE function ignores the imaginary portion and returns the real portion converted to double precision.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| – | INTEGER | REAL*8 |
| – | REAL*4 | REAL*8 |
| – | REAL*8 | REAL*8 |
| – | COMPLEX | REAL*8 |

**See Also:** REAL

### 17.3.23. DCMPLX Function

**Usage:** numeric data type conversion

**Syntax:** x = DCMPLX(number[,number])

The DCMPLX function converts an integer or real number to a COMPLEX*16 number. If you use DCMPLX with one argument, the function uses the argument for the real portion of the complex value. The imaginary portion becomes 0.

If you use DCMPLX with two arguments, the function uses the first argument for the real portion of the complex value and the second argument for the imaginary part. Both arguments must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| – | INTEGER*2 | COMPLEX*16 |
| – | INTEGER | COMPLEX*16 |
| – | REAL*4 | COMPLEX*16 |
| – | REAL*8 | COMPLEX*16 |
| – | COMPLEX | COMPLEX*16 |
| – | COMPLEX*16 | COMPLEX*16 |

**See Also:** CMPLX, REAL

### 17.3.24. DFLOAT Function

**Usage:** numeric data type conversion

**Syntax:** x = DFLOAT(number)

The DFLOAT function converts an integer number to the REAL*8 data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| DFLOTI | INTEGER*2 | REAL*8 |
| DFLOTJ | INTEGER*4 | REAL*8 |

See Also: DBLE, FLOAT, REAL

### 17.3.25. DIM Function

**Usage:** positive difference

**Syntax:** x = DIM(number,number)

The DIM function returns the difference between two integers or real numbers, if that difference is a positive value. If the first number argument is greater than the second, DIM returns the positive difference. If the first number argument is less than the second, DIM returns 0.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIDIM | INTEGER*2 | INTEGER*2 |
| JIDIM | INTEGER*4 | INTEGER*4 |
| DIM | REAL*4 | REAL*4 |
| DDIM | REAL*8 | REAL*8 |

**See Also:** IDIM

### 17.3.26. DPROD Function

**Usage:** double precision product

**Syntax:** x = DPROD(real-number,real-number)

The DPROD function returns the product of two real number factors as a double precision real number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| DPROD | REAL*4 | REAL*8 |

### 17.3.27. EXP Function

**Usage:** exponential

**Syntax:** x = EXP(number)

The EXP function returns the constant e raised to a specified real or complex exponent. The number is the specified exponent.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| EXP | REAL*4 | REAL*4 |
| DEXP | REAL*8 | REAL*8 |
| CEXP | COMPLEX | COMPLEX |
| CDEXP | COMPLEX*16 | COMPLEX*16 |

**F77 Compatibility Notes:**

The CDEXP function must be replaced by ZEXP when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.28. FLOAT Function

**Usage:** numeric data type conversion

**Syntax:** x = FLOAT(number)

The FLOAT function converts an integer number to the REAL*4 data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| FLOATI | INTEGER*2 | REAL*4 |
| FLOATJ | INTEGER*4 | REAL*4 |

**See Also:** DBLE, DFLOAT, REAL

### 17.3.29. IABS Function

**Usage:** absolute value

**Syntax:** x = IABS(number)

The IABS function returns the absolute value of an integer number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIABS | INTEGER*2 | INTEGER*2 |
| JIABS | INTEGER*4 | INTEGER*4 |

**See Also:** ABS

### 17.3.30. IAND Function

**Usage:** logical AND operation

**Syntax:** x = IAND(int1,int2)

The IAND function performs a logical AND of two integer arguments and returns an integer result. The following two lines of code are identical in function:

```
x = int1.AND.int2
x = IAND(int1,int2)
```

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIAND | INTEGER*2 | INTEGER*2 |
| JIAND | INTEGER*4 | INTEGER*4 |

**See Also:** IOR, IEOR, NOT

**F77 Compatibility Notes:**

The IAND function must be replaced by AND when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.31. IBCLR Function

**Usage:** clear a specified bit in a bit pattern

**Syntax:** x = IBCLR(buf,ibit)

The IBCLR function returns the integer value buf with the ibit bit cleared (set to zero).

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIBCLR | INTEGER*2 | INTEGER*2 |
| JIBCLR | INTEGER*4 | INTEGER*4 |

**See Also:** BTEST, IBSET

**F77 Compatibility Notes:**

The IBCLR function is available only in VMS compatibility mode, or when DOD MIL compatibility mode has been selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

### 17.3.32. IBITS Function

**Usage:** extract a bit field from a bit pattern

**Syntax:** x = IBITS(buf,start,len)

The IBITS function returns a bit field, specified by starting bit and length, from an integer bit pattern.

The first argument, buf is the bit pattern from which len bits will be extracted, starting at bit location start.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIBITS | INTEGER*2 | INTEGER*2 |
| JIBITS | INTEGER*4 | INTEGER*4 |

**See Also:** MVBITS

**F77 Compatibility Notes:**

The IBITS function is available only in VMS compatibility mode, or when DOD MIL compatibility mode has been selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

### 17.3.33. IBSET Function

**Usage:** set a specified bit in a bit pattern

**Syntax:** x = IBSET(buf,ibit)

The IBSET function returns the integer value buf with the ibitbit set to one (1).

