May 1995
Order Number: 312644-002

Paragon™ System

Fortran Language Reference Manual

Intel® Corporation

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

This manual describes the implementation of FORTRAN 77, the language accepted by the *if77* compiler, and is part of a set of manuals describing the Fortran and C compilers and the compilation tools available from Intel Scalable Systems Division. This manual presents a description of the statements and intrinsics accepted by *if77* FORTRAN 77. The Fortran compilation system consists of an ANSI-conformant Fortran compiler, macro-processor, assembler, linker, utilities, a debugger and a profiler. You can use these tools to create, debug, optimize and profile your software. Refer to the section "Related Publications" for a list of the other manuals in the manual set.

Audience Description

This manual is intended for people who are writing programs in Fortran and are familiar with the language. To use *if77*, you should be aware of the role of Fortran and of assembly-language programs in the software development process. The *if77* compiler runs on a variety of host systems. To use *if77*, you need to be familiar with the basic commands available on your host system.

Compatibility and Conformance to Standards

The *if77* compiler accepts an enhanced version of FORTRAN 77 and runs on a variety of host systems. This version of FORTRAN 77 conforms to the ANSI standard for FORTRAN 77 and includes various extensions from VAX/VMS Fortran, IBM/VS Fortran, and MIL-STD-1753.

For further information, you can also refer to the following:

- *American National Standard Programming Language Fortran, ANSI X3.-1978* (1978).
- *Programming in VAX Fortran, Version 4.0,* Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran,* IBM Corporation, Rev. GC26-4119.
- *Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI X3.-1978*, MIL-STD-1753 (November 9, 1978).

Organization

This manual is divided into the following chapters and appendices:

Chapter 1	"Language Overview" provides a description of the language structures and the overall language features.
Chapter 2	"Data Types" describes the Fortran data types and constants that *if77* supports.
Chapter 3	"Fortran Statements" provides an alphabetical listing of each statement, with a summary of each statement, a syntax description, and a complete description.
Chapter 4	"Input and Output" describes the types of input and output available with *if77* Fortran.
Appendix A	"Intrinsics" specifies the *if77* intrinsic functions.
Appendix B	"VAX System Subroutines and Built-in Functions" discusses the VAX/VMS system subroutines and the built-in functions.

Hardware and Software Constraints

This manual describes a version of Fortran that is accepted by *if77*, operates on a variety of host systems and produces object code for the i860™ XR and the i860™ XP microprocessors. Details concerning environment-specific values and defaults and host-specific features or limitations are presented in the release notes and installation instructions sent with the *if77* software.

Conventions

This manual uses the following conventions:

italic	is used for commands, filenames, directories, arguments, options and for emphasis.
`Constant Width`	is used in examples and for language statements in the text.
[*item1*]	square brackets indicate optional items. In this case *item1* is optional.
{ *item2* | *item3* }	braces indicate that a selection is required. In this case, you must select either *item2* or *item3*.
filename ...	ellipsis indicates a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.
FORTRAN	Fortran language statements are shown using upper-case characters.
<TAB>	non-printing characters, such as TAB, are shown enclosed in greater than and less than characters.
§	this symbol indicates an area in the text that describes a FORTRAN 77 enhancement. Enhancements may be VAX/VMS FORTRAN enhancements, IBM/VS enhancements or military standard MIL-STD-1753 enhancements.

Related Publications

The following documents contain additional information related to the *if77* compiler.

- *Paragon™ System Fortran Compiler User's Guide*
- *Paragon™ System i860™ 64-Bit Microprocessor Assembler Reference Manual*
- *System V Application Binary Interface i860 Intel i860™ Processor Supplement* by AT&T Unix System Laboratories, Inc (available from Prentice Hall, Inc.).
- *American National Standard Programming Language Fortran*, ANSI x.3-1978 (1978).
- *Programming in VAX Fortran, Version 4.0,* Digital Equipment Corporation (September, 1984).
- *IBM VS Fortran,* IBM Corporation, Rev. GC26-4119.
- *Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI X3.-1978*, MIL-STD-1753 (November 9, 1978).

Comments and Assistance

Intel Scalable Systems Division is eager to hear of your experiences with our new software product. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
Phone: 800-421-2823
Internet: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Intel Japan K.K.
Scalable Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 677-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 677-9147

Table of Contents

Chapter 1
Language Overview

Chapter 2
Data Types

Chapter 3
Fortran Statements

Chapter 4
Input and Output

Appendix A
Intrinsics

Appendix B
VAX Built-in Functions and System Subroutines

List of Illustrations

List of Tables

Language Overview 1

This chapter describes the basic elements of the Fortran language, the format of Fortran records and the types of expressions and assignments accepted by *if77* Fortran.

Elements of a Fortran Module

A Fortran module is either a SUBROUTINE, FUNCTION, BLOCK DATA or PROGRAM.

Fortran source consists of a sequence of modules which are to be compiled into object modules. Every module consists of statements and optionally comments beginning with the module statement, either a SUBROUTINE, FUNCTION, BLOCK DATA or PROGRAM statement, and finishing with an END statement.

In the absence of a module statement, the compiler will insert a PROGRAM statement.

Statements and Comments

Statements are either executable statements or specification statements. Each statement consists of a single line or source record, possibly followed by one or more continuation lines. Comments may appear anywhere in the source.

§ To append a comment to a Fortran statement line, precede the comment with an exclamation mark (!) followed by the comment on the same line.

Debug Statements

The letter "D" in column 1 designates the statement on that line to be a debugging statement. The compiler will treat the debugging statement as a comment unless the command line option *–Mdlines* is set during the compilation. In that case, the compiler acts as if the "D" were a blank and compiles the line according to the standard rules.

Statement Ordering

The rules defining the order in which statements appear in a program unit have been relaxed, as compared to the ANSI standard, as follows:

- DATA statements can be freely interspersed with PARAMETER statements and other specification statements.
- NAMELIST statements are supported and have the same order requirements as FORMAT and ENTRY statements.
- The IMPLICIT NONE statement can precede other IMPLICIT statements.

Figure 1-1 shows the required order of statements in a Fortran subprogram. In Figure 1-1, read from top to bottom and left to right. For example, since the column for comments spans the entire table, up to the END row, this indicates that comments may occur anywhere within a Fortran subprogram, before an END statement.

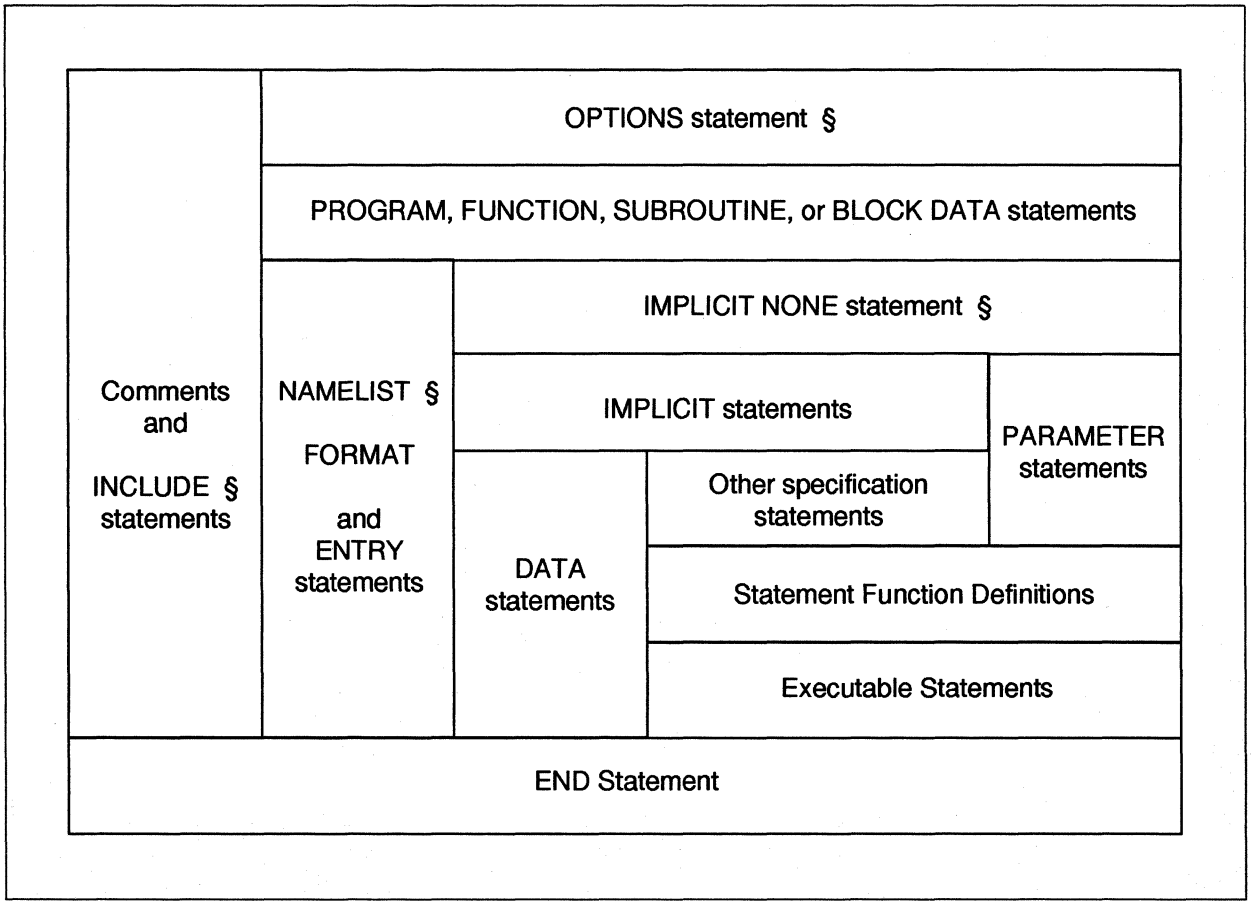

Figure 1-1. Order of Statements

The Fortran Character Set

Table 1-1 shows the Fortran character set. Character variables and constants can use any ASCII character.

Table 1-1. Fortran Characters

Character	Description
A-Z, a-z	alphabetic
0-9	numeric
<space>	space character
=	equals
+	plus
-	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
_	underscore character
.	decimal point
!	exclamation mark
<TAB>	tabulation character
<CR>	carriage return

Table 1-2 shows C language character escape sequences that are recognized in *if77* Fortran character string constants. The *if77* option *–Mbackslash* enables and disables this enhancement.

§ **Table 1-2. C Compatibility Characters**

Character	Description
\v	vertical tab
\a	alert (bell)
\n	newline
\t	tab
\b	backspace
\f	formfeed
\r	carriage return
\0	null
\'	apostrophe (does not terminate a string)
\"	double quotes (does not terminate a string)
\\	\
x	*x*, where *x* is any other character
ddd	character with the given octal representation.

Formatting

A Fortran record may be formatted with tabs or by column formatting.

Column Formatting

A Fortran record consists of a sequence of up to 73 ASCII characters, the last being <CR>. It has a fixed layout as shown in Table 1-3.

Table 1-3. Record Positions and Fields

Position	Field
1-5	Label field
6	Continuation field
7-72	Statement field

Characters beyond position 72 on a line are ignored. Extended lines containing up to 132 characters are valid if you use the *if77 –Mextend* option. For information on this option, refer to the *Paragon™ System Fortran Compiler User's Guide*.

§ Tab Formatting

A tab formatted record consists of up to 72 ASCII characters. It is made up of a label field, an optional continuation indicator and a statement field. The label field is terminated by a tab character. The label cannot be more than 5 characters.

A continuation line is indicated by a tab character followed immediately by a digit. The statement field starts after a continuation indicator, when one is present. The 73rd and subsequent characters are ignored. Extended lines containing up to 132 characters are valid if you use the *if77 –Mextend* option. For information on this option, refer to the *Paragon™ System Fortran Compiler User's Guide*.

Label Field

The label field holds up to five characters. The characters C or * in the first character position of a label field indicate a comment line.

§ In addition, to C or *, either of the characters D or ! in the first character position of a label field also indicate a comment line.

When a numeric field drawn from digits 0 to 9 is placed in the label field, the field is a label. A line with no label, and with five space characters or a <TAB> (the tab is an extension §) in the label field, is an unlabeled statement. Each label must be unique in its module. Continuation lines must not be labeled. Labels can only be jumped to when they are on executable statements.

Continuation Field

The sixth character position, or the position after the tab, is the continuation field. This field is ignored in comment lines. It is invalid if the label field is not five spaces. A value of 0, <space> or <TAB> indicates the first line of a statement. Any other value indicates a subsequent (continuation) line to the preceding statement.

Statement Field

This consists of valid identifiers and symbols, possibly separated by <space> or <TAB> and terminated by <CR>.

§ Within the statement field tabs and spaces are ignored as are characters following a ! or beyond the 72nd character. Extended lines containing up to 132 characters are valid if you use the *if77 –Mextend* option. For information on this option, refer to the *Paragon™ System Fortran Compiler User's Guide*.

Including Fortran Source Files

The sequence of consecutive compilation of source statements may be interrupted so that an extra source file can be included. This is carried out using the INCLUDE statement which takes the following form:

```
INCLUDE "filename"
```

where *filename* is the name of the file to be included. Pairs of either single or double quotes are acceptable enclosing *filename*.

The INCLUDE file is compiled to replace the INCLUDE statement, and on completion of that source the file is closed and compilation continues with the statement following the INCLUDE.

INCLUDE files are especially recommended when the same COMMON blocks and the same COMMON block data mappings are used in several modules.

For example the following statement includes the file `MYFILE.CMN`.

```
INCLUDE "MYFILE.CMN"
```

§ Input File Format – Summary of Extensions

Input source file format has been extended from FORTRAN 77 to allow the following extensions:

- A continuation line may also be indicated by using an ampersand (&) in column one of a line.
- Tab-Format lines are supported. A tab in columns 1-6 ends the statement label field and begins an optional continuation indicator field. If a non-zero digit follows the tab character, the continuation field exists and indicates a continuation field. If anything other than a non-zero digit follows the tab character, the statement body begins with that character and extends to the end of the source statement. Note that this does not override FORTRAN 77's source line handling since no valid Fortran statement can begin with a non-zero digit. The tab character is ignored if it occurs in a line except in Hollerith or character constants.
- Input lines may be of varying lengths. If there are fewer than 72 characters, the line is padded with blanks; characters after the 72nd are ignored unless you use the *–Mextend* option on the command line.
- If the *–Mextend* option is used on the command line then the input line can extend to 132 characters. The line is padded with blanks if it is fewer than 132; characters after the 132nd are ignored. Note that use of this option extends the statement field to position 132.
- Blank lines are allowed at the end of a program unit.
- The number of continuation lines allowed is extended to 99.

The Components of Fortran Statements

Fortran modules are made up of statements which consist of expressions and elements. An expression can be broken down to simpler expressions and eventually to its elements combined with operators. Hence the basic building block of a statement is an element. An element takes one of the following forms:

- A *constant* represents a fixed value.
- A *variable* represents a value which may change during program execution.
- An *array* is a group of values, stored contiguously, that can be referred to as a whole or separately. The separate values are known as the elements of the array. The array has a symbolic name.
- A *function reference* is the name of a function followed by an argument list. The reference causes the code specified at function definition to be executed and the result substituted for the function reference.

Symbolic Names

Symbolic names identify different entities in Fortran source. A symbolic name is a string of letters and digits, which must start with a letter and is terminated by a character not in the symbolic names set (for example a <space> or a <TAB> character). Underscore (_) characters may appear within symbolic names. Symbolic names may start with a dollar sign ($) or underscore (_) character (this is a *if77* extension). Only the first thirty-one characters identify the symbol. Below are several symbolic names:

```
NUM
CRA9
Y
numericabcdefghijklmnopqrstuvwxyz
```

The last example is identified by its first 31 characters and is equivalent to:

```
numericabcdefghijklmnopqrstuvwx
```

The following are examples are *invalid* symbolic names.

```
8Q
```

This is invalid because it begins with a number.

```
FIVE.4
```

This is invalid because it contains a period which is an invalid character.

Symbolic Name Scope

Symbolic names may be declared locally or globally.

Names of COMMON blocks, SUBROUTINEs and FUNCTIONs are global to those modules that reference them. They must refer to unique objects, not only during compilation, but also in the link stage.

The scope of names other than these is local to the module in which they occur, and any reference to the name in a different module will imply a new local declaration. This includes the arithmetic function statement.

Expressions

Each data item, such as a variable or a constant, represents a particular value at any point during program execution. These elements may be combined together to form expressions, using binary or unary operators, so that the expression itself yields a value.

An expression is formed as:

expr binary-operator expr

or

unary-operator expr

where an *expr* is formed as

expression or *element*

For example,

```
A+B
-C
+D
```

These are simple expressions whose components are elements. Expressions fall into one of four classes: arithmetic, relational, logical or character.

Arithmetic Expressions

Arithmetic expressions are formed from arithmetic elements and arithmetic operators. An arithmetic element may be:

- an arithmetic expression
- a variable
- a constant
- an array element
- a function reference
- § a field of a structure or union

The arithmetic operators specify a computation to be performed on the elements. The result is a numeric result. Table 1-4 shows the arithmetic operators.

Table 1-4. Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus
–	Subtraction or unary minus

Note that a value should be associated with a variable or array element before it is used in an expression. Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. Table 1-5 shows the precedence of each arithmetic operator.

Table 1-5. Operator Precedence

Operator	Precedence
**	First
* and /	Second
+ and –	Third

This following example is resolved into the arithmetic expressions `(A) + (B * C)` rather than `(A + B) * (C)`.

```
A + B * C
```

Normal ranked precedence may be overcome using parentheses which force the item(s) enclosed to be evaluated first.

```
(A + B) * C
```

The compiler resolves this into the expressions `(A + B) * (C)`.

The type of an arithmetic expression is:

INTEGER if it contains only integer elements.

REAL if it contains only real and integer elements.

DOUBLE PRECISION
if it contains only double precision, real and integer elements.

COMPLEX if any element is complex. Any element which needs conversion to complex will be converted by taking the real part from the original value and setting the imaginary part to zero.

§ DOUBLE COMPLEX
if any element is double complex.

Relational Expressions

A relational expression is composed of two arithmetic expressions separated by a relational operator. The value of the expression is true or false (`.TRUE.` or `.FALSE.`) depending on the value of the expressions and the nature of the operator.

Table 1-6 shows the relational operators.