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIBSET | INTEGER*2 | INTEGER*2 |
| JIBSET | INTEGER*4 | INTEGER*4 |

**See Also:** BTEST, IBCLR

**F77 Compatibility Notes:**

The IBSET function is available only in VMS compatibility mode, or when DOD MIL compatibility mode has been selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

### 17.3.34. ICHAR Function

**Usage:** character data type conversion

**Syntax:** x = ICHAR(character)

The ICHAR function converts an ASCII character to its corresponding decimal integer value. The character argument for ICHAR must have a length of 1.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ICHAR | CHARACTER | INTEGER |

**See Also:** CHAR

### 17.3.35. IDIM Function

**Usage:** positive difference

**Syntax:** x = IDIM(number,number)

The IDIM function returns the difference between two integer numbers, if that difference is a positive value. If the first number argument is greater than the second, IDIM returns the positive difference. If the first number argument is less than the second, IDIM returns 0.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIDIM | INTEGER*2 | INTEGER*2 |
| JIDIM | INTEGER*4 | INTEGER*4 |

**See Also:** DIM

### 17.3.36. IDINT Function

**Usage:** numeric data type conversion, truncation

**Syntax:** x = IDINT(number)

The IDINT function converts a real number to an integer.

If the number you want to convert is a real number with an absolute value less than 1, the IDINT function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the IDINT function returns the largest integer that does not exceed the value of the original number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIDINT | REAL*8 | INTEGER*2 |
| JIDINT | REAL*8 | INTEGER*4 |

**See Also:** AINT, IFIX, INT

### 17.3.37. IDNINT Function

**Usage:** nearest integer

**Syntax:** x = IDNINT(number)

The IDNINT function converts a real number to the nearest integer value. Note that the IDNINT function converts the data type to integer.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIDNNT | REAL*8 | INTEGER*2 |
| JIDNNT | REAL*8 | INTEGER*4 |

**See Also:** AINT, ANINT, IDINT, IFIX, INT, NINT

### 17.3.38. IEOR Function

**Usage:** exclusive OR operation

**Syntax:** iresult = IEOR(int1,int2)

The IEOR function performs an exclusive OR of the two integer arguments and returns an integer result. The following two lines of code are functionally equivalent.

```
iresult = (int1.XOR.int2)
iresult = IEOR(int1,int2)
```

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIEOR | INTEGER*2 | INTEGER*2 |
| JIEOR | INTEGER*4 | INTEGER*4 |

**See Also:** AND, IOR, NOT

### 17.3.39. IFIX Function

**Usage:** numeric data type conversion, truncation

**Syntax:** x = IFIX(number)

The IFIX function converts a real number to an integer. If the number you want to convert is a real number with an absolute value less than 1, the IFIX function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the IFIX function returns the largest integer that does not exceed the value of the original number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IFIX | INTEGER | INTEGER |
| IIFIX | REAL*4 | INTEGER*2 |
| JIFIX | REAL*4 | INTEGER*4 |

**See Also:** AINT, IDINT, INT

### 17.3.40. INDEX Function

**Usage:** index of a substring

**Syntax:** x = INDEX(string,substring)

The INDEX function returns an integer value that indicates the starting position of a substring within a string.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| INDEX | CHARACTER | INTEGER |

### 17.3.41. INT Function

**Usage:** numeric data type conversion, truncation

**Syntax:** x = INT(number)

The INT function converts a real or complex number to an integer. If the number you want to convert is already an integer, the INT function simply returns that integer.

If the number you want to convert is a real number with an absolute value less than 1, the INT function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the INT function returns the largest integer that does not exceed the value of the original number.

If the number you want to convert is a complex number, the INT function applies the same rules as for real numbers to the real portion of the complex number. The INT function ignores the imaginary portion of a complex number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| INT | INTEGER | INTEGER |
| INT | REAL*4 | INTEGER |
| IINT | REAL*4 | INTEGER*2 |
| JINT | REAL*4 | INTEGER*4 |
| IIDINT | REAL*8 | INTEGER*2 |
| JIDINT | REAL*8 | INTEGER*4 |
| – | COMPLEX | INTEGER*2 |
| – | COMPLEX | INTEGER*4 |
| – | COMPLEX*16 | INTEGER*2 |
| – | COMPLEX*16 | INTEGER*4 |

**See Also:** AINT, IDINT, IFIX

### 17.3.42. IOR Function

**Usage:** inclusive OR operation

**Syntax:** iresult = IOR(int1,int2)

The IOR function performs an inclusive OR of two integers and returns an integer result. The following two lines of code are functionally equivalent.

```
iresult = (int1.OR.int2)
iresult = IOR(int1,int2)
```

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IIOR | INTEGER*2 | INTEGER*2 |
| JIOR | INTEGER*4 | INTEGER*4 |

**See Also:** AND, IEOR, NOT

**F77 Compatibility Notes:**

The IOR function must be replaced by OR when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.43. ISHFT Function

**Usage:** shift bits logically left or right

**Syntax:** iresult = ISHFT(buf,num)

The ISHFT function performs a linear shift, either right or left, of a bit pattern. Bits shifted out either end are discarded, and zero values are added on one end as old values are shifted out the other end.

The first argument, buf, is the integer value to be shifted, and the absolute value of the second argument, num, indicates the number of bits to be shifted. A positive value for num indicates a shift to the left is to be performed, a negative value indicates a shift to the right.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IISHFT | INTEGER*2 | INTEGER*2 |
| JISHFT | INTEGER*4 | INTEGER*4 |

**See Also:** ISHFTC

**F77 Compatibility Notes:**

The ISHFT function is available only in VMS compatibility mode, or when DOD MIL compatibility mode has been selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

### 17.3.44. ISHFTC Function

**Usage:** shift bits logically left or right

**Syntax:** iresult = ISHFTC(buf,num,field)

The ISHFTC function performs a linear shift, either right or left, of a bit pattern. Bits shifted out either end are shifted in at the other end.

The rightmost field bits of the variable buf will be shifted num times, where num is taken as an absolute value. A positive value for num indicates a left shift, a negative value indicates a right shift is to be performed.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IISHFTC | INTEGER*2 | INTEGER*2 |
| JISHFTC | INTEGER*4 | INTEGER*4 |

**See Also:** ISHFT

**F77 Compatibility Notes:**

The ISHFTC function is available only in VMS compatibility mode, or when DOD MIL compatibility mode has been selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

### 17.3.45. ISIGN Function

**Usage:** transfer of sign

**Syntax:** x = ISIGN(number,number)

The ISIGN function transfers the sign of the second number argument to the first number argument. ISIGN returns the absolute value of the first argument if the second argument is greater than or equal to 0, and the negative of the first argument if the second argument is less than 0.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IISIGN | INTEGER*2 | INTEGER*2 |
| JISIGN | INTEGER*4 | INTEGER*4 |

**See Also:** SIGN

### 17.3.46. LEN Function

**Usage:** character value length

**Syntax:** x = LEN(character-entity)

The LEN function returns an integer value that indicates the length of a specified character-entity.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| LEN | CHARACTER | INTEGER |

### 17.3.47. LGE Function

**Usage:** character value relational

**Syntax:** x = LGE(string,string)

The LGE function compares two strings according to the rules of the ASCII character collating sequence. LGE returns a logical true value if the first string argument equals or follows the second string argument in the collating sequence. Otherwise, LGE returns a logical false.

If the string arguments are unequal in length, the LGE function pads the shorter string on the right with blanks.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| LGE | CHARACTER | LOGICAL |

### 17.3.48. LGT Function

**Usage:** character value relational

**Syntax:** x = LGT(string,string)

The LGT function compares two strings according to the rules of the ASCII character collating sequence. LGT returns a logical true value if the first string argument follows the second string argument in the collating sequence. Otherwise, LGT returns a logical false.

If the string arguments are unequal in length, the LGT function pads the shorter string on the right with blanks.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| LGT | CHARACTER | LOGICAL |

### 17.3.49. LLE Function

**Usage:** character value relational

**Syntax:** x = LLE(string,string)

The LLE function compares two strings according to the rules of the ASCII character collating sequence. LLE returns a logical true value if the first string argument equals or precedes the second string argument in the collating sequence. Otherwise, LLE returns a logical false.

If the string arguments are unequal in length, the LLE function pads the shorter string on the right with blanks.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| LLE | CHARACTER | LOGICAL |

### 17.3.50. LLT Function

**Usage:** character value relational

**Syntax:** x = LLT(string,string)

The LLT function compares two strings according to the rules of the ASCII character collating sequence. LLT returns a logical true value if the first string argument precedes the second string argument in the collating sequence. Otherwise, LLT returns a logical false.

If the string arguments are unequal in length, the LLT function pads the shorter string on the right with blanks.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| LLT | CHARACTER | LOGICAL |

### 17.3.51. LOG Function

**Usage:** natural logarithm

**Syntax:** x = LOG(number)

The LOG function returns the natural logarithm of a real or complex number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ALOG | REAL*4 | REAL*4 |
| DLOG | REAL*8 | REAL*8 |
| CLOG | COMPLEX | COMLPEX |
| CDLOG | COMPLEX*16 | COMPLEX*16 |

**F77 Compatibility Notes:**

The CDLOG function must be replaced by ZLOG when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.52. LOG10 Function

**Usage:** common logarithm

**Syntax:** x = LOG10(number)

The LOG10 function returns the base 10 logarithm of a real number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ALOG10 | REAL*4 | REAL*4 |
| DLOG10 | REAL*8 | REAL*8 |

### 17.3.53. MAX Function

**Usage:** selecting the largest value

**Syntax:** x = MAX(number,number[,number...])

The MAX function returns the largest value from a list of integer or real numbers. All arguments that you specify must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMAX0 | INTEGER*2 | INTEGER*2 |
| JMAX0 | INTEGER*4 | INTEGER*4 |
| AMAX1 | REAL*4 | REAL*4 |
| DMAX1 | REAL*8 | REAL*8 |

**See Also:** AMAX0, MAX0, MAX1

### 17.3.54. MAX0 Function

**Usage:** selecting the largest value

**Syntax:** x = MAX0(number,number[,number...])

The MAX0 function returns the largest value from a list of integer numbers. All arguments that you specify must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMAX0 | INTEGER*2 | INTEGER*2 |
| JMAX0 | INTEGER*4 | INTEGER*4 |

**See Also:** AMAX0, MAX, MAX1

### 17.3.55. MAX1 Function

**Usage:** selecting the largest value

**Syntax:** x = MAX1(number,number[,number...])

The function MAX1 determines the largest value from a list of real numbers and converts the data type to integer. All arguments in the list must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMAX1 | REAL*4 | INTEGER*2 |
| JMAX1 | REAL*4 | INTEGER*4 |

**See Also:** AMAX0, MAX0, MAX, MIN

### 17.3.56. MIN Function

**Usage:** selecting the smallest value

**Syntax:** x = MIN(number,number[,number...])

The MIN function returns the smallest value from a list of integer or real numbers. All arguments that you specify must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMIN0 | INTEGER*2 | INTEGER*2 |
| JMIN0 | INTEGER*4 | INTEGER*4 |
| AMIN1 | REAL*4 | REAL*4 |
| DMIN1 | REAL*8 | REAL*8 |

See Also: MAX, MIN0, MIN1, AMIN0

### 17.3.57. MIN0 Function

**Usage:** selecting the smallest value

Syntax: x = MIN0(number,number[,number...])