Table 1-6. Relational Operators

Operator	Relationship
`.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 relational expressions the arithmetic elements are evaluated to obtain their values. The relationship is then evaluated to obtain the true or false result. Thus the relational expression:

```
TIME + MEAN .LT. LAST
```

means if the sum of TIME and MEAN is less than the value of LAST, then the result is true, otherwise it is false.

Logical Expressions

A logical expression is composed of two relational or logical expressions separated by a logical operator. Each logical expression yields the value true or false (`.TRUE.` or `.FALSE.`).

Table 1-7 shows the logical operators.

Table 1-7. Logical Operators

Operator	Relationship
.AND.	True if both expressions are true.
.OR.	True if either expression or both is true.
.NOT.	This is a unary operator; it is true if the expression that follows is false, otherwise it is false.
.NEQV.	False if both expressions have the same logical value
.XOR.	Same as .NEQV.
.EQV.	True if both expressions have the same logical value

In the following example, `TEST` will be `.TRUE.` if `A` is greater than `B` or `I` is not equal to `J+17`.

```
TEST = A .GT. B .OR. I .NE. J+17
```

Character Expressions

An expression of type CHARACTER can consist of one or more printable characters. Its length is the number of characters in the string. Each character is numbered consecutively from left to right beginning with 1. For example:

```
'ab_&'
'A@HJi2'
'var[1,12]'
```

Character Concatenation

A character expression can be formed by concatenating two (or more) valid character expressions using the concatenation operator //.

Table 1-8 shows several examples of concatenation

Table 1-8. Character Concatenation

Expression	Value
'ABC'//'YZ'	"ABCYZ"
'JOHN '//'SMITH'	"JOHN SMITH"
'J '//'JAMES '//'JOY'	"J JAMES JOY"

Precedence Rules

Arithmetic, relational and logical expressions may be identified to the compiler by the use of parentheses, as described in the section on arithmetic expressions. When no guidance is given to the compiler it will impose a set of precedence rules to identify each expression uniquely. Table 1-9 shows the operator precedence rules for expressions.

Table 1-9. Operator Precedence

Operator	Evaluated
**	First
* and /	Second
+ and –	Third
Relational operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.NEQV. and .EQV.	Eighth

Operators of equal rank are evaluated left to right. Thus:

```
A*B+B**C .EQ. X+Y/Z .AND. .NOT. K-3.0 .GT. T
```

is equivalent to:

```
((((A*B)+(B**C)) .EQ. (X+(Y/Z))) .AND. (.NOT. ((K-3.0) .GT. T)))
```

Assignment Statements

A Fortran assignment statement can be any of the following:

- An arithmetic assignment
- A logical assignment
- A character assignment
- A statement label assignment

§ • A structure field assignment (if the field is a scalar data type)

Arithmetic Assignment

The arithmetic assignment statement has the following form:

object = *arithmetic-expression*

where *object* is one of the following:

- Variable
- Function name (within a function body)
- Subroutine argument
- Array element
- Field of a structure

The type of *object* determines the type of the assignment (INTEGER, REAL, DOUBLE PRECISION or COMPLEX) and the *arithmetic-expression* is coerced into the correct type if necessary.

In the case of:

complex = *real-expression*

the implication is that the real part of the complex number becomes the result of the expression and the imaginary part becomes zero. The same applies if the expression is double precision, except that the expression will be coerced to real.

The following are examples of arithmetic assignment statements.

```
A=(P+Q)*(T/V)
B=R**T**2
```

Logical Assignment Statement

The logical assignment statement has the following form:

object = logical-expression

where *object* is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- A field of a structure

The type of *object* must be logical.

In the following example, `FLAG` takes the logical value `.TRUE.` if `P+Q` is greater than `R`; otherwise `FLAG` has the logical value `.FALSE.`

```
FLAG=(P+Q) .GT. R
```

Character assignment

The form of a character assignment is

object = character-expression

where *object* is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- Character substring
- A field of a structure

The *object* must be of type character.

None of the character positions being defined in *object* can be referenced in the character expression and only such characters as are necessary for the assignment to *object* need to be defined in the character expression. The character expression and `object` can have different lengths. When *object* is longer than the character expression trailing blanks are added to the *object*; and if *object* is shorter than the character expression the right-hand characters of the character expression are truncated as necessary.

In the following example, note that all the variables and arrays are assumed to be of type character.

```
FILE = 'BOOKS'
PLOT(3:8) = 'PLANTS'
TEXT(I,K+1)(2:B-1) = TITLE//X
```

Listing Controls

if77 recognizes three VAX/VMS compiler directives that affect the program listing process:

`%LIST`	Turns on the listing process beginning at the following source code line.
`%NOLIST`	Turns off the listing process (including the `%NOLIST` line itself).
`%EJECT`	Causes a new listing page to be started.

These directives have an effect only when the *–Mlist* compile-time switch is used.

All of the directives must begin in column one.

Data Types 2

Every Fortran element and expression has a data type. The data type of an element may be implicit in its definition or explicitly attached to the element in a declaration statement. This chapter describes the Fortran data types and constants that *if77* supports.

Table 2-1 lists the standard FORTRAN 77 data types. Table 2-2 shows additional data types that *if77* Fortran supports.

Table 2-1. Fortran Standard Data Types

Data Type	Value
INTEGER	An integer number.
REAL	A real number.
DOUBLE PRECISION	A double precision floating point number (real number) taking up two numeric storage units and whose precision is greater than REAL.
LOGICAL	A value which can be either true or false.
COMPLEX	A pair of real numbers used in complex arithmetic.
CHARACTER	A string consisting of one or more printable characters.

A symbolic name for a data type can be followed by a data type length specifier of the form **s*, where *s* is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies and assigns a specific length to the specified item, regardless of the compiler options specified. For example, REAL*8 is equivalent to DOUBLE PRECISION. Table 2-2 shows the lengths of data types, their meanings, and their sizes.

Table 2-2. Data Type Extensions

Type	Meaning	Size
LOGICAL*1	Small LOGICAL	1 byte
LOGICAL*2	Short LOGICAL	2 bytes
LOGICAL*4	LOGICAL	4 bytes
BYTE	Small INTEGER	1 byte
INTEGER*1	Same as BYTE	1 byte
INTEGER*2	Short INTEGER	2 bytes
INTEGER*4	INTEGER	4 bytes
REAL*4	REAL	4 bytes
REAL*8	DOUBLE PRECISION	8 bytes
COMPLEX*8	COMPLEX	8 bytes
COMPLEX*16	DOUBLE COMPLEX	16 bytes

The BYTE type is treated as a signed one-byte integer and is equivalent to LOGICAL*1.

Assignment of a value too big for the data type to which it is assigned is an undefined operation.

A specifier is allowed after a CHARACTER function name even if the CHARACTER type word has a specifier. For example:

```
CHARACTER*4 FUNCTION C*8 (VAR1)
```

Above, the function size specification `C*8` overrides the `CHARACTER*4` specification. Logical data items can be used with any operation where a similar sized integer data item is permissible and vice versa. The logical data item is treated as an integer or the integer data item is treated as a logical of the same size and no type conversion is performed.

§ Floating point data items of type REAL or DOUBLE PRECISION may be used as array subscripts, in computed GOTOs, in array bounds and in alternate returns. *if77* converts the floating point number to an integer.

The data type of the result of an arithmetic expression corresponds to the type of its data. The type of an expression is determined by the rank of its elements. Table 2-3 shows the ranks of the various data types, from lowest to highest.

Table 2-3. Data Type Ranks

Data Type	Rank
LOGICAL	1 (lowest)
INTEGER*2	2
INTEGER*4	3
REAL*4	4
REAL*8 (Double precision)	5
COMPLEX*8 (Complex)	6
COMPLEX*16 (Double complex)	7 (highest)

The data type of a value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. The exception to this rule is that an operation involving a COMPLEX*8 element and a REAL*8 element produces a COMPLEX*16 result. In this operation, the COMPLEX*8 element is converted to a COMPLEX*16 element, which consists of two REAL*8 elements, before the operation is performed.

The type of a logical expression is always a LOGICAL*4 result.

Constants

A constant is an unchanging value. It takes a form corresponding to one of the data types.

if77 supports octal, hexadecimal and Hollerith constants. The use of character constants in a numeric context, for example, in the right-hand side of an arithmetic assignment statement, is supported. These constants assume a data type that conforms to the context in which they appear.

Integer Constants

The form of a decimal integer constant is:

$$[s]\,d_1 d_2 ... d_n$$

where d_i is a digit in the range 0 to 9 and where s is an optional sign. The value of an integer constant must be within the range -2147483648 to 2147483647 inclusive (-2^{31} to (2^{31} - 1)). All integer constants assume a data type of INTEGER*4 and have a 32-bit storage requirement.

Below are several examples of integer constants.

```
+2
-36
437
```

Real Constants

Real constants have two forms, scaled and unscaled. An unscaled real constant consists of a signed or unsigned decimal number. A scaled real constant takes the same form as an unscaled constant, but is followed by a scaling factor using the form:

E+*digits*
E*digit*
E-*digits*

where *digits* is the scaling factor (the power of ten) to be applied to the unscaled constant. The first two forms above are equivalent, that is, a scaling factor without a sign is assumed to be positive.

Table 2-4 shows several examples of real constants.

Table 2-4. Real Constants

Constants	Value
1.0	unscaled single precision constant
1.	unscaled single precision constant
-1.0	signed unscaled single precision constant
6.1E2	is equivalent to 610.0
+2.3E3	is equivalent to 2300.0
-3.5E-1	is equivalent to -0.35

Double Precision Constants

A double precision constant has the same form as a scaled real constant except that the E is replaced by D. Table 2-5 shows several double precision constants.

Table 2-5. Double Precision Constants

6.1D2	is equivalent to 610.0
+2.3D3	is equivalent to 2300.0
-3.5D-1	is equivalent to -0.35
+4D4	is equivalent to 40000

Logical Constants

A logical constant is one of:

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

The logical constants .TRUE. and .FALSE. are defined to be the four-byte values -1 and 0 respectively. By default, a logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise. The option *–Munixlogical* defines a logical expression to be true if its value is non-zero and false otherwise, and defines the internal value of .TRUE. to be 1. Refer to the *Paragon™ System Fortran Compiler User's Guide* for details.

The abbreviations T and F can be used in place of .TRUE. and .FALSE. in data initialization statements and in namelist input.

Complex Constants

A complex constant is held as two real constants separated by a comma and surrounded by parentheses. The first real number is the real part and the second real number is the imaginary part. Together these values represent a complex number. Below are several examples:

```
(3.5,-3.5)
(6.1E2,+2.3E3)
```

Character Constants

Character string constants may be delimited using either an apostrophe (') or a double quote ("). The apostrophe or double quote acts as a delimiter and is not part of the character constant. Use two apostrophes together to include an apostrophe as part of the expression. If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escape. Within character constants, blanks are significant. The length of the string must be at least one character. For further information on the use of the backslash character, refer to *–Mbackslash* in the *Paragon™ System Fortran Compiler User's Guide*.

Below are several examples of character constants.

```
'abc'
'abc '
'ab''c'
```

§ If a character constant is used in a numeric context, for example as the expression on the right side of an arithmetic assignment statement, it is treated as a Hollerith constant. The rules for typing and sizing character constants used in a numeric context are outlined later in the description of Hollerith constants.

Octal and Hexadecimal Constants

The form of an octal constant is:

$'c_1c_2...c_n'O$

The form of a hexadecimal constant is:

$'a_1a_2...a_n'X$

where c_i is a digit in the range 0 to 7 and a_i is a digit in the range 0 to 9 or a letter in the range A to F or a to f (case mixing is allowed). You can specify up to 64 bits (22 octal digits or 16 hexadecimal digits).

Octal and hexadecimal constants are stored as either 32-bit or 64-bit quantities. They are padded on the left with zeroes if needed and assume data types based on how they are used.

The following are the rules for converting these data types:

- An octal or hexadecimal constant is *always* either 32 or 64 bits in size and is typeless. Sign-extension and type-conversion are never performed. All binary operations are performed on 32-bit or 64-bit quantities. This implies that the rules to follow are only concerned with mixing 32-bit and 64-bit data.

- When a constant is used with an arithmetic binary operator (including the assignment operator) and the other operand is typed, the constant assumes the type and size of the other operand.

- When a constant is used in a relational expression such as `.EQ.`, its size is chosen from the operand having the largest size. This implies that 64-bit comparisons are possible.

- When a constant is used as an argument to the generic AND, OR, EQV, NEQV, SHIFT, or COMPL function, a 32-bit operation is performed if no argument is more than 32 bits in size; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.

  When a constant is used as an actual argument in any other context, no data type is assumed; however, a length of four bytes is always used. If necessary, truncation on the left occurs.

- When a specific 32-bit or 64-bit data type is required, that type is assumed for the constant. Array subscripting is an example.

- When a constant is used in a context other than those mentioned above, an INTEGER*4 data type is assumed. Logical expressions and binary arithmetic operations with other untyped constants are examples.

- When the required data type for a constant implies that the length needed is more than the number of digits specified, the left-most digits have a value of zero. When the required data type for a constant implies that the length needed is less than the number of digits specified, the constant is truncated on the left. Truncation of nonzero digits is allowed.

  In the example below, the constant I (of type INTEGER*4) and the constant J (of type INTEGER*2) will have hex values 1234 and 4567, respectively. The variable D (of type REAL*8) will have the hex value x4000012345678954 after its second assignment:

```
I = '1234'X        ! Leftmost Pad with zero.
J = '1234567'X     ! Truncate Leftmost 3 hex digits
D = '40000123456789ab'X
D = NEQV(D,'ff'X)  ! 64-bit Exclusive Or
```

Hollerith Constants

The form of a Hollerith constant is:

$$n\text{H}c_1c_2...c_n$$

where n specifies the positive number of characters in the constant and cannot exceed 2000 characters. A Hollerith constant is stored as a byte string with four characters per 32-bit word. Hollerith constants are untyped arrays of INTEGER*4. The last word of the array is padded on the right with blanks if necessary. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected. Hollerith constants are permitted with the %REF built-in function (for more information on the built-in VAX/VMS functions, see Appendix B, *VAX System Subroutines and Built-in Functions*.) The data type of a Hollerith constant used in a numeric expression is determined by the following rules:

- Sign-extension is never performed.
- The byte size of the Hollerith constant is determined by its context and is not strictly limited to 32 or 64 bits like hexadecimal and octal constants.
- When the constant is used with a binary operator (including the assignment operator), the data type of the constant assumes the data type of the other operand.
- When a specific data type is required, that type is assumed for the constant. When an integer or logical is required, INTEGER*4 and LOGICAL*4 are assumed. When a float is required, REAL*4 is assumed (array subscripting is an example of the use of a required data type).
- When a constant is used as an argument to certain generic functions (AND, OR, EQV, NEQV, SHIFT, and COMPL), a 32-bit operation is performed if no argument is larger than 32 bits; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.

  When a constant is used as an actual argument, no data type is assumed and the argument is passed as an INTEGER*4 array. Character constants are passed by descriptor only.

- When a constant is used in any other context, a 32-bit INTEGER*4 array type is assumed.

When the length of the Hollerith constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right.

Arrays

An array is a group of consecutive, contiguous storage locations associated with a symbolic name which is the array name. Each individual element of storage, called the array element, is referenced by the array name modified by a list of subscripts. Arrays are declared with type declaration statements, DIMENSION statements and COMMON statements; they are not defined by implicit reference. These declarations will introduce an array name and establish the number of dimensions and the bound of each dimension. If a symbol, modified by a list of subscripts is not defined as an array, then it will be assumed to be a FUNCTION reference with an argument list.

An Array Declaration Element

An array declaration has the following form:

```
name([lb:]ub[,[lb:]ub]... )
```

where *name* is the symbolic name of the array, *lb* is the specification of the lower bound of the dimension and *ub* is the specification of the upper bound. The upper bound *ub* must be greater than the lower bound *lb*. The values *lb* and *ub* may be negative. The bound *lb* is taken to be 1 if it is not specified. The difference (*ub*-*lb*+1) specifies the number of elements in that dimension. The number of *lb*,*ub* pairs specifies the dimension of the array. The total amount of storage of the array is:

```
(ub-lb+1)*(ub-lb+1)*...
```

However, the dimension specifiers of a subroutine argument may themselves be subroutine arguments or members of COMMON.

Subscripts

A subscript is used to locate an array element for access. An array name qualified by a subscript list has the following form:

```
name(sub[,sub]...)
```

where there must be one *sub* entry for each dimension in array *name*.

Each *sub* must be an integer expression yielding a value which is within the range of the lower and upper bounds. Arrays are stored as a linear sequence of values in memory and are held such that the first element is in the first store location and the last element is in the last store location. In a multi-dimensional array the first subscript varies more rapidly than the second, the second more rapidly than the third, and so on (column major order).

Character Substring

A character substring is a contiguous portion of a character variable and is of type character. A character substring can be referenced, assigned values and named. It can take either of the following forms:

```
character_variable_name (x1 : x2)

character_array (subscripts) (x1 : x2)
```

where *x1* and *x2* are integers and *x1* denotes the left-hand character position and *x2* the right-hand one. These are known as substring expressions. In substring expressions *x1* must be both greater than or equal to 1 and less than *x2* and *x2* must be less than or equal to the length of the character variable or array element.

For example:

`J(2:4)` the characters in positions 2 to 4 of character variable `J`.

`K(3,5)(1:4)` the characters in positions 1 to 4 of array element `K(3,5)`.

A substring expression can be any valid integer expression and may contain array element or function references.

§ Structures

A structure is a user-defined aggregate data type having the following form:

```
STRUCTURE [/structure_name/][field_namelist]
          field_declaration
          [field_declaration]
                  ...
          [field_declaration]
END STRUCTURE
```

Where:

structure_name is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.

field_namelist is a list of fields having the structure of the associated structure declaration. A *field_namelist* is allowed only in nested structure declarations.

field_declaration can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

Fields within structures conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict. Also, because records use periods to separate fields, it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Fields in a structure are aligned as required by hardware; therefore a structure's storage requirements are machine-dependent. Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.

Data initialization can occur for the individual fields.

§ Records

A record is a user-defined aggregate data item having the following form:

```
RECORD /structure_name/record_namelist
       [,/structure_name/record_namelist]
            ...
       [,/structure_name/record_namelist]
```

Where:

structure_name is the name of a previously declared structure.

record_namelist is a list of one or more variable or array names separated by commas.

You create memory storage for a record by specifying a structure name in the RECORD statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, recordname.fieldname). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, INTEGER), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of COMMON, SAVE, NAMELIST, DATA and EQUIVALENCE statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

The following is an example of RECORD and STRUCTURE usage.

```
STRUCTURE /person/    ! Declare a structure to define a person
   INTEGER id
   LOGICAL living
   CHARACTER*5 first, last, middle
   INTEGER age
END STRUCTURE
   ! Define population to be an array where each element is of
   ! type person.  Also define a variable, me, of type person.
RECORD /person/ population(2), me
   ...
me.age = 34             ! Assign values for the variable me to
me.living = .TRUE.      ! some of the fields.
me.first = 'Steve'
me.id = 542124822
   ...
population(1).last = 'Jones' ! Assign the "last" field of
                             ! element 1 of array population.
population(2) = me           ! Assign all the values of record
                             ! "me" to the record population(2)
```

§ UNION and MAP Declarations

A UNION declaration is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a MAP declaration, with one or more fields per MAP declaration.

Union declarations are used when one wants to use the same area of memory to alternately contain one of two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. The format of a UNION statement is as follows:

```
UNION
      map_declaration
      [map_declaration]
                  ...
      [map_declaration]
END UNION
```

The format of the *map_declaration* is as follows:

```
MAP
      field_declaration
      [field_declaration]
                  ...
      [field_declaration]
END MAP
```

where *field_declaration* is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

Field alignment within multiple map declarations is performed as previously defined in structure declarations.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus possibly space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations more closely resemble union declarations for the language C. The main difference is that the language C requires one to associate a name with each "map" (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following is an example of RECORD, STRUCTURE, MAP and UNION usage. The size of each element of the *recarr* array would be the size of *typetag* (4 bytes) plus the size of the largest MAP, the *employee* map (24 bytes).

```
STRUCTURE /account/
    INTEGER typetag                              ! Tag to determine defined map.
    UNION
       MAP                                       ! Structure for an employee
                CHARACTER*12      ssn            ! Social Security Number
                REAL*4            salary
                CHARACTER*8       empdate        ! Employment date
       END MAP
       MAP                                       ! Structure for a customer
                INTEGER*4         acct_cust
                REAL*4            credit_amt
                CHARACTER*8       due_date
       END MAP
       MAP                                       ! Structure for a supplier
                INTEGER*4         acct_supp
                REAL*4            debit_amt
                BYTE              num_items
                BYTE              items(12)      ! Items supplied
       END MAP
    END UNION
END STRUCTURE

RECORD /account/ recarr(1000)
```

Data Initialization

§ Within data type declaration statements, data initialization is allowed. Data is initialized by placing values bounded by slashes immediately following the symbolic name (variable or array) to be initialized. Initialization of fields within structure declarations is allowed, but initialization of unnamed fields and records is not.

Hollerith, octal and hexadecimal constants can be used to initialize data in both data type declarations and in DATA statements. Truncation and padding occur for constants that differ in size from the declared data item (as specified in the discussion of constants above).

§ Pointer Variables

The POINTER statement declares a scalar variable to be a *pointer* variable (of data type INTEGER), and another variable to be its *pointer-based* variable.

The syntax of the POINTER statement is:

```
POINTER (p1, v1) [, (p2, v2) ...]
```

v1 and *v2* are *pointer-based* variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, according to the rules for *adjustable* dummy arrays and dimension declarators.

p1 and *p2* are the pointer variables corresponding to *v1* and *v2*. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is *dereferenced*).

The pointer-based variable does not have an address until its corresponding pointer is defined. The pointer *is defined* in one of the following ways:

- By assigning the value returned by the LOC function.
- By assigning a value defined in terms of another pointer variable.
- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

The following code illustrates the use of pointers:

```
REAL XC(10)
REAL X
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))

P = LOC(IC)
I = 0                          ! IC gets 0

P = LOC(XC)
Q =  P + 20                    ! same as LOC(XC(6))
X(1) = 0                       ! XC(6) gets 0
ALLOCATE (X)                  ! Q  locates  an  allocated  memory  area
```

Restrictions

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is *assumed to be defined.*
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram. If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarator(s) are defined with an integer value at the time the SUBROUTINE or FUNCTION is called. For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is *assumed not to overlap* with another pointer-based variable.

Fortran Statements 3

This chapter describes each of the Fortran statements. Each description includes a brief summary of the statement, a syntax description, a complete description and an example. The statements are listed in alphabetical order. The first section lists terms that are used throughout the chapter.

Definition of Terms

character scalar memory reference
: is an character variable, a character array element, or a character member of a structure.

integer scalar memory reference
: is an integer variable, an integer array element, or an integer member of a structure.

logical scalar memory reference
: is an logical variable, a logical array element, or a logical member of a structure.

§ ACCEPT

The ACCEPT statement causes formatted input to be read on standard input, *stdin*. ACCEPT is identical to the READ statement with a unit specifier of asterisk (*).

Syntax

```
ACCEPT f [,iolist]

ACCEPT namelist
```

f format-specifier. A * indicates list directed input.

iolist is a list of variables to be input.

namelist is the name of a namelist specified with the NAMELIST statement.

Examples

```
      ACCEPT *, IA, ZA
      ACCEPT 99 I, J, K
      ACCEPT SUM
99    FORMAT(I2, I4, I3)
```

§ ALLOCATE

The ALLOCATE statement allocates storage for each pointer-based variable and allocatable common block which appears in the statement.

Syntax

```
ALLOCATE ( name[, name ] ... [ , STAT= var ] )
```

name is a pointer-based variable or the name of an allocatable COMMON enclosed in slashes.

var is an integer variable, integer array element or an integer member of a STRUCTURE (that is, an integer scalar memory reference).

Description

For a pointer based variable, its associated pointer variable is defined with the address of the allocated memory area. If the specifier STAT= is present, successful execution of the ALLOCATE statement causes the status variable to be defined with a value of zero. If an error occurs during execution of the statement and the specifier STAT= is present, the status variable is defined to have the integer value one. If an error occurs and the specifier STAT= is not present, program execution is terminated.

A dynamic, or allocatable COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed.

Examples

```
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

ASSIGN

The assign statement assigns a statement label to a variable.

Syntax

```
ASSIGN a TO b
```

a is the statement label.

b is an integer variable.

Description

Executing an ASSIGN statement assigns a statement label to an integer variable. This is the only way that a variable may be defined with a statement label value. The statement label must be:

- A statement label in the same module as the ASSIGN statement.
- The label of an executable statement or a FORMAT statement.

A variable must be defined with a statement label when it is referenced:

- in an assigned GOTO statement.
- as a format identifier in an input/output statement and while so defined must not be referenced in any other way.

An integer variable defined with a statement label can be redefined with a different statement label, the same statement label or with an integer value.

Example

```
      ASSIGN 40 TO K

      GO TO K

40    L = P + I + 56
```

BACKSPACE

When a BACKSPACE statement is executed the file connected to the specified unit is positioned before the preceding record.

Syntax

```
BACKSPACE unit

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

unit is the unit specifier.

errs an error specifier which is a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by *errs*.

ios is an integer scalar memory reference that is defined as zero if no error condition exists or a positive integer when there is an error condition.

Description

If there is no preceding record the position of the file is not changed. A BACKSPACE statement cannot be executed on a file that does not exist. You must not issue a BACKSPACE statement for a file that is open for direct or append access.

Examples

```
BACKSPACE 4

BACKSPACE ( UNIT=3 )

BACKSPACE ( 7, IOSTAT=IOCHEK, ERR=50 )
```

BLOCK DATA

The BLOCK DATA statement introduces a module that sets up initial values in COMMON blocks. No executable statements are allowed in a BLOCK DATA module.

Syntax

```
BLOCK DATA [name]
```

name is a symbol identifying the module and must be unique among all global names (COMMON block names and among all other module names). If missing, the module is given a default name.

Example

```
      BLOCK DATA
      COMMON /SIDE/ BASE, ANGLE, HEIGHT, WIDTH
      INTEGER SIZE
      PARAMETER (SIZE=100)
      INTEGER BASE(0:SIZE)
      REAL WIDTH(0:SIZE), ANGLE(0:SIZE), WIDTH(O:SIZE)
      DATA/(BASE(I),I=0,SIZE)/SIZE*-1,-1/,
     +(WIDTH(I),I=0,SIZE)/SIZE*0.0,0.0/
      END
```

§ BYTE

The BYTE statement establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
BYTE name [/clist/], ...
```

name is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist is a list of constants that initialize the data, as in a DATA statement.

Description

Byte statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. BYTE declaration statements must not be labeled.

Example

```
BYTE TB3, SEC, STORE (5,5)
```

CALL

The CALL statement transfers control to a subroutine.

Syntax

```
CALL subroutine [([ argument [, argument]...])]
```

subroutine is the name of the subroutine.

argument is the actual argument being passed to the subroutine. The first argument corresponds to the first dummy argument in the SUBROUTINE statement and so on.

Description

Actual arguments can be expressions including: constants, scalar variables, function references and arrays.

Actual arguments can also be alternate return specifiers. Alternate return specifiers are labels prefixed by asterisks (*) or ampersands (&) (the ampersand is an extension from FORTRAN 77§).

Examples

```
CALL CRASH            ! no arguments

CALL BANG(1.0)        ! one argument

CALL WALLOP(V, INT)   ! two arguments

CALL ALTRET(I, *10, *20)


SUBROUTINE ONE
DIMENSION  ARR ( 10, 10 )
REAL WORK
INTEGER ROW, COL
PI=3.142857
CALL EXPENS(ARR,ROW,COL,WORK,SIN(PI/2)+3.4)
RETURN
END
```

CHARACTER

The CHARACTER statement establishes the data type of a variable by explicitly attaching the name of a variable to a character data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
CHARACTER [*len] name [*len] [/clist/], ...
```

name is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

len is a constant or *. (*) is only valid if the corresponding name is a dummy argument.

clist is a list of constants that initialize the data, as in a DATA statement.

Description

Character type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Examples

```
CHARACTER A*4, B*6, C
```

A is 4 and B is 6 characters long and C is 1 character long.

CLOSE

The CLOSE statement terminates the connection of the specified file to a unit.

Syntax

```
CLOSE ([UNIT=] u [,IOSTAT=ios] [,ERR= errs ]
      [,STATUS= sta] [,DISPOSE= sta] [,DISP= sta])
```

u — the external unit specifier where *u* is an integer.

ios — is an integer scalar memory reference; if this is included *ios* becomes defined with `0 (zero)` if no error condition exists or a positive integer when there is an error condition.

errs — is an error specifier in the form of a statement label of an executable statement in the same module. If an error condition occurs, execution continues with the statement specified by *errs*.

sta — is a character expression, where case is insignificant, specifying the file status and the same keywords are used for the dispose status. Status can be KEEP or DELETE. KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement; conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

Description

A unit may be the subject of a CLOSE statement from within any module. If the unit specified does not exist or has no file connected to it the use of the CLOSE statement has no effect. Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. Note that an implicit CLOSE is executed when a program stops.

Example

In the following example the file on unit 6 is closed and deleted.

```
CLOSE(UNIT=6,STATUS='DELETE')
```

COMMON (Static and Dynamic)

The COMMON statement defines contiguous blocks of storage. Each block is identified by a symbolic name and the order of variables and arrays is defined in the COMMON block containing them. There are two forms of the COMMON statement, a static form and a dynamic form.

Syntax

```
        COMMON /name/nlist [, /name/nlist]...
§       COMMON [,ALLOCATABLE] /name/nlist [,/name/nlist]...
```

name is the name of each common block and is declared between the /.../ delimiters.

nlist is a list of scalar and array names where the arrays may be defined in DIMENSION statements or formally declared by their inclusion in the COMMON block.

Description (static COMMON)

The name of the COMMON block need not be supplied; this is the Fortran BLANK COMMON feature. In this case the compiler will use a default name which is implementation-specific. There can be several COMMON block statements of the same name in a module; these are effectively treated as one statement, with variables and array addresses concatenated from one COMMON statement of the same name to the next. This is an alternative to the use of continuation lines when declaring a common block with many symbols.

Common blocks with the same name that are declared in different modules share the same storage area when combined into one executable program.

COMMON (Static and Dynamic) *(cont.)*

Example (static COMMON)

```
DIMENSION R(10)
COMMON /HOST/ A, R, Q(3), U
```

This declares a common block of data memory called HOST where A will be held in the first memory location, R(1) . . . R(10) will be held in the next ten locations, Q(1) . . . Q(3) in the next three and U in the fifteenth location. Note the different types of declaration used for R (declared in a DIMENSION statement) and Q (declared in the COMMON statement). The declaration of HOST in a SUBROUTINE in the same executable program will share the same data area.

```
SUBROUTINE DEMO
COMMON/HOST/STORE(15)
   .
   .
   .
RETURN
END
```

If the main program has the common block declaration as in the previous example, the COMMON statement in the subroutine causes STORE(1) to correspond to A, STORE(2) to correspond to R(1), STORE(3) to correspond to R(2), and so on through to STORE(15) corresponding to the variable U.

You can name records within a COMMON block. Because the storage requirements of records are machine-dependent, the size of a COMMON block containing records may vary between machines. Note that this may also affect subsequent equivalence associations to variables within COMMON blocks that contain records.

§ Both character and non-character data may reside in one COMMON block. Data is aligned within the COMMON block in order to conform to machine-dependent alignment requirements.

A COMMON block may be data initialized in more than one program unit if the existing system environment allows it (note that COFF-based systems do not). It is up to the programmer to make sure that data within one COMMON block is not initialized more than once.

Blank COMMON may be data initialized.

COMMON (Static and Dynamic) *(cont.)*

§ Description (dynamic COMMON)

A *dynamic*, or *allocatable*, COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed.

If the allocatable attribute is present, all named COMMON blocks appearing in the COMMON statement are marked as *allocatable*. Like a normal COMMON statement, the name of an *allocatable* COMMON block may appear in more than one COMMON statement. Note that the ALLOCATABLE attribute need not appear in every COMMON statement.

The following restrictions apply to the dynamic COMMON statement:

- Before members of an allocatable COMMON block can be referenced, the common block must have been explicitly allocated using the ALLOCATE statement.
- The data in an allocatable common block cannot be initialized.
- The memory used for an allocatable common block may be freed using the DEALLOCATE statement.
- If a SUBPROGRAM declares a COMMON block to be allocatable, all other subprograms containing COMMON statements of the same COMMON block must also declare the COMMON to be allocatable.

§ Example (dynamic COMMON)

```
COMMON, ALLOCATABLE /ALL1/ A, B, /ALL2/ AA, BB
COMMON /STAT/ D, /ALL1/ C
```

This declares the following variables:

ALL1	is an allocatable COMMON block whose members are A, B, and C.
ALL2	is an allocatable COMMON block whose members are AA, and BB.
STAT	is a statically-allocated COMMON block whose only member is D.

A reference to a member of an allocatable COMMON block appears in a Fortran statement just like a member of a normal (static) COMMON block. No special syntax is required to access members of allocatable common blocks. For example, using the above declarations, the following is a valid *if77* statement:

```
AA = B * D
```

COMPLEX

The COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a complex data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
COMPLEX name [/clist/] [, name] [/clist/] ...
```

name is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement below for an explanation of array declarators).

clist is a list of constants that initialize the data, as in a DATA statement.

Description

COMPLEX statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. COMPLEX statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Example

```
COMPLEX CURRENT
```

CONTINUE

This CONTINUE statement passes control to the next statement. It is supplied mainly to overcome the problem that transfer of control statements are not allowed to terminate a DO loop.

Syntax

```
CONTINUE
```

Example

```
      DO 100 I = 1,10
         SUM = SUM + ARRAY (I)
         IF(SUM .GE. 1000.0) GOTO 200
100   CONTINUE
200   ...
```

DATA

The DATA statement assigns initial values to variables before execution.

Syntax

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

vlist is a list of variable names, array element names or array names separated by commas.

dlist is a list of constants or PARAMETER constants, separated by commas, corresponding to elements in the *vlist*. An array name in the *vlist* demands that *dlist* constants be supplied to fill every element of the array.

Repetition of a constant is provided by using the form:

```
n*constant-value
```

n a positive integer, is the repetition count.

Example

```
REAL A, B, C(3), D(2)
DATA A, B, C(1), D /1.0, 2.0, 3.0, 2*4.0/
```

This performs the following initialization:

```
A = 1.0
B = 2.0
C(1) = 3.0
D(1) = 4.0
D(2) = 4.0
```

§ DEALLOCATE

The DEALLOCATE statement causes the memory allocated for each pointer-based variable or allocatable COMMON block that appears in the statement to be deallocated (freed).

Syntax

```
DEALLOCATE ( al [, al ] ... [ , STAT= var ] )
```

al is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes.

var is an integer variable, integer array element, or an integer member of a structure.

Description

An attempt to deallocate a pointer-based variable or an allocatable COMMON block which was not created by an ALLOCATE statement results in an error condition.

If the specifier `STAT=` is present, successful execution of the statement causes *var* to be defined with the value of zero. If an error occurs during the execution of the statement and the specifier `STAT=` is present, the status variable is defined to have the integer value one. If an error occurs and the specifier `STAT=` is not present, program execution is terminated.

Examples

```
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

§ DECODE

The DECODE statement transfers data between variables or arrays in internal storage and translates that data from internal to character form, and vice versa, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE and READ statements.

Syntax

```
DECODE (c, f, b [ ,IOSTAT= ios ] [, ERR= errs]) [ list ]
```

c is an integer expression specifying the number of bytes involved in translation.

f is the format identifier.

b is a scalar or array reference for the buffer area.

ios is the an integer scalar memory reference which is the input/output specifier: if this is specified *ios* becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

errs an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by *errs*.

list is the buffer area either containing data or receiving data.

Example

```
      DIMENSION K(3)
      CHARACTER*12 A
      DATA A/'123456789012'/
      DECODE (12,100,A) K
100   FORMAT (3I4)
```

This translates the 12 characters in A to integer form and stores them in array K.

DIMENSION

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

Syntax

```
DIMENSION name ([lb:]ub[,[lb:]ub]...)[,name([lb:]ub[,[lb:]ub]...)]
```

name is the symbolic name of an array.

lb:ub is a dimension declarator specifying the bounds for a dimension (the lower bound *lb* and the upper bound *ub*). *lb* and *ub* must be integers with *ub* greater than lb. The lower bound *lb* is optional; if it is not specified, it is taken to be 1.

Description

DIMENSION can be used in a subroutine to establish an argument as an array, and in this case the declarator can use expressions formed from integer variables and constants to establish the dimensions (adjustable arrays). Note however that these integer variables must be either arguments or declared in COMMON; they cannot be local. Note that in this case the function of DIMENSION is merely to supply a mapping of the argument to the subroutine code, and not to allocate storage.