The MIN0 function returns the smallest value from a list of integer numbers. All arguments that you specify must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMIN0 | INTEGER*2 | INTEGER*2 |
| JMIN0 | INTEGER*4 | INTEGER*4 |

**See Also:** MAX, MIN, MIN1, AMIN0

### 17.3.58. MIN1 Function

**Usage:** selecting the smallest value

**Syntax:** x = MIN1(number,number[,number...])

The MIN1 function returns the smallest value from a list of real numbers and converts the data type to integer. All arguments that you specify must have the same data type.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMIN1 | REAL*4 | INTEGER*2 |
| JMIN1 | REAL*4 | INTEGER*4 |

**See Also:** MAX, MIN, MIN0, AMIN0

### 17.3.59. MOD Function

**Usage:** remaindering

**Syntax:** x = MOD(number,number)

The MOD function returns the remainder from an integer or real number division. MOD divides the first number argument by the second and returns the remainder.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IMOD | INTEGER*2 | INTEGER*2 |
| JMOD | INTEGER*4 | INTEGER*4 |
| AMOD | REAL*4 | REAL*4 |
| DMOD | REAL*8 | REAL*8 |

### 17.3.60. NINT Function

**Usage:** nearest integer

**Syntax:** x = NINT(number)

The NINT function converts a real number to the nearest integer value. Note that the NINT function converts the data type to integer.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| ININT | REAL*4 | INTEGER*2 |
| JNINT | REAL*4 | INTEGER*4 |
| IIDNNT | REAL*8 | INTEGER*2 |
| JIDNNT | REAL*8 | INTEGER*4 |

**See Also:** AINT, ANINT, IDNINT, IDINT, IFIX, INT

### 17.3.61. NOT Function

**Usage:** bit complement operation

**Syntax:** iresult = NOT(int1)

The NOT function returns the bit complement of an integer argument. The following two lines of code are functionally equivalent.

```
iresult = .NOT.int1
iresult = NOT(int1)
```

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| INOT | INTEGER*2 | INTEGER*2 |
| JNOT | INTEGER*4 | INTEGER*4 |

**See Also:** AND, IEOR, IOR

### 17.3.62. REAL and DREAL Functions

**Usage:** numeric data type conversion

**Syntax:** x = REAL(number)

x = DREAL(number)

The REAL function converts an integer, real, or complex number to the REAL*4 data type. If the number you want to convert is already a REAL*4, the REAL function simply returns that number.

For a complex number, the REAL function simply ignores the imaginary portion and returns the real portion.

The DREAL function ignores the imaginary portion of a COMPLEX*16 number and returns the real portion as a REAL*8 number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| FLOATI | INTEGER*2 | REAL*4 |
| FLOATJ | INTEGER*4 | REAL*4 |
| - | REAL*4 | REAL*4 |
| SNGL | REAL*8 | REAL*4 |
| - | COMPLEX | REAL*4 |
| - | COMPLEX*16 | REAL*4 |
| DREAL | COMPLEX*16 | REAL*8 |

**See Also:** DBLE, DFLOAT, FLOAT

### 17.3.63. SIGN Function

**Usage:** transfer of sign

**Syntax:** x = SIGN(number,number)

The SIGN function transfers the sign of the second number argument to the first number argument. SIGN returns the absolute value of the first argument if the second argument is greater than or equal to 0, and the negative of the first argument if the second argument is less than 0.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IISIGN | INTEGER*2 | INTEGER*2 |
| JISIGN | INTEGER*4 | INTEGER*4 |
| SIGN | REAL*4 | REAL*4 |
| DSIGN | REAL*8 | REAL*8 |

**See Also:** ISIGN

### 17.3.64. SIN Function

**Usage:** trigonometric sine

**Syntax:** x = SIN(number)

The SIN function returns the trigonometric sine of a real or complex number in radians.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| SIN | REAL*4 | REAL*4 |
| DSIN | REAL*8 | REAL*8 |
| CSIN | COMPLEX | COMPLEX |
| CDSIN | COMPLEX*16 | COMPLEX*16 |

**See Also:** SIND, SINH

**F77 Compatibility Notes:**

The CDSIN function must be replaced by ZSIN when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.65. SIND Function

**Usage:** trigonometric sine

**Syntax:** x = SIND(number)

The SIND function returns the trigonometric sine of a real or complex number, expressed in degrees.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| SIND | REAL*4 | REAL*4 |
| DSIND | REAL*8 | REAL*8 |

**See Also:** SIN, SINH

### 17.3.66. SINH Function

**Usage:** trigonometric hyperbolic sine

**Syntax:** x = SINH(number)

The SINH function returns the trigonometric hyperbolic sine of a real number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| SINH | REAL*4 | REAL*4 |
| DSINH | REAL*8 | REAL*8 |

**See Also:** SIN, SIND

### 17.3.67. SQRT Function

**Usage:** square root

**Syntax:** x = SQRT(number)

The SQRT function returns the square root of a real or complex number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| SQRT | REAL*4 | REAL*4 |
| DSQRT | REAL*8 | REAL*8 |
| CSQRT | COMPLEX | COMPLEX |
| CDSQRT | COMPLEX*16 | COMPLEX*16 |

**F77 Compatibility Notes:**

The CDSQRT function must be replaced by ZSQRT when using F77 compatibility mode. Please refer to your User's Guide for details on compiler options and default modes.

### 17.3.68. TAN Function

**Usage:** trigonometric tangent

**Syntax:** x = TAN(number)

The TAN function returns the trigonometric tangent of a real number in radians.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| TAN | REAL*4 | REAL*4 |
| DTAN | REAL*8 | REAL*8 |

**See Also:** TAND, TANH

### 17.3.69. TAND Function

**Usage:** trigonometric tangent

**Syntax:** x = TAND(number)

The TAND function returns the trigonometric tangent of a real number, expressed in degrees.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| TAND | REAL*4 | REAL*4 |
| DTAND | REAL*8 | REAL*8 |

**See Also:** TAN, TANH

### 17.3.70. TANH Function

**Usage:** trigonometric hyperbolic tangent

**Syntax:** x = TANH(number)

The TANH function returns the trigonometric hyperbolic tangent of a real number.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| TANH | REAL*4 | REAL*4 |
| DTANH | REAL*8 | REAL*8 |

### 17.3.71. ZEXT Function

**Usage:** zero-extend

**Syntax:** x = ZEXT(arg)

Zero extend argument.

| Specific Names | Type of Argument | Type of Return Value |
|---|---|---|
| IZEXT | LOGICAL*1 | INTEGER*2 |
| | LOGICAL*2 | |
| | INTEGER*2 | |
| JZEXT | LOGICAL*1 | INTEGER*4 |
| | LOGICAL*2 | |
| | LOGICAL*4 | |
| | INTEGER*2 | |
| | INTEGER*4 | |

# Chapter 18
# Fortran Glossary

The FORTRAN-77 standard clearly defines all concepts and terminology specific to the language. This glossary presents the general Fortran terms listed alphabetically for easy reference. Most of the terms are explained in greater detail within the manual along with terms of more specialized meaning.

**array:** Sequence of data items collectively identified with one unique symbolic name and a data type.

**array element:** Individual data items that form an array. To reference a particular element in an array, specify the array name with a subscript. The subscript value is an integer expression that determines which element is referenced.

**array declarator:** Symbolic name and number of dimensions in an array. The number of dimensions determines the number and configuration of array elements.

**association:** Enables data to be identified by different symbolic names within the same program unit or in different program units within the same executable program. There are four forms of association: common, equivalence, argument, and entry.

**block data subprogram:** Nonexecutable program unit used to provide initial values for variables and array elements in named common blocks. A block data subprogram has a BLOCK DATA statement as its first statement.

**character storage unit:** Amount of storage required to hold one character of data. Fortran uses one byte of storage for one character of data. The storage unit establishes a means of referring to data storage without implying a specific storage technology.

**comment line:** Character sequence within the program code, used to provide program documentation. A comment line does not affect an executable program in any way. All comment lines start with the letter C or an asterisk.

**constant:** Program entity that has an unchanging value during the execution of a program. Constants can be arithmetic constants, logical constants, or character constants.

**continuation line:** Used to contain portions of a Fortran statement that exceed the 72 character columns available in the initial line for statement syntactic items. A statement can have up to nineteen continuation lines.

**defined:** Definition status of a program entity. A defined entity has a value that does not change until the entity becomes undefined or is redefined with a different value. An entity must be defined before it can be referenced.

**definition status:** Defined or undefined condition of a syntactic entity.

**dummy argument:** Symbolic name or an asterisk, *, used in the argument list of a procedure. Symbolic name dummy arguments hold a place for actual arguments passed to the procedure in the procedure reference. Symbolic name dummy arguments can be variables, arrays, array elements, functions, or subroutines, but must correspond to the number, type, and sequence of actual arguments in the procedure reference. An asterisk dummy argument indicates that the corresponding actual argument in a subroutine reference is an alternate return specifier for the subroutine.

**entity:** Generic term that refers to any language or program element, such as a program unit, a procedure, a variable, or an array. The term syntactic entity or syntactic item refers to individual elements that make up statements and expressions, such as statement labels, keywords, symbolic names, constants, operators, and special characters.

**executable program:** Program units that consists of a main program and any number, including zero, of subprograms and external procedures. An executable program cannot have more than one main program.

**executable statement:** Any statement that specifies some processing action, such as a GOTO or RETURN statement.

**external procedure:** Fortran subroutine or external function specified outside of the program unit that calls or references it. External procedures can be written in another language for use in a Fortran program.

**function subprogram:** Executable procedure that can be referenced in an expression. A function subprogram returns a value to the expression that references it. There are three categories of functions: intrinsic, statement, and external.

**initial line:** First line of a Fortran statement. If the statement exceeds the initial line, you can use up to nineteen continuation lines.

**initially defined:** Definition status of an entity. An entity is initially defined if it is assigned a value in a DATA statement.

**keyword:** Sequence of letters that has a special purpose in Fortran. Keywords identify Fortran statements, intrinsic functions, or statement separators. Examples are DIMENSION, CONTINUE, ABS, SQRT, THEN, and TO.

**list:** Nonempty sequence of syntactic entities separated by commas. The entities in the list are called list items.

**main program:** Program unit that receives control from the operating system to begin execution of a Fortran program. Main programs can execute isolated from any other program unit or can call and reference subprograms during execution. However, you cannot reference the main program from a subprogram. There can be only one main program in an executable Fortran program.

**nonexecutable statement:** Statements that classify and define program units, specify entry points in subprograms, specify editing information, and specify initial values and execution characteristics for data.

**numeric storage unit:** Amount of storage required to hold an integer, real, or logical numeric value. A double precision or complex numeric value uses two numeric storage units in a storage sequence. The storage unit establishes a means of referring to data storage without implying a specific storage technology.

**procedure, procedure subprogram:** Subroutine and function subprograms. There are three categories of function subprograms: intrinsic functions, statement functions, and external functions. The term external procedure refers to subroutines and external functions only. You can write external procedures in another programming language for use in a Fortran program.

**program unit:** Sequence of Fortran statements and optional comment lines. A program unit is either a main program or subprogram.

**reference:** Applies to syntactic entities and function procedures. To reference a syntactic entity means to use the name of an entity in a statement that requires the value of that particular entity for execution within the program context. To reference a function procedure means to use the name of a function in an expression or statement that requires that particular function operation for execution within the program context.

**scope:** Extent to which a given symbolic name or statement label can affect a program. For example, a statement label has the scope of a program unit. A statement label in one program unit does not affect any other program unit.

**sequence:** Set of elements ordered by a one-to-one correspondence with the numbers 1, 2, 3, through n. The number of elements in a sequence is the number n. An empty sequence contains no elements.