If an array is a dummy argument its last dimension may be an * (assumed size array)

The typing of the array in a DIMENSION statement is defined by the initial letter of the array name in the same way as variable names. The letters `I,J,K,L,M` and `N` imply that the array is of INTEGER type and an array with a name starting with any of the letters `A` to `H` and `O` to `Z` will be of type REAL, unless overridden by an IMPLICIT or type declaration statement. Arrays may appear in type declaration and COMMON statements but the array name can appear in only one array declaration.

DIMENSION statements must not be labeled.

DIMENSION *(cont.)*

Example

```
DIMENSION ARRAY1(3:10), ARRAY2(3,-2:2)
```

This specifies ARRAY1 as a vector having eight elements with the lower bound of 3 and the upper bound of 10 and ARRAY2 as a matrix of two dimensions having fifteen elements. The first dimension has three elements and the second has five with bounds from -2 to 2.

```
CHARACTER B(0:20)*4
```

This sets up an array B with 21 character elements each having a length of four characters. Note that the array has been dimensioned in a type declaration statement and therefore cannot subsequently appear in a DIMENSION statement.

DO (Iterative)

The DO statement introduces an iterative loop and specifies the loop control index and parameters.

Syntax

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

label labels the last executable statement in the loop (this must not be a transfer of control statement).

i is the name of a variable called the DO variable.

e1 is an expression which yields an initial value for *i*.

e2 is an expression which yields a final value for i.

e3 is an optional expression yielding a value specifying the increment value for *i*. The default for *e3* is 1.

Description

§ If the optional label, *label*, is not included, the DO statement must be terminated by an END DO statement.

The DO loop consists of all the executable statements after the specifying DO statement up to and including the labeled statement, called the terminal statement. The label is optional. If omitted, the terminal statement of the loop is an END DO statement.

§ END DO may be used to terminate the DO loop even if a label is specified.

DO (Iterative) *(cont.)*

Before execution of a DO loop, an iteration count is initialized for the loop. This value is the number of times the DO loop is executed, and is

```
INT((e2-e1+e3)/e3)
```

If the value obtained is negative or zero that the loop is *not* executed.

The DO loop is executed first with *i* taking the value *e1*, then the value (*e1+e3*), then the value (*e1+e3+e3*), etc.

It is possible to jump out of a DO loop and jump back in, as long as the do index variable has not been adjusted.

§ Nested DO loops may share the same labeled terminal statement if required. They may not share an END DO statement.

In a nested DO loop, it is legal to transfer control from an inner loop to an outer loop. It is illegal, however, to transfer into a nested loop from outside the loop.

Example

```
      DO 100 J = -10,10
            DO 100 I = -5,5
100            SUM = SUM + ARRAY (I,J)
```

§ DO WHILE

The DO WHILE statement introduces a logical do loop and specifies the loop control expression.

The DO WHILE statement executes for as long as a logical expression tested at the beginning of each iteration continues to be true. If the expression is false, control transfers to the statement following the loop.

Syntax

```
DO [label[,]] WHILE expression
```

The end of the loop is specified in the same way as for an iterative loop, either with a labeled statement or an END DO.

label labels the last executable statement in the loop (this must not be a transfer of control).

expression is a logical expression and label.

Description

The *expression* is evaluated. If it is .FALSE., the loop is not entered. If it is .TRUE., the loop is executed once. Then *expression* is evaluated again, and the cycle is repeated until *expression* evaluates .FALSE..

Example

```
CHARACTER*132 LINE
I = 1
LINE(132:) = 'x')
DO WHILE (LINE(I:I) .EQ. ' ')
   I = I + 1
END DO
```

DOUBLE COMPLEX

The DOUBLE COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
DOUBLE COMPLEX name [/clist/] [, name] [/clist/]...
```

name is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist is a list of constants that initialize the data, as in a DATA statement.

Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Examples

```
DOUBLE COMPLEX CURRENT, NEXT
```

DOUBLE PRECISION

The DOUBLE PRECISION statement establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
DOUBLE PRECISION name [/clist/] [, name] [/clist/]...
```

name is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist is a list of constants that initialize the data, as in a DATA statement.

Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Examples

```
DOUBLE PRECISION PLONG
```

ELSE

The ELSE statement begins an ELSE block of an IF block and encloses a series of statements that are conditionally executed.

Syntax

```
IF logical-expression THEN
 statements
ELSE IF logical-expression THEN
 statements
ELSE
 statements
ENDIF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an IF block.

Example

```
IF (I.GT.70) THEN
     M=1
ELSE IF (I.LT.5) THEN
     M=2
ELSE IF (I.LT.16) THEN
     M=3
ENDIF
IF (I.LT.15) THEN
     M = 4
ELSE
     M=5
ENDIF
```

ELSE IF

The ELSE IF statement begins an ELSE IF block of an IF block series and encloses statements that are conditionally executed.

Syntax

```
IF logical-expression THEN
 statements
ELSE IF logical-expression THEN
 statements
ELSE
 statements
ENDIF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an IF block.

Example

```
IF (I.GT.70) THEN
     M=1
ELSE IF (I.LT.5) THEN
     M=2
ELSE IF (I.LT.16) THEN
     M=3
ENDIF
IF (I.LT.15) THEN
     M = 4
ELSE
     M=5
ENDIF
```

§ ENCODE

The ENCODE statement transfers data between variables or arrays in internal storage and translates that data from internal to character form, and vice versa, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE and READ statements.

Syntax

```
ENCODE (c,f,b[,IOSTAT=ios] [,ERR=errs])[list]
```

c is an integer expression specifying the number of bytes involved in translation.

f is the format identifier.

b is a scalar or array reference for the buffer area.

ios is the an integer scalar memory reference which is the input/output specifier: if this is included, *ios* becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

errs an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by *errs*.

list is the buffer area either containing data or receiving data.

Example

```
      DIMENSION K(3)
      CHARACTER*12 A
      DATA A/'123456789012'/
      DECODE (12,100,A) K
100   FORMAT (3I4)
      ENCODE (12,100,B) K(3), K(2), K(1)
```

The DECODE statement translates the 12 characters in the character variable A to integer form and stores them in array K. The ENCODE statement translates the values in K(3), K(2), and K(1) to character form and stores them in the character variable B.

END

The END statement terminates a module. It may be the last statement in a compilation or it may be followed by a new module.

Syntax

```
END
```

Description

The END statement is executable, and has the same effect as a RETURN statement in a SUBROUTINE or FUNCTION, or the effect of a STOP statement in a PROGRAM module.

§ END DO

The END DO statement terminates a DO or DO WHILE loop.

Syntax

```
END DO
```

Description

The END DO statement terminates an indexed DO or DO WHILE statement which does not contain a terminal-statement label.

The END DO statement may also be used as a labeled terminal statement if the DO or DO WHILE statement contains a terminal-statement label.

END FILE

When an END FILE statement is executed an endfile record is written to the file as the next record. The file is then positioned after the endfile record. Note that only records written prior to the endfile record can be read later.

Syntax

```
END FILE u
END FILE ([UNIT=]u, [,IOSTAT=ios] [,ERR=errs])
```

u is the external unit specifier where u is an integer.

IOSTAT=*ios* an integer scalar memory reference which is the input/output specifier: *ios* becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

ERR=*errs* an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs, execution continues with the statement specified by *errs*.

A BACKSPACE or REWIND statement must be used to reposition the file after an END FILE statement prior to the execution of any data transfer statement. A file is created if there is an END FILE statement for a file connected but not in existence.

Examples

```
END FILE(20)
END FILE(UNIT=34, IOSTAT=IOERR, ERR=140)
```

END IF

The END IF statement terminates an IF or ELSE IF block.

Syntax

```
END IF
```

Description

See the BLOCK IF statement for details.

§ END MAP

The END MAP statement terminates a MAP declaration.

Syntax

```
END MAP
```

Description

See the MAP statement for details.

§ END STRUCTURE

The END STRUCTURE statement terminates a STRUCTURE declaration.

Syntax

```
END STRUCTURE
```

Description

See the STRUCTURE statement for details.

§ END UNION

The END UNION statement terminates a UNION declaration.

Syntax

```
END UNION
```

Description

See the UNION statement for details.

ENTRY

The ENTRY statement allows a subroutine or function to have more than one entry point.

Syntax

```
ENTRY name [(variable, variable...)]
```

name is the symbolic name, or entry name, by which the subroutine or function may be referenced.

variable is a dummy argument. A dummy argument may be a variable name, array name, dummy procedure or, if the ENTRY is in a subroutine, an asterisk. If there are no dummy arguments *name* may optionally be followed by ( ). There may be more than one ENTRY statement within a subroutine or function, but they must not appear within a block IF or DO loop.

Description

The *name* of an ENTRY must *not* be used as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement, nor may it appear in an EXTERNAL statement.

Within a function a variable name which is the same as the entry name may not appear in any statement that precedes the ENTRY statement, except in a type statement.

If *name* is of type character the names of each entry in the function and the function name must be of type character. If the function name or any entry name has a length of (*) all such names must have a length of (*); otherwise they must all have a length specification of the same integer value.

A *name* which is used as a dummy argument must not appear in an executable statement preceding the ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement. Neither must it appear in the expression of a statement function unless the name is also a dummy argument of the statement function, or appears in a FUNCTION or SUBROUTINE statement, or in an ENTRY statement that precedes the statement function statement.

If a dummy argument appears in an executable statement, execution of that statement is only permitted during the execution of a reference to the function or subroutine if the dummy argument appears in the dummy argument list of the procedure name referenced.

When a subroutine or function is called using the entry name, execution begins with the statement immediately following the ENTRY statement. If a function entry has no dummy arguments the function must be referenced by *name*() but a subroutine entry without dummy arguments may be called with or without the parentheses after the entry name.

ENTRY *(cont.)*

An entry may be referenced from any module except the one in which it is defined.

The order, type, number and names of dummy arguments in an ENTRY statement can be different from those used in the FUNCTION, SUBROUTINE or other ENTRY statements in the same module but each reference must use an actual argument list which agrees in order, number and type with the dummy argument list of the corresponding FUNCTION, SUBROUTINE or ENTRY statement. When a subroutine name or an alternate return specifier is used as an actual argument there is no need to match the type.

Entry names within a FUNCTION subprogram need not be of the same data type as the function name, but they all must be consistent within one of the following groups of data types:

- BYTE, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*2, LOGICAL*4, REAL*4, REAL*8, COMPLEX*8
- COMPLEX*16
- CHARACTER

If the function is of character data type, all entry names must also have the same length specification as that of the function.

Example

```
FUNCTION SUM(TALL,SHORT,TINY)
    .
SUM=TALL-(SHORT+TINY)
RETURN
ENTRY SUM1(X,LONG,TALL,WIDE,NARROW)
  .
  .
SUM1=(X*LONG)+(TALL*WIDE)+NARROW
RETURN

ENTRY SUM2(SHORT,SMALL,TALL,WIDE)
  .
  .
SUM2=(TALL-SMALL)+(WIDE-SHORT)
RETURN
END
```

ENTRY *(cont.)*

When the calling program calls the function SUM it can do so in one of three ways depending on which ENTRY point is desired.

For example if the call is:

```
Z=SUM2(LITTLE,SMALL,BIG,HUGE)
```

the ENTRY point is SUM2.

If the call is:

```
Z=SUM(T,X,Y)
```

the ENTRY point is SUM and so on.

EQUIVALENCE

The EQUIVALENCE statement allows two or more named regions of data memory to share the same start address.

Syntax

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

list is a set of identifiers (variables, arrays or array elements) which are to be associated with the same address in data memory. The items in a *list* are separated by commas, and there must be at least two items in each *list*. When an array element is chosen, the subscripts must be integer constants or integer PARAMETER constants.

Description

§ An array element may be identified with a single subscript in an EQUIVALENCE statement even though the array is defined to be a multidimensional array.

§ Equivalence of character and non-character data is allowed as long as misalignment of non-character data does not occur.

Records and record fields cannot be specified in EQUIVALENCE statements.

The statement can be used to make a single region of data memory have different types, so that, for example, the imaginary part of a complex number can be treated as a real value or arrays can be made to overlap. This allows the same region of memory to be dimensioned in several different ways.

Example

```
COMPLEX NUM
REAL QWER(2)
EQUIVALENCE (NUM,QWER(1))
```

In the above example `QWER(1)` is the real part of `NUM` and `QWER(2)` is the imaginary part. EQUIVALENCE statements are illegal if there is any attempt to make a mapping of data memory inconsistent with its linear layout.

EXTERNAL

The EXTERNAL statement identifies a symbolic name as an external or dummy procedure. This procedure can then be used as an actual argument.

Syntax

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

proc is the name of an external procedure, dummy procedure or block data module. When an external or dummy procedure name is used as an actual argument in a module it must appear in an EXTERNAL statement in that module.

Description

If an intrinsic function appears in an EXTERNAL statement an intrinsic function of the same name cannot then be referenced in the module. A symbolic name can appear only once in all the EXTERNAL statements of a module.

FORMAT

The FORMAT statement specifies format requirements for input or output.

Syntax

label FORMAT (*list-items*)

list-items can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant which defines the number of repeats.
- Nonrepeatable editor commands.
- A format specification list optionally preceded by an integer constant which defines the number of repeats.

Description

Each action of format control depends on the next edit code and the next item in the input/output list where one is used. If an input/output list contains at least one item there must be at least one repeatable edit code in the format specification. An empty format specification ( ) can only be used if no list items are specified; in such a case one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. Where a repeat specification is used the appropriate item is repeated the required number of times.

Examples

```
      WRITE (6,90) NPAGE
90    FORMAT('1PAGE NUMBER ',I2,16X,'SALES REPORT, Cont.')
```

produces:

```
      PAGE NUMBER  SALES REPORT, Cont.
```

FORMAT *(cont.)*

The following example shows use of the tabulation specifier T:

```
      PRINT 25
25    FORMAT (T41,'COLUMN 2',T21,'COLUMN 1')
```

produces:

```
COLUMN 1    COLUMN 2
```

The following:

```
      DIMENSION A(6)
      DO 10 I = 1,6
10    A(I) = 25.
      TYPE 100,A
100   FORMAT(' ',F8.2,2PF8.2,F8.2) ! ' '
C                                  !gives single spacing
```

produces:

```
25.00  2500.00  2500.00  2500.00  2500.00  2500.00
```

Note that the effect of the scale factor continues until another scale factor is used.

FUNCTION

The function statement introduces a module; the statements that follow all apply to the function itself and are laid out in the same order as those of a PROGRAM module.

Syntax

```
[type] FUNCTION name ([argument [,argument]...])
```

type will explicitly apply a type to the function. If the function is not explicitly typed then the function type is taken from the initial letter and is dictated by the usual default.

name is the name of the function and must be unique among all the module names in the program. *name* must not clash with any local, COMMON or PARAMETER names.

argument is a symbolic name, starting with a letter and containing only letters and digits. An *argument* can be of type REAL, INTEGER, DOUBLE PRECISION, CHARACTER, LOGICAL or COMPLEX.

Description

The statements and names in the module apply only to the function, except for subroutine or function references and the names of COMMON blocks. The module must be terminated by an END statement.

A function produces a result; this allows a function reference to appear in an expression, where the result is assumed to replace the actual reference. The symbolic name of the function must appear as a variable in the module. The value of this variable, on exit from the function, is the result of the function. The result is undefined if the variable has not been defined.

The type of a FUNCTION refers to the type of its result.

FUNCTION *(cont.)*

Examples

```
      FUNCTION FRED(A,B,C)
      REAL X
        .
        .
      END

      FUNCTION EMPTY()  ! Note parentheses
        .
        .
      END

      PROGRAM FUNCALL
        .
        .
      SIDE=TOTAL(A,B,C)
        .
        .
      END

      FUNCTION TOTAL(X,Y,Z)
        .
        .
      END

      FUNCTION AORB(A,B)
      IF(A-B)1,2,3
    1 AORB = A
      RETURN
    2 AORB = B
      RETURN
    3 AORB = A + B
      RETURN
      END
```

GOTO (Computed)

The computed GOTO statement allows transfer of control to one of a list of labels according to the value of an expression.

Syntax

```
GOTO (list) [,] expression
```

list is a list of labels separated by commas.

expression selects the label from the list to which to transfer control. Thus a value of 1 implies the first label in the list, a value of 2 implies the second label and so on. An expression value outside the range will result in transfer of control to the statement following the computed GOTO statement.

Example

```
      READ *, A, B
      GO TO (50,60,70)A
      WRITE (*, 10) A, B
10    FORMAT (' ', I3, F10.4, 5X, 'A must be 1, 2
     +   or 3')
      STOP
50    X=A**B                ! Come here if A has the value 1
      GO TO 100
      60 X=(A*56)*(B/3)     ! Come here if A has the value 2
      GO TO 100
70    X=A*B                 ! Come here if A has the value 3
100   WRITE (*, 20) A, B, X
20    FORMAT (' ', I3, F10.4, 5X, F10.4)
```

GOTO (Unconditional)

The GOTO statement unconditionally transfers control to the statement with the label *label*. The statement label *label* must be declared within the code of the module containing the GOTO statement and must be unique within that module.

Syntax

```
GOTO label
```

label is a statement label

Example

```
      TOTAL=0.0
30    READ *, X
      IF (X.GE.0) THEN
          TOTAL=TOTAL+X
          GOTO 30
      END IF
```

GOTO (Assigned)

The assigned GOTO statement transfers control so that the statement identified by the statement label is executed next.

Syntax

```
GOTO integer-variable-name[[,] (list)]
```

integer-variable-name
must be defined with the value of a statement label of an executable statement within the same module. This type of definition can only be done by the ASSIGN statement.

list consists of one or more statement labels attached to executable statements in the same program unit. If a list of statement labels is present, the statement label assigned to the integer variable must be in that list. *list* does not affect statement execution.

Examples

The first example is equivalent to GOTO 100:

```
ASSIGN 100 TO INEXT
GO TO INEXT
```

The next example is equivalent to GOTO 50:

```
ASSIGN 50 TO K
GOTO K(50,90)
```

IF (Arithmetic)

The arithmetic IF statement transfers control to one of three labeled statements. The statement chosen depends upon the value of an arithmetic expression.

Syntax

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

Control transfers to *label-1*, *label-2* or *label-3* if the result of the evaluation of the *arithmetic-expression* is less than zero, equal to zero or greater than zero respectively.

Example

```
IF X 10, 20, 30
```

If X is less than zero then control is transferred to label 10.

If X equals zero then control is transferred to label 20.