**statement:** Sequence of syntactic items. Except for assignment and statement function statements, all statements begin with a keyword. Statements are written in one or more lines.

**statement label:** Sequence of one to five digits. One of the digits must be nonzero. Statement labels are used to identify specific statements in a program.

**subprogram:** Program unit that is called or referenced from either the main program or another subprogram. There are two classes of subprograms: block data and procedures.

**subprogram, suprogram:** External procedure. An external procedure is specified outside of the program unit that calls it. A subroutine is referenced with the CALL statement.

**substring:** Contiguous sequence of characters that represents a portion of a character datum. A character datum is a string of one or more characters. Substrings are identified with a substring name. The substring name is used to define and reference the substring.

**symbolic name:** A sequence of alphanumeric characters. The first character in a symbolic name must be a letter. Symbolic names can identify constants, variables, arrays, main programs, subprograms, common blocks, and dummy procedures. Character sequences that serve within the program context as format edit descriptors and keywords are not considered symbolic names.

**syntactic item:** Generic terms that refer to individual elements that make up statements and expressions, such as statement labels, keywords, symbolic names, constants, operators, and special characters. See entity for more information.

**variable:** Entity that can assume a changing value during program execution through implicit or explicit redefinition. A variable has both a name and a type.

# Chapter 19
# Language Extensions and Features

The following sections list the VAX/VMS extensions available in Green Hills Fortran, along with a list of unimplemented extensions.

## 19.1. Implemented VAX/VMS Extensions

The following list summarizes the VAX/VMS Fortran Language Extensions implemented in Green Hills Fortran. Some of these features are only available in VMS compatibility mode (normally the default). Please refer to your Green Hills Fortran User's Guide for details on compile time options and default modes.

### 19.1.1. Line Formatting Extensions

Comments may be indicated by 'C', '!', or '*' in column 1
Comments may be indicated by '!' in the statement field
D in column 1 is a debug statement indicator
Continuation convention extension: 'tab' character followed by any of the characters '1' ... '9'
Allow up to 99 continuation lines
INCLUDE statement syntax is INCLUDE "pathname"
    Include files in text libraries are not supported
    The /LIST and /NOLIST options are not supported

### 19.1.2. Lexical Extensions

'$', '_' in identifiers
Identifiers up to 31 characters long
'nnn'O and 'nnn'X octal and hex integer constants
Radix 50 constants
Hollerith typeless constants

### 19.1.3. Declaration Extensions

BYTE data type
DATA statement in any place in program unit
IMPLICIT NONE statement
VOLATILE statement
VIRTUAL statement
Extended PARAMETER statement with additional operators and functions
"*n" size qualifier on function names and identifier declarations
Declare multifield records with STRUCTURE statement (no initialization allowed)
Single subscript in EQUIVALENCE of multidimensional array

### 19.1.4. Initialization Extensions

Initialization in type declaration statements
Initialize CHARACTER variables with integer values
Initialize static and COMMON storage to 0 if not specified (Unix only)

### 19.1.5. Expression Extensions

%VAL(), %REF(), %LOC(), accept and ignore %DESC
Logical type allowed in integer context in expressions
Logical operators apply (bitwise) to integers
Use non-integer type expression in integer context: convert to integer

### 19.1.6. Builtin Subroutine and Function Extensions

Degree-style trig functions
System subroutines: DATE,IDATE,ERRSNS,EXIT,SECNDS,TIME,RAN,MVBITS

### 19.1.7. Statement Extensions

DO WHILE statement
END DO statement
Extended range DO loops
Ampersand (&) or asterisk (*) for alternate return
OPTIONS statement with full VAX syntax. Ignore all options except /I4, /NOI4, /D_LINES

### 19.1.8. Input/Output Extensions

NAMELIST
ACCEPT statement
TYPE statement
ENCODE/DECODE statements
Accept full VAX/VMS syntax for OPEN, INQUIRE and CLOSE statements, but ignore most options
Allow 99 files open at the same time
IBM ('n) form of relative record specification

### 19.1.9. Format Extensions

O (octal) and Z (hex) Edit descriptors, including the form 'On.n'
H edit descriptor on input
Q edit descriptor
$ edit descriptor
Default field widths for IOZLFEDGA edit descriptors match data type
'$' and '\0' carriage control — mapped to Unix carriage control
Short field terminators to separate numeric data on input

## 19.2. Compiler Complexity Equal To Or Better Than VAX/VMS Fortran

| | |
|---|---|
| 20 | DO and IF statement nesting |
| 255 | Arguments in a CALL or function reference |
| 250 | Named COMMON blocks |
| 8 | Format group nesting |
| 500 | Labels in computed GOTO |
| 40 | Parentheses in nested expressions |
| 10 | Include file nesting |
| 99 | Continuation lines (with compiler option) |
| 132 | Source line length in characters |
| 31 | Identifier length |
| 2000 | Constant length, character and Hollerith |
| 12 | Constant length, radix-50 |
| 7 | Array dimensions |
| 250 | Number of names in a NAMELIST group |

## 19.3. Unimplemented VAX/VMS Fortran Extensions

REAL*16, COMPLEX*32
Indexed Files
Expressions in Format Statements (F<i+j>.<k-l>)
The DELETE statement for relative files
The REWRITE statement for relative files
Initialization of STRUCTUREs
The DEFINE FILE statement
The FIND statement

# Chapter 20
# DOD MIL-STD-1753 Syntax and Semantic

The following section summarizes the syntax and semantics required by the Department of Defense, as defined in MIL-STD-1753 as additions to ANSI X3.9-1978. The information below is based on the Department of Defense document MIL-STD-1753 9 November 1978, MILITARY STANDARD FORTRAN, DOD Supplement to American National Standard X3.9-1978.

This information is provided for programmer convenience only and is not intended as a substitute for the official DOD document. Unless otherwise noted, the Green Hills Fortran compiler operation is compatible with MIL-STD-1753.

The following subroutines and functions are only available when either VMS compatibility mode or DOD MIL compatibility mode has been selected. These routines are not available under F77 compatibility mode unless the DOD MIL compatibility mode switch is also selected. Please refer to your system-specific User's Guide for details on compile time options and default modes.

| | | | |
|---|---|---|---|
| MVBITS | ISHFT | IBITS | IBSET |
| IBCLR | ISHFTC | BTEST | |

## 20.1. END DO

The END DO statement must only be used as the terminal statement of a DO-loop and has no other effect. If END DO is used as the terminal statement of an ANSI X3.9-1978 DO statement, the END DO statement must be labeled.

## 20.2. DO WHILE

**Syntax:** DO [ label [,]] WHILE logical-expression
...
[label] END DO

The DO WHILE statement allows execution of a DO-loop while a logical expression is true. The logical expression is evaluated and tested at the beginning of the DO-loop. Each DO WHILE DO-loop must be terminated by a separate END DO statement. The label is optional. If the label is used in a DO WHILE statement, the END DO statement that terminates the DO-loop must be labeled with the same label. The rules for transfers into the range of the DO-loop are the same as for the current standard DO-loop.

## 20.3. INCLUDE

The INCLUDE statement provides the capability to copy source code from a file into a compiler source stream. The INCLUDE statement must be contained on one line.

The filename identifies the source code to be copied into the source stream. The filename can be processor dependent.

An INCLUDE statement will initiate copying at the beginning of the file. The file must not be empty and the first line that is not a comment line must not be a continuation line.

Implementation of the INCLUDE statement must permit the concept of nonrecursive nesting (i.e., the included copy may contain another INCLUDE statement).

## 20.4. IMPLICIT

The IMPLICIT statement is extended to provide the capability to the programmer to void all default implicit types except for the intrinsic functions. An additional form of the IMPLICIT statement is IMPLICIT NONE.

If the IMPLICIT NONE appears in a program unit, no other IMPLICIT statements may appear in the same program unit.

## 20.5. READ and WRITE Past END-OF-FILE

The processor must provide a facility that permits reading and writing to continue past an endfile record on an unlabeled magnetic tape sequential file. Reading past an endfile record is not permitted if the READ statement does not contain an END= or an IOSTAT= specifier. The processor may require execution of a special subroutine or statement before it permits such reading or writing.

## 20.6. Bit Field Manipulations

Bit manipulation capability is provided through a standard set of external functions. This capability is compatible with similar functions included in ANSI/ISA S61.1-1976. In each of the defined functions, it is assumed that the integer arguments m and n are represented in binary form.

### 20.6.1. Binary Pattern Processing

#### 20.6.1.1. Logical Operations

Logical operations provided are the Boolean functions OR, AND, EOR and NOT. These operations are provided as integer external functions. The implicit type for OR, AND and EOR is indicated by the use of I as the first letter of the function name. Their arguments, m and n, can be integer constants, integer variables, integer array elements, or integer expressions. After execution of the functions, the arguments remain unchanged. The operations are performed on all corresponding bits of the two operands.

#### 20.6.1.2. Inclusive OR

Function reference: IOR (m,n)

The arguments, m and n, are combined according to the following truth table:

```
  m   =  0 1 0 1
  n   =  0 0 1 1
-------------------
Value =  0 1 1 1
```

### 20.6.1.3. Logical AND

Function reference: IAND (m,n)

The arguments, m and n, are combined according to the following truth table:

| | | |
|---|---|---|
| m | = | 0 1 0 1 |
| n | = | 0 0 1 1 |
| Value | = | 0 0 0 1 |

### 20.6.1.4. Logical Complement

Function reference: NOT (m)

The argument m is logically complemented according to the following truth table:

| | | |
|---|---|---|
| m | = | 0 1 |
| Value | = | 1 0 |

### 20.6.1.5. Exclusive OR

Function reference: IEOR (m,n)

The arguments, m and n, are combined according to the following truth table:

| | | |
|---|---|---|
| m | = | 0 1 0 1 |
| n | = | 0 0 1 1 |
| Value | = | 0 1 1 0 |

## 20.6.2. Shift Operations

The shift operations provided are logical and circular. The shift operations are implemented as integer functions. The arguments may be integer constants, integer variables, integer array elements, or integer expressions. The arguments m and k are as follows:

m specifies the value (binary pattern) to be shifted
k specifies the shift count
- k > 0 indicates a left shift
- k = 0 indicates no shift
- k < 0 indicates a right shift

If the absolute value of the shift count is greater than the number of bits in a numeric storage unit, the result is undefined. The arguments are not changed by the shift operations.

#### 20.6.2.1. Logical Shift

Function reference: ISHFT (m,k)

All bits representing the argument m are shifted k places. Bits shifted out from the left end or the right end, as the case may be, are lost. Zeros are shifted in from the opposite end.

#### 20.6.2.2. Circular Shift

Function reference: ISHFTC (m,k,ic)

The rightmost ic bits of the argument m are shifted circularly k places; i.e., the bits shifted out of one end are shifted into the opposite end. No bits are lost. The unshifted bits of the result are the same as the unshifted bits of the argument m. The absolute value of the argument k must be less than or equal to ic. The argument ic must be greater than or equal to one and less than or equal to the number of bits in a numeric storage unit.

### 20.6.3 Bit Subfields

Bit subfields are referenced by specifying a bit position and a length. Bit positions within a numeric storage unit are numbered from right to left and the rightmost bit position is numbered 0. Bit fields may not extend from one numeric storage unit into another numeric storage unit, and the length of a field must be greater than zero.