If X is greater than zero then control is transferred to label 30.

IF (Block)

The block IF statement consists of a series of statements that are conditionally executed.

Syntax

```
IF logical expression THEN
statements
ELSE IF logical expression THEN
statements
ELSE
statements
ENDIF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an IF block.

Example

```
IF (I.GT.70) THEN
   M=1
ELSE IF (I.LT.5) THEN
   M=2
ELSE IF (I.LT.16) THEN
   M=3
ENDIF
IF (I.LT.15) THEN
   M = 4
ELSE
   M=5
ENDIF
```

IF (Logical)

The logical IF statement executes or does not execute a statement based on the value of a logical expression.

Syntax

```
IF (logical-expression) statement
```

logical-expression
is evaluated and if it is true the *statement* is executed. If it is false *statement* is not executed and control is passed to the next executable statement.

statement can be an assignment statement, a CALL statement or a GOTO statement.

Examples

```
IF (N .LE. 2) GOTO 27
IF (HEIGHT .GT. 1000.0 .OR. HEIGHT .LT. 0.0)
HEIGHT=1000.0
```

IMPLICIT

The IMPLICIT statement redefines the implied data type of symbolic names from their initial letter. Without the use of the IMPLICIT statement all names that begin with the letters I, J, K, L, M or N are assumed to be of type integer and all names beginning with any other letters are assumed to be real.

Syntax

```
IMPLICIT spec (a[,a]...) [,spec (a[,a]...)]
```

§ `IMPLICIT NONE`

spec is a data type specifier.

a is an alphabetic specification expressed either as *a* or *a1-a2*, specifying an alphabetically ordered range of letters.

Description

IMPLICIT statements must not be labeled.

Symbol names may begin with a dollar sign ($) or underscore (_) character, both of which are of type REAL by default. In an IMPLICIT statement, these characters may be used in the same manner as other characters, but they cannot be used in a range specification.

The IMPLICIT NONE statement specifies that all symbolic names must be explicitly declared, otherwise an error is reported. If IMPLICT NONE is used, no other IMPLICIT can be present.

Examples

```
IMPLICIT REAL (L,N)
IMPLICIT INTEGER (S,W-Z)
IMPLICIT INTEGER (A-D,$,_)
```

§ INCLUDE

The INCLUDE statement directs the compiler to start reading from another file.

Syntax

```
INCLUDE 'filename[/[NO]LIST]'
```

The INCLUDE statement may be nested to a depth of 20 and can appear anywhere within a program unit as long as Fortran's statement-ordering restrictions are not violated.

The qualifiers `/LIST` and `/NOLIST` can be used to control whether the include file is expanded in the listing file (if generated).

Note that there is no support for VAX/VMS *text libraries* or the *module_name* pathname qualifier that exists in the VAX/VMS version of the INCLUDE statement.

INQUIRE

An INQUIRE statement has two forms and is used to inquire about the current properties of a particular *file* or the current connections of a particular *unit*. INQUIRE may be executed before, while or after a file is connected to a unit.

Syntax

```
INQUIRE (file,list)
INQUIRE ([UNIT=] unit,list)
```

list of specifiers is as follows:

ACCESS= *acc*

acc — a character scalar memory reference which specifies the access method for file connection as either DIRECT or SEQUENTIAL; the default is SEQUENTIAL.

BLANK= *blnk*

blnk — a character scalar memory reference taking the value NULL or ZERO. NULL causes all blank characters in numeric formatted input fields to be ignored with the exception of an all blank field which has a value of zero. ZERO causes all blanks other than leading blanks to be treated as zeros. This specifier must only be used when a file is connected for formatted input/output.

DIRECT= *dir*

dir — a character scalar memory reference which takes the value YES if DIRECT is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if DIRECT is included.

ERR= *errs*

errs — an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by *errs*.

EXIST= *ex*

ex — a logical scalar memory reference which becomes .TRUE. if there is a file/unit with the specified name or .FALSE. otherwise.

INQUIRE *(cont.)*

FORM= *fm*

fm — a character scalar memory reference specifying whether the file is being connected for FORMATTED or UNFORMATTED output; the default is UNFORMATTED.

FORMATTED= *fmt*

fmt — a character scalar memory reference which takes the value YES if FORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if FORMATTED is included.

IOSTAT= *ios*

ios — an integer scalar memory reference which is the input/output specifier: if this is included in *list*, *ios* is defined with 0 if no error condition exists or a positive integer when there is an error.

NAME= *fn*

fn — a character scalar memory reference which is assigned the name of the file when the file has a name, otherwise it is undefined.

NAMED= *nmd*

nmd — a logical scalar memory reference which becomes .TRUE. if the file has a name, otherwise it becomes .FALSE..

NEXTREC= *nr*

nr — an integer scalar memory reference which is assigned the value n+1, where n is the number of the record read or written. It takes the value 1 if no records have been read or written. If the file is not connected, or its position is indeterminate, *nr* is undefined.

NUMBER= *num*

num — an integer scalar memory reference or integer array element assigned the value of the external unit number of the currently connected unit. It becomes undefined if no unit is connected.

OPENED= *od*

od — a logical scalar memory reference which becomes .TRUE. if the file/unit specified is connected as appropriate.

RECL= *rcl*

rcl — an integer scalar memory reference defining the record length in a file connected for direct access and is the number of characters when formatted input/output is specified. This specifier must only be given when a file is connected for direct access.

INQUIRE *(cont.)*

`SEQUENTIAL=` *seq*

seq a character scalar memory reference which is assigned the value `YES` if SEQUENTIAL is included in the set of allowed access methods, `NO` if SEQUENTIAL is not included and `UNKNOWN` if it cannot be determined whether or not SEQUENTIAL is included.

`UNFORMATTED=` *unf*

unf a character scalar memory reference which takes the value `YES` if UNFORMATTED is one of the allowed access methods for the file, `NO` if not and `UNKNOWN` if it is not known if UNFORMATTED is included.

Description

When an INQUIRE by file statement is executed the following specifiers will only be assigned values if the file name is acceptable.

nmd, *fn*, *seq*, *dir*, *fmt* and *unf*. *num* is defined, and *acc*, *fm*, *rcl*, *nr* and *blnk* may become defined only if *od* is defined as `.TRUE.`.

When an INQUIRE by unit statement is executed the specifiers *num*, *nmd*, *fn*, *acc*, *seq*, *dir*, *fm*, *fmt*, *unf*, *rcl*, *nr* and *blnk* are assigned values provided that the specified unit exists and a file is connected to that unit. Should an error condition occur during the execution of an `INQUIRE` statement all the specifiers except *ios* become undefined.

INTEGER

The INTEGER statement establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
INTEGER[*n] name[*n] [([lb:]ub[,[lb:]ub]...)] [/clist/]
            [, name[*n] [([lb:]ub[,[lb:]ub]...)] [/clist/]]...
```

n is an optional size specification.

name is the symbolic name of a variable or array.

lb:ub is a dimension declarator specifying the bounds for a dimension (the lower bound *lb* and the upper bound *ub*). *lb* and *ub* must be integers with *ub* greater than lb. The lower bound *lb* is optional; if it is not specified, it is taken to be 1.

clist is a list of constants that initialize the data, as in a DATA statement.

Description

Integer type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. INTEGER statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Example

```
INTEGER TIME, SECOND, STORE  (5,5)
```

INTRINSIC

An INTRINSIC statement identifies a symbolic name as an intrinsic function and allows it to be used as an actual argument.

Syntax

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

func is the name of an intrinsic function such as SIN, COS, etc.

Description

Do *not* use any of the following functions in INTRINSIC statements:

- type conversions:
  INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR
- lexical relationships:
  LGE, LGT, LLE, LLT
- values:
  MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1

When a specific name of an intrinsic function is used as an actual argument in a module it must appear in an INTRINSIC statement in that module. If the name used in an INTRINSIC statement is also the name of a generic intrinsic function, it retains its generic properties. A symbolic name can appear only once in all the INTRINSIC statements of a module and cannot be used in both an EXTERNAL and INTRINSIC statement in a module.

INTRINSIC *(cont.)*

Example

The following example illustrates the use of INTRINSIC and EXTERNAL:

```
EXTERNAL MYOWN
INTRINSIC SIN, COS
.
.
CALL TRIG (ANGLE,SIN,SINE)
.
CALL TRIG (ANGLE,MYOWN,COTANGENT)
.
CALL TRIG (ANGLE,COS,SINE)


SUBROUTINE TRIG (X,F,Y)
Y=F(X)
RETURN
END


FUNCTION MYOWN
MYOWN=COS(X)/SIN(X)
RETURN
END
```

In this example, when TRIG is called with a second argument of SIN or COS the function reference `F(X)` references the intrinsic functions SIN and COS; however when TRIG is called with MYOWN as the second argument `F(X)` references the user function MYOWN.

LOGICAL

The LOGICAL statement establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
LOGICAL[*n] name[*n] [([lb:]ub[,[lb:]ub]...)] [/clist/]
            [, name[*n] [([lb:]ub[,[lb:]ub]...)] [/clist/]]...
```

n is an optional size specification.

name is the symbolic name of a variable or array.

lb:ub is a dimension declarator specifying the bounds for a dimension (the lower bound *lb* and the upper bound *ub*). *lb* and *ub* must be integers with *ub* greater than lb. The lower bound *lb* is optional; if it is not specified, it is taken to be 1.

clist is a list of constants that initialize the data, as in a DATA statement.

Description

Integer type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Example

```
LOGICAL TIME, SECOND, STORE  (5,5)
```

§ MAP

The MAP statement initiates a map declaration within a union declaration.

Syntax

```
MAP
   field_declaration
   [field_declaration]
         ...
   [field_declaration]
END MAP
```

field_declaration
is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Description

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration.

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to what happens using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each "map" (union). Fortran field names must be unique within the same declaration nesting level of maps.

MAP *(cont.)*

Example

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the *recarr* array would be the size of *typetag* (4 bytes) plus the size of the largest MAP - the *employee* map (24 bytes).

```
STRUCTURE /account/
  INTEGER typetag              ! Tag to determine defined map.
  UNION
        MAP                              ! Structure for an employee
            CHARACTER*12  ssn            ! Social Security Number
            REAL*4        salary
            CHARACTER*8   empdate        ! Employment date
        END MAP
        MAP                              ! Structure for a customer
            INTEGER*4     acct_cust
            REAL*4        credit_amt
            CHARACTER*8   due_date
        END MAP
        MAP                              ! Structure for a supplier
            INTEGER*4     acct_supp
            REAL*4        debit_amt
            BYTE          num_items
            BYTE          items(12) ! Items supplied
        END MAP
  END UNION
END STRUCTURE

RECORD /account/ recarr(1000)
```

§ NAMELIST

The NAMELIST statement allows for the definition of namelist groups for namelist-directed I/O.

Syntax

NAMELIST */group-name/ namelist* [[,] */group-name/ namelist*]...

group-name is the name of the namelist group.

namelist is the list of variables in the namelist group.

Example

In the following example a name group PERS consists of a name, an account, and a value.

```
CHARACTER*12   NAME
INTEGER*$      ACCOUNT
REAL*4         VALUE
NAMELIST        /PERS/  NAME, ACCOUNT, VALUE
```

OPEN

The OPEN statement can be used to do the following: connect an existing file to a unit; create and connect a file to a unit; create a file that is preconnected; change certain specifiers of a connection between a file and a unit

Syntax

```
OPEN ( list )
```

list must contain exactly one unit specifier of the form:

```
[UNIT=] u
```

where the `UNIT=` is optional and the external unit specifier *u* is an integer.

In addition *list* may contain one of each of the following specifiers.

`ACCESS=` *acc*

acc is a character expression specifying the access method for file connection as either DIRECT or SEQUENTIAL - the default is SEQUENTIAL.

`BLANK=` *blnk*

blnk is a character expression which takes the value 'NULL' or 'ZERO': 'NULL ' causes all blank characters in numeric formatted input fields to be ignored with the exception of an all blank field which has a value of zero. 'ZERO' causes all blanks other than leading blanks to be treated as zeros. The default is 'NULL.' This specifier must only be used when a file is connected for formatted input/output.

`FORM=`*fm*

fm is a character expression specifying whether the file is being connected for 'FORMATTED' or 'UNFORMATTED' output respectively and is 'UNFORMATTED' by default.

`ERR=`*errs*

errs is an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by *errs*.

`FILE=` *fin*

fin is a character expression whose value is the file name to be connected to the specified unit.

`IOSTAT=` *ios*

ios is an input/output status specifier where *ios* is an integer scalar memory reference: if this is included in *list*, *ios* becomes defined with 'ZERO ' if no error condition exists or a positive integer when there is an error condition.

OPEN *(cont.)*

`RECL=` *rl*

rl is an integer expression defines the record length in a file connected for direct access and is the number of characters when formatted input/output is specified. This specifier must only be given when a file is connected for direct access.

`STATUS=` *sta*

sta is a character expression whose value can be: "NEW", "OLD", "SCRATCH" or "UNKNOWN". When "OLD" or "NEW" is specified a file specifier must be given. "SCRATCH" must not be used with a named file. The default is UNKNOWN.

Description

The record length, RECL=, must be specified if a file is connected for direct access and optionally one of each of the other specifiers may be used. RECL is ignored if the access method is sequential.

The unit specified must exist and once connected by an OPEN statement can be referenced in any module of the executable program. If a file is connected to a unit it cannot be connected to a different unit by the OPEN statement.

If a unit is connected to an existing file, execution of an OPEN statement for that file is allowed. Where FILE= is not specified the file to be connected is the same as the file currently connected. If the file specified for connection to the unit does not exist but is the same as a preconnected file, the properties specified by the OPEN statement become part of the connection. However, if the file specified is not the same as the preconnected file this has the same effect as the execution of a CLOSE statement without a STATUS= specifier immediately before the execution of the OPEN statement. When the file to be connected is the same as the file already connected only the BLANK= specifier may be different from the one currently defined.

Example

In the following example a new file, BOOK, is created and connected to unit 12 for direct formatted input/output with a record length of 98 characters. Numeric values will have blanks ignored and E1 will be assigned some positive value if an error condition exists when the OPEN statement is executed; execution will then continue with the statement labeled 20. If no error condition occurs, E1 is assigned the value zero (0) and execution continues with the next statement.

```
OPEN( 12, IOSTAT=E1, ERR=20, FILE='BOOK',
1BLANK='NULL', ACCESS='DIRECT', RECL=98,
1FORM='FORMATTED',STATUS='NEW')
```

OPEN *(cont.)*

Environment Variables

For an OPEN statement which does not contain the `FILE=` specifier, an environment variable may be used to specify the file to be connected to the unit. If the environment variable `FOR`*ddd* exists, where *ddd* is a 3 digit string whose value is the unit, the environment variable's value is the name of the file to be opened.

VAX/VMS Fortran

VAX/VMS introduces a number of extensions to the OPEN statement. Many of these relate only to the VMS file system and are not supported (e.g., KEYED access for indexed files). The following keywords for the OPEN statement have been added or augmented as shown below. Refer to *Programming in VAX Fortran* for additional details on these keywords.

ACCESS — The value of 'APPEND' is recognized and implies sequential access and positioning after the last record of the file. Opening a file with append access means that each appended record is written at the end of the file.

ASSOCIATEVARIABLE
: This new keyword specifies an INTEGER*4 integer scalar memory reference which is updated to the next sequential record number after each direct access I/O operation. Only for direct access mode.

DISPOSE and DISP
: These new keywords specify the disposition for the file after it is closed. 'KEEP' or 'SAVE' is the default on anything other than STATUS='SCRATCH' files. 'DELETE' indicates that the file is to be removed after it is closed. The PRINT and SUBMIT values are not supported.

NAME — This new keyword is a synonym for FILE.

READONLY — This new keyword specifies that an existing file can be read but prohibits writing to that file. The default is read/write.

RECL=*len* — The record length given is interpreted as number of words in a record if the environment variable *FTNOPT* is set to "*vaxio*". This simplifies the porting of VAX/VMS programs. The default is that *len* is given in number of bytes in a record.

TYPE — This keyword is a synonym for STATUS.

§ OPTIONS

The OPTIONS statement allows you to specify certain compiler command-line options for a particular program unit.

Syntax

```
OPTIONS /option [/option ...]
```

Table 3-1 shows what options are available for the OPTIONS statement.

Table 3-1. OPTIONS Statement

Option	Action Taken
CHECK=ALL	No effect (recognized, but ignored).
CHECK=[NO]OVERFLOW	No effect.
CHECK=[NO]BOUNDS	No effect.
CHECK=[NO]UNDERFLOW	No effect.
CHECK=NONE	No effect.
NOCHECK	No effect.
[NO]EXTEND_SOURCE	(Don't) enable the *-Mextend* switch.
[NO]F77	(Don't) enable the *-Mstandard* switch.
[NO]G_FLOATING	No effect.
[NO]I4	(Don't) enable the *-Mi4* switch.
[NO]RECURSIVE	(Don't) enable the *-Mrecursive* switch.
[NO]REENTRANT	(Don't) enable the *-Mreentrant* switch.
[NO]STANDARD	(Don't) enable the *-Mstandard* switch.

OPTIONS *(cont.)*

The following restrictions apply to the OPTIONS statement:

- The OPTIONS statement must be the first statement in a program unit; it must precede the PROGRAM, SUBROUTINE, FUNCTION, and BLOCKDATA statements.
- The options listed in the OPTIONS statement override values from the compiler command-line for the program unit immediately following the OPTIONS statement.
- Any abbreviated version of an *option* that is long enough to identify the option uniquely is a legal abbreviation for the *option*.
- Case is not significant, unless the *–Mupcase* switch is present on the command line. If it is, each *option* must be in lower case.

Example

```
OPTIONS /RECURSIVE/REENTRANT
```

This specifies that the program unit following the OPTIONS statement is to be compiled with recursive and reentrant options activated, regardless of the settings on the Fortran command line.

PARAMETER

The PARAMETER statement gives a symbolic name to a constant.

Syntax

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

expression is an arithmetic expression formed from constant or PARAMETER elements using the arithmetic operators + - * /. The usual precedence order can be changed by using parentheses. *expression* may include a previously defined PARAMETER.

Examples

```
PARAMETER ( PI = 3.142 )
PARAMETER ( INDEX = 1024 )
PARAMETER ( INDEX3 = INDEX * 3 )
```

The following VAX/VMS extensions to the PARAMETER statement are fully supported:

- Its list is not bounded with parentheses.
- The form of the constant (rather than the implicit or explicit typing of the symbolic name) determines the data type of the variable.

The form of the alternative PARAMETER statement is:

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

where *p* is a symbolic name and *c* is a constant, symbolic constant, or a compile time constant expression.

PAUSE

The PAUSE statement stops the program's execution.

Syntax

```
PAUSE [ display ] ]
```

display is a character constant or a string of decimal digits.

Description

The PAUSE statement stops the program's execution. The contents of *display* are displayed at your terminal when the PAUSE statement is executed. The program may be restarted later and execution will then continue with the statement following the PAUSE statement.

Example

The following PAUSE statement:

```
PAUSE 'Error Detected'
```

would cause program execution to be suspended, and the following would display at your terminal:

```
Error Detected
$
```

§ POINTER

The POINTER statement declares a scalar variable to be a *pointer* variable (of type INTEGER), and another variable to be its *pointer-based* variable.

Syntax

```
POINTER (p1, v1) [, (p2, v2) ...]
```

v1 and *v2* are *pointer-based* variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for *adjustable* dummy arrays regarding any variables which appear in the dimension declarators apply.

p1 and *p2* are the pointer variables corresponding to *v1* and *v2*. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is *dereferenced*).

The pointer-based variable does not have an address until its corresponding pointer is defined. The pointer *is defined* in one of the following ways:

- By assigning the value of the LOC function
- By assigning a value defined in terms of another pointer variable
- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

POINTER *(cont.)*

Example

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))

P = LOC(IC)
I = 0            ! IC gets 0

P = LOC(XC)
Q =  P + 20      ! same as LOC(XC(6))
X(1) = 0         ! XC(6) gets 0
ALLOCATE (X)     ! Q locates a dynamically
                 ! allocated memory area
```

Restrictions.

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is *assumed to be defined.*
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON,EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram. If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarator(s) are defined with an integer value at the time the SUBROUTINE or FUNCTION is called. For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is *assumed not to overlap* with another pointer-based variable.

PRINT

The PRINT statement is a data transfer output statement.

Syntax

```
PRINT format-identifier [, iolist]
```

or

§ `PRINT namelist-group`

format-identifier
a label of a format statement or a variable containing a format string.

iolist (output list) must either be one of the items in an input list or any other expression. However a character expression involving concatenation of an operand of variable length cannot be included in an *output list* unless the operand is the symbolic name of a constant.

namelist-group the name of a namelist group.

Description

When a PRINT statement is executed the following operations are carried out: data is transferred to the standard output device from the items specified in the output list and format specification.[1] The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

Example

In the following example, the PRINT statement writes one record to the standard output device. The record has four fields of character data.

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=',A,'JOB=',A)
```

1. If an asterisk (*) is used instead of a format identifier, the list-directed formatting rules apply.

PROGRAM

The PROGRAM statement specifies the entry point for the linked Fortran program.

Syntax

```
PROGRAM [name]
```

name is optional; if supplied it becomes the name of the program module and must not clash with any other names used in the program. If it is not supplied, a default name is used.

Description

The program statement specifies the entry point for the linked Fortran program. An END statement terminates the program.

Example

```
PROGRAM MYOWN
 REAL MEAN, TOTAL
 .
 CALL TRIG(A,B,C,MEAN)
 .
 END
```

READ

The READ statement is the data transfer input statement.

Syntax

```
      READ ([unit=] u, format-identifier [, control-information) [iolist]
      READ format-identifier [,iolist]
§     READ ([unit=] u, [NML=] namelist-group [, control-information])
```

format-identifier
label of a format statement or a variable containing a format string.

iolist (input list) must either be one of the items in an input list or any other expression.

Description

When a READ statement is executed the following operations are carried out: data is transferred from the standard input device to the items specified in the input and format specification.[1] The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

Example

```
      READ(2,110) I,J,K
110   FORMAT(I2, I4, I3)
```

1. If an asterisk (*) is used instead of a format identifier, the list-directed formatting rules apply.

REAL

The REAL statement establishes the data type of a variable by explicitly attaching the name of a variable to a data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
REAL[*n] name[*n] [([lb:]ub[,[lb:]ub]...)] [/clist/]
            [, name[*n] [([lb:]ub[,[lb:]ub]...)] [/clist/]]...
```

n is an optional size specification.

name is the symbolic name of a variable or array

lb:ub is a dimension declarator specifying the bounds for a dimension (the lower bound *lb* and the upper bound *ub*). *lb* and *ub* must be integers with *ub* greater than lb. The lower bound *lb* is optional; if it is not specified, it is taken to be 1.

clist is a list of constants that initialize the data, as in a DATA statement.

Description

The REAL type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Example

```
REAL KNOTS
```

§ RECORD

The RECORD statement defines a user-defined aggregate data item.

Syntax

```
RECORD /structure_name/record_namelist
       [,/structure_name/record_namelist]
                    ...
       [,/structure_name/record_namelist]
END RECORD
```

structure_name is the name of a previously declared structure.

record_namelist is a list of one or more variable or array names separated by commas.

Description

You create memory storage for a record by specifying a structure name in the RECORD statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, `recordname.fieldname`). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, INTEGER), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of COMMON, SAVE, NAMELIST, DATA and EQUIVALENCE statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

RECORD *(cont.)*

Example

```
STRUCTURE /person/    ! Declare a structure to define a person
   INTEGER ID
   LOGICAL LIVING
   CHARACTER*5 FIRST, LAST, MIDDLE
   INTEGER AGE
END STRUCTURE
   ! Define population to be an array where each element is of
   ! type person.  Also define a variable, me, of type person.
RECORD /PERSON/ POPULATION(2), ME
      ...
ME.AGE = 34             ! Assign values for the variable me to
ME.LIVING = .TRUE.      ! some of the fields.
ME.FIRST = 'Steve'
ME.ID = 542124822
      ...
POPULATION(1).LAST = 'Jones' ! Assign the "LAST" field of
                             ! element 1 of array population.
POPULATION(2) = ME           ! Assign all the values of record
                             ! "ME" to the record population(2)
```

RETURN

The RETURN statement causes a return to the statement following a CALL when used in a subroutine, and to within the relevant arithmetic expression when used in a function.

Syntax

```
RETURN
```

Alternate RETURN Statement

The alternate RETURN statement takes the following form:

```
RETURN (i)
```

i — *i* is an integer expression . The value of *i* specifies that the *i*th alternate return in the subroutine argument list is to be taken (see the example that follows).

Example

```
      SUBROUTINE FIX (A,B,*,*,C)

40    IF (T) 50, 60, 70
50    RETURN
60    RETURN 1
70    RETURN 2
      END


      PROGRAM FIXIT
      CALL FIX(X, Y, *100, *200, S)
      WRITE(*,5) X, S   ! Come here if (T) < 0
      STOP
100   WRITE(*, 10) X, Y   ! Come here if (T) = 0
      STOP
200   WRITE(*,20) Y, S   ! Come here if (T) > 0
```

REWIND

The REWIND statement positions the file at its beginning. The statement has no effect if the file is already positioned at the start or if the file is connected but does not exist.

Syntax

```
REWIND  unit
REWIND  (unit, list)
```

unit is an integer value which is the external unit.

list contains the optional specifiers as follows:

IOSTAT= *ios*

ios an integer scalar memory reference which is the input/output specifier: if this is included in *list*, *ios* becomes defined with zero if no error condition exists or with a positive integer if there is an error condition.

ERR= *errs*

errs an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by *errs*.

Examples

```
REWIND 5
REWIND(2, ERR=30)
REWIND(3, IOSTAT=IOERR)
```

SAVE

The SAVE statement retains the definition status of an entity after a RETURN or END statement in a subroutine or function has been executed.

Syntax

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

v is the name of an array, variable, or common block.

Description

Using a *common-block* name, preceded and followed by a slash, ensures that all entities within that COMMON block are saved. SAVE may be used without a list, in which case all the allowable entities within the module are saved (this has the same effect as using the *–Msave* command-line option). Dummy arguments, names of procedures and names of entities within a common block may not be specified in a SAVE statement. Use of the SAVE statement with local variables ensures the values of the local variables are retained for the next invocation of the SUBROUTINE or FUNCTION. Within a main program the SAVE statement is optional and has no effect.

When a RETURN or END is executed within a subroutine or function, all entities become undefined with the exception of:

- Entities specified by a SAVE statement
- Entities in blank common or named common
- Entities initially defined which have not been changed in any way

Example

```
PROGRAM SAFE
  .
  CALL KEEP
  .
  SUBROUTINE KEEP
  COMMON /LIST/ TOP, MIDDLE
  INTEGER LOCAL1.
  .
  SAVE /LIST/, LOCAL1
```

STOP

The STOP statement stops the program's execution and precludes any further execution of the program.

Syntax

```
STOP [ display ] ]
```

display is a character constant or a string of decimal digits.

Description

The STOP statement stops the program's execution. The contents of *display* are displayed at your terminal when the STOP statement is executed.

Example

The following are examples of valid STOP statements:

```
STOP

STOP 'End of program'

STOP 99
```

§ STRUCTURE

The STRUCTURE statement defines an aggregate data type.

Syntax

```
STRUCTURE [/structure_name/][field_namelist]
      field_declaration
      [field_declaration]
            ...
      [field_declaration]
END STRUCTURE
```

structure_name is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.

field_namelist is a list of fields having the structure of the associated structure declaration. A *field_namelist* is allowed only in nested structure declarations.

field_declaration
can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

Description

Fields within structures conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict. Also, because records use periods to separate fields, it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Fields in a structure are aligned as required by hardware and a structure's storage requirements are therefore machine-dependent. Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.

NOTE

Field alignment and padding are not performed by VAX/VMS Fortran.

STRUCTURE *(cont.)*

Data initialization can occur for the individual fields.

The UNION and MAP statements are also supported.

Example

The following example shows the definition of two structure types, named `str1` and `str2`, and the declaration of one record object of type `str2`, named `rec1`. Note that `str2` has one field, named `r1`, which is itself a structure of type `str1`.

```
STRUCTURE /STR1/
    CHARACTER*1 CHR
    INTEGER*4 INT
END STRUCTURE

STRUCTURE /STR2/
    RECORD /STR1/ R1
    STRUCTURE /STR3/ RN1
        CHARACTER*1 CHR
        INTEGER*4 INT
    END STRUCTURE
    RECORD /STR1/ R2
END STRUCTURE

RECORD /STR2/ REC1

REC1.RN1.CHR = 'A'
REC1.RN1.INT = 100
```

SUBROUTINE

The SUBROUTINE statement introduces a module. The statements that follow should be laid out in the same order as a PROGRAM module.

Syntax

```
SUBROUTINE name [(argument[,argument...])]
```

name is the name of the subroutine being declared and must be unique amongst all the subroutine and function names in the program. *name* should not clash with any local, COMMON, PARAMETER or ENTRY names.

argument is a symbolic name, starting with a letter and containing only letters and digits. The type of argument can be REAL, INTEGER, DOUBLE PRECISION, CHARACTER, LOGICAL or COMPLEX.

Description

The SUBROUTINE module must be terminated by an END statement. The statements and names in the module only apply to the subroutine except for subroutine or function references and the names of COMMON blocks. Dummy arguments may be specified as * which indicates that the SUBROUTINE contains alternate returns.

Example

```
SUBROUTINE STAR(A,B,C,*,*)
```

Note the dummy arguments represented by the two *s.

```
IF (ANY) THEN
    A=45
    B=36.33
    C=0
    RETURN 1
ELSE
    C=100
    RETURN 2
END IF
END
```

SUBROUTINE *(cont.)*

```
      PROGRAM SHOWME
        .
        .
      CALL STAR(R,S,T,*30,*40)
        .
        .
30    WRITE(*,10) R,S    ! Come here if RETURN 1
        .
        .
40    WRITE(*,20) T      ! Come here if RETURN 2
```

THEN

The THEN statement is part of a block IF statement and surrounds a series of statements that are conditionally executed. See IF (block) for details.

§ TYPE

The TYPE statement has the same syntax and effect as the PRINT statement. Refer to the PRINT entry for a description of its syntax and function.

§ UNION

A union declaration is a multistatement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration.

Syntax

```
UNION
   map_declaration
   [map_declaration]
         ...
   [map_declaration]
END UNION
```

The format of the *map_declaration* is as follows:

```
MAP
   field_declaration
   [field_declaration]
         ...
   [field_declaration]
END MAP
```

field_declaration
is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Description

Union declarations are used when one wants to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. The format of a UNION statement is as follows:

UNION *(cont.)*

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to what happens using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each "map" (union). Fortran field names must be unique within the same declaration nesting level of maps.

Example

The following is an example of RECORD, STRUCTURE and UNION usage. The union elements are of different sizes (5 bytes for `chr`, 4 bytes each for `short` and `int`), so the size of the union is the size of the largest element (5 bytes) plus any padding that may be required.

```
      STRUCTURE /STR1/
         UNION
             MAP
                 CHARACTER*1 CHR(4)
             ENDMAP
             MAP
                 INTEGER*2   SHORT(2)
             ENDMAP
             MAP
                 INTEGER*4   INT
             ENDMAP
         END UNION
      END STRUCTURE

      RECORD /STR1/ REC1

      INTEGER TK1/0/

CC
      REC1.CHR(1) = 'A'
      REC1.CHR(2) = 'B'
      REC1.CHR(3) = 'C'
      REC1.CHR(4) = 'D'

      IF (REC1.SHORT(1) .NE. 'AB')      TK1 = TK1 + 1
      IF (REC1.SHORT(2) .NE. 'CD')      TK1 = TK1 + 2
      IF (REC1.INT .NE. 'ABCD')         TK1 = TK1 + 4
```

§ VOLATILE

The VOLATILE statement inhibits all optimizations on the variables, arrays and common blocks that it identifies.

Syntax

```
VOLATILE /nitem/ [, /nitem/ ...]
```

nitem is the name of a variable, an array, or a common block.

Description

If *nitem* names a common block, all members of the block are volatile. The volatile attribute of a variable is inherited by any direct or indirect equivalences, as shown in the example.

Example

```
COMMON /COM/ C1, C2
VOLATILE /COM/, /DIR/    ! /COM/ and DIR are volatile
EQUIVALENCE (DIR, X)     ! X is volatile
EQUIVALENCE (X, Y)       ! Y is volatile
```

WRITE

The WRITE statement is a data transfer output statement.

Syntax

```
WRITE ([unit=] u, format-identifier [, control-information) [iolist])
```

or

§ `WRITE ([unit=] u, [NML=] namelist-group [, control-information])`

format-identifier
: a label of a format statement or a variable containing a format string.

iolist
: (output list) must either be one of the items in an input list or any other expression. However a character expression involving concatenation of an operand of variable length cannot be included in an *output list* unless the operand is the symbolic name of a constant.

Description

When a WRITE statement is executed the following operations are carried out: data is transferred to the standard output device from the items specified in the output list and format specification. The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

Example

```
      WRITE (6,90) NPAGE
90    FORMAT('1PAGE NUMBER ',I2,16X,'SALES REPORT, Cont.')
```

Input and Output 4

Input, output, and format statements provide the means for transferring data to or from files. Data is transferred as records to or from files. A *record* is a sequence of data which may be values or characters and a *file* is a sequence of such records. A file may be internal, that is, held in memory, or external such as those held on disk. To access an external file a formal connection must be made between a *unit,* for example a disk file, and the required file. An external unit must be identified either by a positive integer expression, the value of which indicates a unit, or by an asterisk (`*`) which identifies the standard input or output device.

This chapter describes the types of input and output available and provides examples of input, output and format statements. There are four types of input/output you can use to transfer data to or from files: unformatted, formatted, list directed, and namelist.

- *unformatted* data is transferred between the item(s) in the input/output list (*iolist*) and the current record in the file. Exactly one record may be read or written.
- *formatted* data is edited to conform to a format specification, and the edited data is transferred between the item or items in the *iolist*, and the file. One or more records may be read or written.
- *list directed* input/output is an abbreviated form of formatted input/output that does not require the use of a format specification. Depending on the type of the data item or data items in the *iolist*, data is transferred to or from the file, using a default, and not necessarily accurate format specification.
- *namelist* input/output is a special type of formatted data transfer; data is transferred between a named group (namelist group) of data items and one or more records in a file.

File Access Methods

You can access files using one of two methods, *sequential access*, or *direct access* (random access). The access method is determined by the specifiers supplied when the file is opened using the OPEN statement. Sequential access files are accessed one after the other, and are written in the same manner. Direct access files are accessed by specifying a record number for input, and by writing to the currently specified record on output.

Files may contain one of two types of records, fixed length records or variable length records. To specify the size of the fixed length records in a file, use the RECL specifier with the OPEN statement. RECL sets the record length in bytes. RECL can only be used when access is direct.

A record in a variable length formatted file is terminated with a new line. A record in a variable length unformatted file is preceded and followed by a word indicating the length of the record.

Standard Preconnected Units

Certain input and output units are predefined, depending on the value of compiler options. The *if77* option *-Mdefaultunit* tells the compiler to treat "*" as a synonym for standard input for reading and standard output for writing. When the option is set to *–Mnodefaultunit*, then the compiler treats "*" as a synonym for unit 5 on input and unit 6 on output.

Opening and Closing Files

The OPEN statement establishes a connection to a file. OPEN allows you to do any of the following

- Connect an existing file to a unit.
- Create and connect a file to a unit.
- Create a file that is preconnected.
- Establish the access method and record format for a connection.

`OPEN` has the form:

```
OPEN (list)
```

where *list* contains a unit specifier of the form:

```
[UNIT]= u
```

where *u*, an integer, is the external unit specifier.

In addition *list* may contain one of each of the specifiers shown in Table 4-1.

Table 4-1. OPEN Specifiers

Specifiers	Description
IOSTAT= *ios*	An input/output status specifier where *ios* is an integer scalar memory reference. If this is included in *list*, *ios* becomes defined with 0 if no error exists or a positive integer when there is an error condition.[1]
ERR= *errs*	An error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs, execution continues with the statement specified by *errs*.
FILE= *fin*	A file specifier, where *fin* is a character string defining the file name to be connected to the specified unit.
STATUS= *sta*	A file status specifier, where *sta* is a character expression: it can be NEW, OLD, SCRATCH or UNKNOWN. When OLD or NEW is specified a file specifier must be given. SCRATCH must not be used with a named file. The default is UNKNOWN.
ACCESS= *acc*	An access specifier, where *acc* is a character string specifying the access method for file connection as DIRECT (random access) or SEQUENTIAL. The default is SEQUENTIAL.
FORM= *fm*	A format specifier, where *fm* is a character string specifying whether the file is being connected for FORMATTED or UNFORMATTED output respectively. The default is FORMATTED.
RECL= *rl*	A record length specifier, where *rl* is an integer which defines the record length in a file connected for direct access and is the number of characters when formatted input/output is specified. This specifier must only be given when a file is connected for direct access.
BLANK= *blnk*	A character specifier, where *blnk* is a character string which takes the value NULL or ZERO: NULL causes all blank characters in numeric formatted input fields to be ignored with the exception of an all blank field which has a value of zero. ZERO causes all blanks other than leading blanks to be treated as zeros. The default is NULL. This specifier must only be used when a file is connected for formatted input/output.