#### 20.6.3.1. Bit Extraction

Function reference: IBITS (m,i,len)

where m, i, and len are integer expressions.

This function extracts a subfield of len bits from m starting with bit position i and extending left for len bits. The result field is right justified and the remaining bits are set to zero. The value of i+len must be less than or equal to the number of bits in a numeric storage unit.

#### 20.6.3.2. Bit Move Subroutine

CALL MVBITS (m,i,len,n,j)

This subroutine moves len bits from positions i through i+len-1 of argument m to positions j through j+len-1 of argument n. The portion of argument n not affected by the movement of bits remains unchanged. All arguments are integer expressions except n must be a variable or array element. Arguments m and n are permitted to be the same numeric storage unit. The values of i+len and j+len must be less than or equal to the number of bits in a numeric storage unit.

### 20.6.4. Bit Processing

Individual bits of a numeric storage unit can be tested and changed with the following routines for bit processing. The functions have two arguments n and i which are integer expressions.

n specifies the binary pattern
i specifies the bit position (rightmost bit is bit 0)

If i is negative or greater than the number of bits in a numeric storage unit, the result of the function is undefined.

#### 20.6.4.1. Bit Testing

Function reference: BTEST (n,i)

This function is a logical function. The ith bit of argument n is tested. If it is 1, the value of the function is .TRUE.; if it is 0, the value of the function is .FALSE.

#### 20.6.4.2. Set Bit

Function reference: IBSET (n,i)

The result of the IBSET function is equal to the value of n with the ith bit set to 1.

#### 20.6.4.3. Clear Bit

Function reference: IBCLR (n,i)

The result of the IBCLR function is equal to the value of n with the ith bit set to 0.

### 20.7. Bit Constants

The following two forms of bit constants are permitted in DATA statements:

O'di ... dn'
Z'hi ... hn'

where di are octal digits and hi are hexadecimal digits with A-F representing the decimal equivalent of 10-15. These constants are right-justified and may be associated only with integer entities. These constants may appear only in DATA statements.

# Chapter 21
# ASCII and Hexadecimal Conversion Table

| DECIMAL | ASCII | HEX | DECIMAL | ASCII | HEX | DECIMAL | ASCII | HEX | DECIMAL | ASCII | HEX |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | 00 | 32 | SP | 20 | 64 | @ | 40 | 96 | ‘ | 60 |
| 1 | SOH | 01 | 33 | ! | 21 | 65 | A | 41 | 97 | a | 61 |
| 2 | STX | 02 | 34 | “ | 22 | 66 | B | 42 | 98 | b | 62 |
| 3 | ETX | 03 | 35 | # | 23 | 67 | C | 43 | 99 | c | 63 |
| 4 | EOT | 04 | 36 | $ | 24 | 68 | D | 44 | 100 | d | 64 |
| 5 | ENQ | 05 | 37 | % | 25 | 69 | E | 45 | 101 | e | 65 |
| 6 | ACK | 06 | 38 | & | 26 | 70 | F | 46 | 102 | f | 66 |
| 7 | BEL | 07 | 39 | ’ | 27 | 71 | G | 47 | 103 | g | 67 |
| 8 | BS | 08 | 40 | ( | 28 | 72 | H | 48 | 104 | h | 68 |
| 9 | HT | 09 | 41 | ) | 29 | 73 | I | 49 | 105 | i | 69 |
| 10 | LF | 0A | 42 | * | 2A | 74 | J | 4A | 106 | j | 6A |
| 11 | VT | 0B | 43 | + | 2B | 75 | K | 4B | 107 | k | 6B |
| 12 | FF | 0C | 44 | , | 2C | 76 | L | 4C | 108 | l | 6C |
| 13 | CR | 0D | 45 | – | 2D | 77 | M | 4D | 109 | m | 6D |
| 14 | SO | 0E | 46 | . | 2E | 78 | N | 4E | 110 | n | 6E |
| 15 | SI | 0F | 47 | / | 2F | 79 | O | 4F | 111 | o | 6F |
| 16 | DLE | 10 | 48 | 0 | 30 | 80 | P | 50 | 112 | p | 70 |
| 17 | DC1 | 11 | 49 | 1 | 31 | 81 | Q | 51 | 113 | q | 71 |
| 18 | DC2 | 12 | 50 | 2 | 32 | 82 | R | 52 | 114 | r | 72 |
| 19 | DC3 | 13 | 51 | 3 | 33 | 83 | S | 53 | 115 | s | 73 |
| 20 | DC4 | 14 | 52 | 4 | 34 | 84 | T | 54 | 116 | t | 74 |
| 21 | NAK | 15 | 53 | 5 | 35 | 85 | U | 55 | 117 | u | 75 |
| 22 | SYN | 16 | 54 | 6 | 36 | 86 | V | 56 | 118 | v | 76 |
| 23 | ETB | 17 | 55 | 7 | 37 | 87 | W | 57 | 119 | w | 77 |
| 24 | CAN | 18 | 56 | 8 | 38 | 88 | X | 58 | 120 | x | 78 |
| 25 | EM | 19 | 57 | 9 | 39 | 89 | Y | 59 | 121 | y | 79 |
| 26 | SUB | 1A | 58 | : | 3A | 90 | Z | 5A | 122 | z | 7A |
| 27 | ESC | 1B | 59 | ; | 3B | 91 | [ | 5B | 123 | { | 7B |
| 28 | FS | 1C | 60 | < | 3C | 92 | \ | 5C | 124 | \| | 7C |
| 29 | GS | 1D | 61 | = | 3D | 93 | ] | 5D | 125 | } | 7D |
| 30 | RS | 1E | 62 | > | 3E | 94 | ^ | 5E | 126 | ~ | 7E |
| 31 | US | 1F | 63 | ? | 3F | 95 | _ | 5F | 127 | DEL | 7F |