1. If IOSTAT and ERR are not present, a program will terminate if an error occurs.

Direct Access Files

If a file is connected for direct access using OPEN with ACCESS='DIRECT', the record length must be specified using RECL=, and optionally one of each of the other specifiers may be used.

Any file opened for direct access must be via fixed length records.

In the following example a new file, `book.dat`, is created and connected to unit `12` for direct formatted input/output with a record length of `98` characters. Numeric values will have blanks ignored and the variable *E1* will be assigned some positive value if an error condition exists when the OPEN statement is executed; execution will then continue with the statement labeled `20`. If no error condition exists, *E1* is assigned the value `0` and execution continues with the statement following the OPEN statement.

```
 OPEN(12,IOSTAT=E1,ERR=20,FILE='book.dat',BLANK='NULL',
+ACCESS='DIRECT',RECL=98,FORM='FORMATTED',STATUS='NEW')
```

Closing a File

Close a unit by specifying the CLOSE statement from within any module. If the unit specified does not exist or has no file connected to it, the CLOSE statement has no effect.

Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. An implicit CLOSE is executed when a program stops.

The CLOSE statement terminates the connection of the specified file to a unit.

```
CLOSE ([UNIT=] u [,IOSTAT=ios] [,ERR= errs ]
     [,STATUS= sta] [,DISPOSE= sta] [,DISP= sta])
```

Close takes the status values IOSTAT, ERR, and STATUS, similar to those described in Table 4-1. In addition, CLOSE allows the DISPOSE or DISP specifier which can take a status value *sta* which is a character string, where case is insignificant, specifying the file status (the same keywords are used for the DISP and DISPOSE status). Status can be KEEP or DELETE. KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement, conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

In the following example the file on unit 6 is closed and deleted.

```
CLOSE(UNIT=6,STATUS='DELETE')
```

Unformatted Data Transfer

Unformatted data transfer allows data to be transferred between the current record and the items specified in an input/output list. Use OPEN to open a file for unformatted output:

```
OPEN (2, FILE='new.dat', FORM='UNFORMATTED')
```

The unit specified must be an external unit.

After data is transferred, the file is positioned after the last record read or written, if there is no error condition or end-of-file condition set. Unformatted data transfer cannot be carried out if the file is connected for formatted input/output.

The following example shows an unformatted input statement:

```
READ (2, ERR=50) A, B
```

On output to a file connected for direct access, the output list must not specify more values than can fit into a record. If the values specified do not fill the record the rest of the record is undefined.

On input the following conditions must pertain; the file must be positioned so that the record read is either:

- An unformatted record or an endfile record.
- The number of values required by the input list in the input statement must be less than or equal to the number of values in the record being read. The type of each value in the record must agree with that of the corresponding entity in the input list. However one complex value may correspond to two real list entities or vice versa. If the input list item is of type CHARACTER, its length must be the same as that of the character value.
- In the event of an error condition, the position of the file is indeterminate.

Formatted Data Transfer

During formatted data transfer, data is edited to conform to a format specification, and the edited data is transferred between the items specified in the input or output statement's *iolist* and the file; the current record and, possibly, additional records are read or written. On input the file must be positioned so that the record read is either a formatted record or an endfile record. Formatted data transfer is prohibited if the file is connected for unformatted input/output.

For variable length record formatted input, each newline character is interpreted as a record separator. On output, the I/O system writes a newline at the end of each record. If a program writes a newline itself, the single record containing the newline will appear as two records when read or backspaced over. The maximum allowed length of a record in a variable length record formatted file is 2000 characters.

Implied DO List Input Output List

An implied DO list takes the form

```
(iolist,do-var=var1,var2,var3)
```

where the items in *iolist* are either items permissible in an input/output list or another implied DO list,. The value *do-var* is an INTEGER, REAL or DOUBLE PRECISION variable and *var1*, *var2* and *var3* are arithmetic expressions of type INTEGER, REAL or DOUBLE PRECISION. Generally *do-var*, *var1*, *var2* and *var3* are of type INTEGER. Should *iolist* occur in an input statement, the *do-var* cannot be used as an item in *iolist*. If *var3* and the preceding comma are omitted the increment takes the value 1. The list items are specified once for each iteration of the DO loop with the DO-variable being substituted as appropriate.

```
REAL C(6),D(6)
DATA OXO,(C(I),I=7,9),TEMP,(D(J),J=1,2)/4*0.0,3*10.0/
```

In the above example `OXO`, `C(7)`, `C(8)` and `C(9)` are set to `0.0` with `TEMP`, `D(1)` and `D(2)` being set to `10.0`. In the next example:

```
READ *,A,B,(R(I),I=1,4),S
```

has the same effect as

```
READ *,A,B,R(1),R(2),R(3),R(4),S
```

Format Specifications

Format requirements may be given either in an explicit FORMAT statement or alternatively, as fields within an input/output statement (as values in character variables, arrays or other character expressions within the input/output statement).

When a format identifier in a formatted input/output statement is a character array name or other character expression, the leftmost characters must be defined with character data that constitute a format specification when the statement is executed. A character format specification is enclosed in parentheses. Blanks may precede the left parenthesis. Character data may follow the right-hand parenthesis and has no effect on the format specification. When a character array name is used as a format identifier, the length of the format specification can exceed the length of the first element of the array; a character array format specification is considered to be an ordered concatenation of all the array elements. When a character array element is used as a format identifier the length must not exceed that of the element used.

The FORMAT statement has the form:

```
FORMAT (list-of-format-requirements)
```

The list of format requirements can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant which defines the number of repeats.
- Non-repeatable editor commands.
- A format specification list enclosed in parentheses, optionally preceded by an integer constant which defines the number of repeats.

Each action of format control depends on a FORMAT specified edit code and the next item in the input/output list used. If an input/output list contains at least one item there must be at least one repeatable edit code in the format specification. An empty format specification FORMAT( ) can only be used if no list items are specified – in such a case one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. When a repeat specification is used, the appropriate item is repeated the required number of times.

Each repeatable edit code has a corresponding item in the *iolist*; however when a list item is of type complex two edit codes of F, E, D or G are required. The edit codes P, X, T, TL, TR, S, SP, SS, H, BN, BZ, /,: and apostrophe act directly on the record and have no corresponding item in the input/output list.

The file is positioned after the last character read or written when the edit codes I, F, E, D, G, L, A, H or apostrophe are processed. If the specified unit is a printer then the first character of the record is used to control the vertical spacing as shown in Table 4-2:

Table 4-2. Format Character Controls for a Printer

Character	Vertical Spacing
Blank	One line
0	Two lines
1	To first line on next page
+	No advance

A Format Control – Character Data

The A specifier transfers characters. The A can optionally be followed by a field width *w*. When *w* is not specified, the width is determined by the size of the data item.

On output, if *l* is the length of the character item and *w* is the field width, then the following rules apply:

If $w > l$: $w - l$ blanks before the character are included in the transfer.

If $w < l$: the leftmost *w* characters are transferred..

On input, if *l* is the length of the character item and *w* is the field width, then the following rules apply:

If $w > l$: the rightmost *l* characters from the input field are transferred.

If $w < l$: the leftmost *w* characters from the input field are transferred, followed by $l - w$ blanks.

You can also use the A format specifier to process data types other than CHARACTER. For types other than CHARACTER, the number of characters supplied for input/output will equal the size in bytes of the data allocated to the data type. For example, an INTEGER*4 value is represented with 4 characters and a LOGICAL*2 is represented with 2 characters.

The following shows a simple example that reads two CHARACTER arrays from the file *data.src*:

```
      CHARACTER STR1*8, STR2*12
      OPEN(2, FILE='data.src')
      READ(2, 10) STR1, STR2
10    FORMAT ( A8, A12 )
```

D Format Control – Real Double Precision Data with Exponent

The D specifier transfers real values for double precision data with a representation for an exponent. The form of the D specifier is:

```
Dw.d
```

where *w* is the field width and *d* the number of digits in the fractional part.

For input, the same conditions apply as for the F specifier which is described later.

For output the scale factor k controls the decimal normalization. The scale factor k is the current scale factor specified by the most recent P format control; if one hasn't been specified, the default is zero (0). If $-d < k \leq 0$, the output file contains leading zeros and d-|k| significant digits after the decimal point. If $0 < k < d+2$ there are exactly |k| significant digits to the left of the decimal point and d-k+1 significant digits to the right of the decimal point. Other values of *k* are not allowed.

For example:

```
      DOUBLE PRECISION VAL1
      VAL1 = 141.8835
      WRITE( *, 20) VAL1
20    FORMAT ( D10.4 )
```

produces the following:

```
0.1418D+03
```

E Format Control – Real Single Precision Data with Exponent

The E specifier transfers real values for single precision data with an exponent. The E format specifier has two basic forms:

```
Ew.d
Ew.dEe
```

w is the field width, *d* the number of digits in the fractional part and *e* the number of digits to be printed in the exponent part.

For input the same conditions apply as for F editing. For output the scale factor controls the decimal normalization as in D above.

F Format Control - Real Single Precision Data

The F specifier transfers real values. The form of the F specifier is:

```
Fw.d
```

w is the field width and *d* is the number of digits in the fractional part.

On input if the field does not contain a decimal digit or an exponent, right-hand *d* digits, with leading zeros, are interpreted as being the fractional part.

On output a leading zero is only produced to the left of the decimal point if the value is less than one.

G Format Control – Real Data

The G format specifier has two basic forms:

```
Gw.d
Gw.dEe
```

The specifier transfers real values; it acts like the F format control on input and depending on the value's magnitude, like E or F on output. The magnitude of the data determines the output format. For details on the actual format used, based on the magnitude, refer to Section 13.5.9.2.3 "G Editing", in the *ANSI FORTRAN Standard.*

I Format Control – Integer Data

The I format specifier transfers integer values. The I format specifier has two basic forms:

```
Iw
Iw.m
```

where *w* is the field width and *m* is the minimum field width on output, including leading zeros. If present, *m* must not exceed width w.

On input, the external field to be input must contain (unsigned) decimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the I field descriptor transfers the decimal values of the corresponding I/O list element, right-justified, to an external field that is *w* characters long. If the value to be transmitted does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks. If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary. Note that if *m* is zero, and the internal representation is zero, the external field is blank-filled.

L Format Control – Logical Data

```
Lw
```

The L format control transfers logical data of field width *w*. On input the list item will become defined with a logical value; the field consists of optional blanks, followed by an optional decimal point followed by T or F. Also, the values .TRUE. or .FALSE. may appear in the input field.

The output field consists of *w*-1 blanks followed by T or F as appropriate.

Quote Format Control

Quote editing prints a character constant. The format specifier writes the characters enclosed between the quotes and cannot be used on input. The field width is that of the characters contained within quotes (you can also use apostrophes to enclose the character constant).

To write an apostrophe (or quote) use two consecutive apostrophes (or quotes).

For example:

```
      WRITE ( *, 101)
  101 FORMAT ( 'Print an apostrophe '' and end.')
```

Produces:

```
      Print an apostrophe ' and end.
```

Similarly, you can use quotes, for example:

```
      WRITE ( *, 102)
  102 FORMAT ( "Print a line with a "" and end.")
```

Produces:

```
  Print a line with a " and end.
```

BN and BZ Format Control – Blank Control

The BN and BZ formats control blank spacing. BN causes all embedded blanks except leading blanks in numeric input to be ignored, which has the effect of right justifying the remainder of the field. Note that a field of all blanks has the value zero. Only input statements and I, F, E, D and G editing are affected.

BZ causes all blanks except leading blanks in numeric input to be replaced by zeros. Only input statements and I, F, E, D and G editing are affected.

H Format Control – Hollerith Control

The H format control writes the *n* characters following the H in the format specification and cannot be used on input.

The basic form of this format specification is:

```
  nHc1c2...cn...
```

where *n* is the number of characters to print and c1 through c*n* are the characters to print.

O and Z Format Control – Octal and Hexadecimal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with any data type. They have the form:

Ow[. *m*] and *Zw*[. *m*]

where *w* specifies the field width and *m* indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is *w* characters long. If the value to be transmitted does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks. If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary. Note that if *m* is zero, and the internal representation is zero, the external field is blank-filled.

P Format Specifier – Scale Control

*k*P

The P format specifier is the scale factor format which is applied as follows.

- With F, E, D and G editing on input and F editing on output, the external number equals the internal number multiplied by 10**k. If there is an exponent in the field on input editing with F, E, D and G the scale factor has no effect.
- On output with E and D editing the basic real constant part of the number is multiplied by 10**k and the exponent reduced by k , and with G editing the effect of the scale factor is suspended unless the size of the datum to be edited is outside the range permitted for F editing. If E editing is required the scale factor has the same effect as with E output editing.

The following is an example using a scale factor.

```
      DIMENSION A(6)
      DO 10 I = 1,6
10    A(I) = 25.
      TYPE 100,A
100   FORMAT(' ',F8.2,2PF8.2,F8.2)
```

produces:

```
25.00 2500.00  2500.00  2500.00  2500.00  2500.00
```

Note that the effect of the scale factor continues until another scale factor is used.

Q Format Control - Quantity

The Q edit descriptor calculates the number of characters remaining in the input record and stores that value in the next I/O list item. On output, the Q descriptor skips the next I/O item.

S Format Control – Sign Control

The S format specifier restores the default processing for writing a plus; the default is SS processing.

SP forces the processor to write a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

SS stops the processor from writing a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

T , TL, TR, and X Format Controls – Spaces and Tab Controls

The T specifier controls which portion of a record a *iolist* value is read from or written to a file. The general form is as follows:

```
Tn
```

this specifies that the *n*th value is to be written to or from a record.

The TL form specifies the relative position to the left of the data to be read or written.

```
TLn
```

This specifies that the *n*th character to the left of the current position is to be written to or from the record. If the current position is less than or equal to *n* the transmission will begin at position one of the record.

The TR form specifies the relative position to the right of the data to be read or written.

```
TRn
```

Specifies that the *n*th character to the right of the current position is to be written to or from the record.

The X control specifies a number of characters to skip forward.

```
nX
```

Specifies that the next character to be written to or from is *n* characters forward from the current position.

The following example uses the X format specifier.

```
      NPAGE = 19
      WRITE ( 6, 90) NPAGE
   90 FORMAT('1PAGE NUMBER ,I2, 16X, 'SALES REPORT, Cont.')
```

produces:

```
PAGE NUMBER 19                    SALES REPORT, Cont.
```

The following example shows use of the T format specifier.

```
      PRINT 25
   25 FORMAT (T41,'COLUMN 2',T21,'COLUMN 1')
```

produces:

```
            COLUMN 1     COLUMN 2
```

Slash Format Control – End of Record

The slash (/) control indicates the end of data transfer on the current record.

On input from a file connected for sequential access the rest of the current record is skipped and the file positioned at the start of the next record.

On output a new record is created which becomes the last and current record. For an internal file, connected for direct access the record is filled with blank characters. If a direct access file, the record number is increased by one and the file is positioned at the start of the record.

The : Format Specifier – Format Termination

The (:) control terminates format control if there are no more items in the input/output list. It has no effect if there are any items left in the list.

$ Format Control

The $ field descriptor allows the programmer to control carriage control conventions on output. It is ignored on input. For example, on terminal output, it can be used for prompting.

The form of the $ field descriptor is:

```
$
```

Variable Format Expressions <expr>

Variable format expressions are supported. They provide a means for substituting run-time expressions for the field width, other parameters for the field and edit descriptors in a FORMAT statement (except for the H field descriptor and repeat counts).

Variable format expressions are enclosed in "<" and ">" and are evaluated each time they are encountered in the scan of a format. If the value of a variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

List-Directed Formatting

List-directed formatting is an abbreviated form of input/output that does not require the use of a format specification. The type of the data is used to determine how a value is read/written. On output it will not always be accurate enough for certain ranges of values. The characters in a list-directed record constitute a sequence of values which cannot contain embedded blanks except those permitted within a character string. To use list-directed input/output formatting, specify a * for the list of format requirements. For example, the following example uses list-directed output:

```
READ( 1, * ) VAL1, VAL2
```

List-Directed Input

The form of the value being input must be acceptable for the type of item in the *iolist*. Blanks must not be used as zeros nor be embedded in constants except in a character constant or within a type complex form contained in parentheses.

Table 4-3. List Directed Input Values

Input List Type	Form
Integer	A numeric input field.
Real	A numeric input field suitable for F editing with no fractional part unless a decimal point is used.
Double precision	Same as for real.
Complex	An ordered pair of numbers contained within parentheses as shown (real part, imaginary part).
Logical	A logical field without any slashes or commas.
Character	A non-empty character string within apostrophes. A character constant can be continued on as many records as required. Blanks, slashes and commas can be used.

A null value has no effect on the definition status of the corresponding *iolist* item. A null value cannot represent just one part of a complex constant but may represent the entire complex constant. A slash encountered as a value separator stops the execution of that input statement after the assignment of the previous value. If there are further items in the list they are treated as if they are null values.

Commas may be used to separate the input values. If there are consecutive commas, or the if the first non-blank character of a record is a comma, the input value is a null value. Input values may also be repeated.

In the following example of list-directed formatting, assume that

```
A= -1.5
K= 125
```

and all other variables are undefined. When the statement below reads in the list from the input file:

```
READ * I, J, X, Y, Z, A, C, K
```

where the file contains the following record:

```
10,-14,25.2,-76,313,,29/
```

The variables are assigned the following values by the list-directed input/output mechanism:

```
I=10
J=-14
X=25.2
Y=-76.0
Z=313.0
A=-1.5
C=29
K=125.
```

Note that the value for A does not change because the input record is null (consecutive commas). No input is read for K, so it assumes null and K retains it previous value (the / terminates the input).

List-Directed Output

The data type of each item appearing in the *iolist* is formatted according to the rules in Table 4-4.

Table 4-4. Default List Directed Output Formatting

Data Type	Default Formatting
BYTE	I5
INTEGER*2	I7
INTEGER*4	I12
LOGICAL*1	I5 (L2)
LOGICAL*2	L2
LOGICAL*4	L2
REAL*4	G15.7e2
REAL*8	G25.16e3
COMPLEX*8	(G15.7e2, G15.7e2)
COMPLEX*16	(G25.16e3, G25.16e3)
CHAR *n	A*n*

The length of a record is less than 80 characters; if the output of an item would cause the length to exceed 80 characters, a new record is created.

Notes

1. New records may begin as necessary with the exception of character and complex constants.
2. Logical output constants are T for true and F for false.
3. Complex constants are contained within parentheses with the real and imaginary parts separated by a comma.
4. Character constants are not delimited by apostrophes and have each internal apostrophe (if any are present) represented externally by one apostrophe
5. Each output record begins with a blank character to provide carriage control when the record is printed.
6. A typeless value output with list-directed I/O is output in hexadecimal form by default. There is no other octal or hexadecimal capability with list-directed I/O.

Commas in External Field

Use of the comma in an external field eliminates the need to "count spaces" to have data match format edit descriptors. The use of a comma to terminate an input field and thus avoid padding the field is fully supported.

§ Namelist Groups

The NAMELIST statement allows for the definition of namelist groups. A namelist group allows for a special type of formatted input/output, where data is transferred between a named group of data items defined in a NAMELIST statement and one or more records in a file.

The general form of a namelist statement is:

```
NAMELIST /group-name/ namelist [[,] /group-name/ namelist ]...
```

where:

group-name is the name of the namelist group.

namelist is the list of variables in the namelist group.

§ Namelist Input

Namelist input is accomplished using a READ statement by specifying a namelist group as the input item. The following statement shows the format:

```
READ ([unit=] u, [NML=] namelist-group [, control-information])
```

One or more records are processed which define the input for items in the namelist group.

The records are logically viewed as follows:

```
$group-name   item=value   [,item=value]....  $ [END]
```

The following rules describe these input records:

1. The start or end delimiter ($) may be an &.
2. The start delimiter must begin in column 2 of a record.
3. The *group-name* begins immediately after the start delimiter.
4. The spaces or tabs may not appear within the *group-name*, within any *item*, or within any constants.

5. The *value* may be constants as are allowed for list directed input, or they may be a list of constants separated by commas (,). A list of items is used to assign consecutive values to consecutive elements of an array.

6. Spaces or tabs may precede the *item*, the = and the constants.

7. Array items may be subscripted.

8. Character items may be substringed.

§ Namelist Output

Namelist output is accomplished using a READ statement by specifying a namelist group as the output item. The following statement shows the format:

§
```
WRITE ([unit=] u, [NML=] namelist-group [,control-information])
```

The records output are logically viewed as follows:

```
$group-name
item = value
$ [END]
```

The following rules describe these output records:

1. One record is output per value.

2. Multiple values are separated by (,).

3. Values are formatted according to the rules of the list-directed write. Exception: character items are delimited by an apostrophe.

4. An apostrophe (') or a quote (") in the value is represented by two consecutive apostrophes or quotes.

Intrinsics A

This chapter specifies the *if77* intrinsic functions. All the FORTRAN 77 intrinsics are supported and are detailed in the ANSI Fortran manual listed in the section "Related Publications," in the Preface.

Table A-1. Zero Extend Functions

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
IZEXT	Zero-Extend Function	1	IZEXT	LOGICAL*1	INTEGER*2
	(Conversion Routine)			LOGICAL*2	INTEGER*2
				INTEGER*2	INTEGER*2
ZEXT	Zero-Extend Function		JZEXT	LOGICAL*1	INTEGER*4
	(Conversion Routine)			LOGICAL*2	INTEGER*4
				LOGICAL*4	INTEGER*4
				INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4

Table A-2. Math Intrinsic Functions

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
SQRT	Square Root	1	SQRT	REAL*4	REAL*4
			DSQRT	REAL*8	REAL*8
			CSQRT	COMPLEX*8	COMPLEX*8
			CDSQRT	COMPLEX*16	COMPLEX*16
LOG	Natural Logarithm	1	ALOG	REAL*4	REAL*4
			DLOG	REAL*8	REAL*8
			CLOG	COMPLEX*8	COMPLEX*8
			CDLOG	COMPLEX*16	COMPLEX*16
LOG10	Common Logarithm	1	ALOG10	REAL*4	REAL*4
			DLOG10	REAL*8	REAL*8
EXP	Exponential	1	EXP	REAL*4	REAL*4
			DEXP	REAL*8	REAL*8
			CEXP	COMPLEX*8	COMPLEX*8
			CDEXP	COMPLEX*16	COMPLEX*16

Table A-3. Trigonometric Functions (1 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
SIND	Sine(degree)	1	SIND	REAL*4	REAL*4
			DSIND	REAL*8	REAL*8
COSD	Cos (degree)	1	COSD	REAL*4	REAL*4
			DCOSD	REAL*8	REAL*8

Table A-3. Trigonometric Functions (2 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
TAND	Tan (degree)	1	TAND	REAL*4	REAL*4
			DTAND	REAL*8	REAL*8
ASIND	ArcSine (degree)	1	ASIND	REAL*4	REAL*4
			DASIND	REAL*8	REAL*8
ACOSD	ArcCosine (deg.)	1	ACOSD	REAL*4	REAL*4
			DACOSD	REAL*8	REAL*8
ATAND	ArcTangent (deg.)	1	ATAND	REAL*4	REAL*4
	Arc Tangent		DATAND	REAL*8	REAL*8
ATAN2D	ArcTangent (deg.)	2	ATAN2D	REAL*4	REAL*4
			DATAN2D	REAL*8	REAL*8
SIN	Sine	1	SIN	REAL*4	REAL*4
			DSIN	REAL*8	REAL*8
			DSINH	REAL*8	REAL*8
COSH	Hyperbolic Cosine	1	COSH	REAL*4	REAL*4
			DCOSH	REAL*8	REAL*8
TANH	Hyperbolic Tangent	1	TANH	REAL*4	REAL*4
			DTANH	REAL*8	REAL*8
			CSIN	COMPLEX*8	COMPLEX*8
			CDSIN	COMPLEX*16	COMPLEX*16
COS	Cos	1	COS	REAL*4	REAL*4
			DCOS	REAL*8	REAL*8
			CCOS	COMPLEX*8	COMPLEX*8
			CDCOS	COMPLEX*16	COMPLEX*16

Table A-3. Trigonometric Functions (3 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
TAN	Tangent	1	TAN	REAL*4	REAL*4
			DTAN	REAL*8	REAL*8
ASIN	ArcSine	1	ASIN	REAL*4	REAL*4
			DASIN	REAL*8	REAL*8
ACOS	ArcCosine	1	ACOS	REAL*4	REAL*4
			DACOS	REAL*8	REAL*8
ATAN	ArcTangent	1	ATAN	REAL*4	REAL*4
			DATAN	REAL*8	REAL*8
ATAN2	ArcTangent	2	ATAN2	REAL*4	REAL*4
			DATAN2	REAL*8	REAL*8
SINH	Hyperbolic Sine	1	SINH	REAL*4	REAL*4
			DSINH	REAL*8	REAL*8
COSH	Hyperbolic Cosine	1	COSH	REAL*4	REAL*4
			DCOSH	REAL*8	REAL*8
TANH	Hyperbolic Tangent	1	TANH	REAL*4	REAL*4
			DTANH	REAL*8	REAL*8

Table A-4. Arithmetic Functions (1 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
ABS	Absolute Value	1	IIABS	INTEGER*2	INTEGER*2
			JIABS	INTEGER*4	INTEGER*4
			ABS	REAL*4	REAL*4
			DABS	REAL*8	REAL*8
			CABS	COMPLEX*8	COMPLEX*8
			CDABS	COMPLEX*16	COMPLEX*16
IINT	Truncation	1		INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*2
			IINT	REAL*4	INTEGER*2
			IIFIX	REAL*4	INTEGER*2
			IIDINT	REAL*8	INTEGER*2
				COMPLEX*8	INTEGER*2
				COMPLEX*16	INTEGER*2
INT	Truncation	1		INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4
			INT	REAL*4	INTEGER*4
			JIFIX	REAL*4	INTEGER*4
			IDINT	REAL*8	INTEGER*4
				COMPLEX*8	INTEGER*4
				COMPLEX*16	INTEGER*4
JINT	Truncation	1		INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4
			JINT	REAL*4	INTEGER*4
			JIDINT	REAL*8	INTEGER*4
				COMPLEX*8	INTEGER*4
				COMPLEX*16	INTEGER*4

Table A-4. Arithmetic Functions (2 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
AINT	Truncation	1	AINT	REAL*4	REAL*4
			DINT	REAL*8	REAL*8
ININT	Nearest Integer	1	ININT	REAL*4	INTEGER*2
	[a + .5 * sign(a)]		IIDNNT	REAL*8	INTEGER*2
JNINT	Nearest Integer	1	JNINT	REAL*4	INTEGER*4
	[a + .5 * sign(a)]		JIDNNT	REAL*8	INTEGER*4
NINT	Nearest Integer	1	NINT	REAL*4	INTEGER*4
	[a + .5 * sign(a)]		IDNINT	REAL*8	INTEGER*4
ANINT	Nearest Whole Number	1	ANINT	REAL*4	REAL*4
	int(a + .5) if a ≥ 0 int(a - .5)if a < 0		DNINT	REAL*8	REAL*8
MAX	Maximum	n > 1	IMAX0	INTEGER*2	INTEGER*2
			IMAX1	REAL*4	INTEGER*2
			AIMAX0	INTEGER*2	REAL*4
			JMAX0	INTEGER*4	INTEGER*4
			JMAX1	REAL*4	INTEGER*4
			AJMAX0	INTEGER*4	REAL*4

Table A-4. Arithmetic Functions (3 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
DIM	Positive Difference	2	IIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
			DIM	REAL*4	REAL*4
			DDIM	REAL*8	REAL*8
MIN	Minimum	n > 1	IMIN0	INTEGER*2	INTEGER*2
			IMIN1	REAL*4	INTEGER*2
			AIMIN0	INTEGER*2	REAL*4
			JMIN0	INTEGER*4	INTEGER*4
			JMIN1	REAL*4	INTEGER*4
			AJMIN0	INTEGER*4	REAL*4
MOD	Remainder	2	IMOD	INTEGER*2	INTEGER*2
			JMOD	INTEGER*4	INTEGER*4
			AMOD	REAL*4	REAL*4
			DMOD	REAL*8	REAL*8
SIGN	Transfer of Sign	2	IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
			SIGN	REAL*4	REAL*4
			DSIGN	REAL*8	REAL*8

Table A-5. Type Conversion Functions (1 of 2)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
REAL	Convert to REAL*4	1	FLOATI	INTEGER*2	REAL*4
			REAL	INTEGER*4	REAL*4
			FLOATJ	INTEGER*4	REAL*4
				REAL*4	REAL*4
			SNGL	REAL*8	REAL*4
				COMPLEX*8	REAL*4
				COMPLEX*16	REAL*4
DBLE	Convert to REAL*8	1	DFLOTI	INTEGER*2	REAL*8
			DFLOAT	INTEGER*4	REAL*8
			DFLOTJ	INTEGER*4	REAL*8
				REAL*4	REAL*8
				REAL*8	REAL*8
				COMPLEX*8	REAL*8
			DREAL	COMPLEX*16	REAL*8
CMPLX	Convert to COMPLEX*8 or COMPLEX*8 from two arguments	1,2		INTEGER*2	COMPLEX*8
				INTEGER*4	COMPLEX*8
				REAL*4	COMPLEX*8
				REAL*8	COMPLEX*8
		1		COMPLEX*8	COMPLEX*8
				COMPLEX*16	COMPLEX*8

Table A-5. Type Conversion Functions (2 of 2)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
DCMPLX	Convert to COMPLEX*16 or COMPLEX*16 from two arguments	1,2 1		INTEGER*2 INTEGER*4 REAL*4 REAL*8 COMPLEX*8 COMPLEX*16	COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16
AIMAG	Imag Part of Complex	1	AIMAG DIMAG	COMPLEX*8 COMPLEX*16	REAL*4 REAL*8
CONJG	Complex Conjugate	1	CONJG DCONJG	COMPLEX*8 COMPLEX*16	COMPLEX*8 COMPLEX*16

Table A-6. Bitwise Functions (1 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
IAND	Bitwise AND Perform a logical AND on bits	2	IIAND JIAND	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
AND					typeless[1]
IOR	Bitwise OR Perform a logical OR on bits	2	IIOR JIOR	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
OR				1	typeless[1]
IEOR	Bitwise XOR	2	IIEOR	INTEGER*2	INTEGER*2
XOR	logical Exclusive OR		JIEOR	INTEGER*4	INTEGER*4
NEQV				1	typeless[1]
EQV	Bitwise Excl. NOR Performs a logical Exclusive Nor	2		1	typeless[1]

Table A-6. Bitwise Functions (2 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
NOT	Bitwise Complement Complements each bit	1	INOT JNOT	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
COMPL	Bitwise Complement	1		1	typeless[1]
LOC	The address of a data item is returned (Assumes 32-bit address)	1	LOC	INTEGER*2 INTEGER*4 REAL*4 REAL*8 COMPLEX*8 COMPLEX*1	INTEGER*4 INTEGER*4 INTEGER*4 INTEGER*4 INTEGER*4 INTEGER*4
ISHFT	Bitwise Shift a1 logically shifted left a2	2	IISHFT JISHFT	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
SHIFT	bits. If a2 < 0 then right logical shift.				typeless[2]
LSHIFT	Bitwise Left Shift a1 logically shifted left	2	ILSHIFT JLSHIFT	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
RSHIFT	Bitwise Right Shift a1 logically shifted right		IRSHIFT JRSHIFT	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
ISHFTC	Circular Shift Rightmost a3 bits of a1 are shifted circularly by a2 bits; remaining bits in a1 are unaffected.	3	IISHFTC JISHFTC	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
CHAR	Character Returns a character that has the ASCII value specified by the argument.	1	 CHAR	LOGICAL*1 INTEGER*2 INTEGER*4	CHARACTER CHARACTER CHARACTER

Table A-6. Bitwise Functions (3 of 3)

Generic Name	Functions	Number of Args	Specific Name	Type of Argument	Type of Result
IBITS	Bit Extraction	3	IIBITS	INTEGER*2	INTEGER*2
	Extracts bits a2 through (a2 +a 3 - 1) from a1.		JIBITS	INTEGER*4	INTEGER*4
IBSET	Set Bit	2	IIBSET	INTEGER*2	INTEGER*2
	Returns a1 with bit a2 set to 1.		JIBSET	INTEGER*4	INTEGER*4
BTEST	Bit Test	2	BITEST	INTEGER*2	LOGICAL*2
	.TRUE. if bit a2 of a1 is a 1.		BJTEST	INTEGER*4	LOGICAL*4
IBCLR	Bit Clear	2	IIBCLR	INTEGER*2	INTEGER*2
	Returns a1 with bit a2 set to 0		JIBCLR	INTEGER*4	INTEGER*4

1. The arguments to the intrinsics AND, OR, NEQV, EQV, and COMPL may be of any type except for CHARACTER and COMPLEX.
2. The first argument to the SHIFT intrinsic may be of any type except for CHARACTER and COMPLEX. The second argument is any integer type.

VAX Built-in Functions and System Subroutines

B

This appendix discusses the VAX/VMS built-in functions and system subroutines supported by *if77*.

Built-in Functions

The built-in functions perform inter-language utilities for argument passing and location calculations. The following built-in functions are available:

`%LOC(`*arg*`)`	Compute the address of the argument *arg*.
`%REF(`*a*`)`	Pass the argument *a* by reference.
`%VAL(`*a*`)`	Pass the argument as a 32-bit immediate value (64-bit if it is double precision.)

VAX/VMS System Subroutines

DATE

The DATE subroutine returns a nine-byte string containing the ASCII representation of the current date. It has the form:

```
CALL DATE(buf)
```

where *buf* is a nine-byte variable, array, array element, or character substring. The date is returned as a nine-byte ASCII character string of the form:

```
dd-mmm-yy
```

Where:

dd	Is the two-digit day of the month
mmm	Is the three-character abbreviation of the month
yy	Is the last two digits of the year

EXIT

The EXIT subroutine causes program termination, closes all open files, and returns control to the operating system. It has the form:

```
CALL EXIT[(exit_status)]
```

where *exit_status* is an optional integer argument used to specify the image exit value.

GETARG

```
SUBROUTINE GETARG(N, ARG)
   INTEGER*4    N
   CHARACTER* (*) ARG
```

The GETARG subroutine returns the *N*th command line argument in character variable *ARG*. For *N* equal to zero, the name of the program is returned.

IARGC

```
INTEGER*4    FUNCTION IARGC()
```

The IARGC subroutine returns the number of command line arguments following the program name.

IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. It has the form:

```
CALL IDATE(IMONTH, IDAY, IYEAR)
```

If the current date were October 9, 1992, the values of the integer variables upon return would be:

```
IMONTH = 10
IDAY = 9
IYEAR = 92
```

MVBITS

The MVBITS subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination). MVBITS transfers *a3* bits from positions *a2* through (*a2* + *a3* - 1) of the source, *src*, to positions *a5* through (*a5* + *a3* - 1) of the destination, *dest*. Other bits of the destination location remain unchanged. The values of (*a2* + *a3*) and (*a5* + *a3*) must be less than or equal to 32. It has the form:

```
CALL MVBITS(src, a2, a3, dest, a5)
```

Where:

src is an integer variable or array element that represents the source location.

a2 is an integer expression that identifies the first position in the field transferred from *src*.

a3 is an integer expression that identifies the length of the field transferred from *src*.

dest is an integer variable or array element that represents the destination location.

a5 is an integer expression that identifies the starting position within *a4*, for the bits being transferred.

RAN

The RAN subroutine returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1. The result is a floating point number that is uniformly distributed in the range between 0.0 and 1.0 exclusive. It has the form:

```
y = RAN(i)
```

where *y* is set equal to the value associated by the function with the seed argument *i*. The argument *i* must be an INTEGER*4 variable or INTEGER*4 array element.

The argument *i* should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers. The seed is updated automatically, and RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 6969 * SEED + 1 ! MOD 2**32
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

If the command-line option to treat all REAL declarations as DOUBLE PRECISION declarations is in effect, RAN returns a DOUBLE PRECISION value.

SECNDS

The SECNDS subroutine provides system time of day, or elapsed time, as a floating point value in seconds. It has the form:

```
y = SECNDS(x)
```

where (REAL or DOUBLE PRECISION) *y* is set equal to the time in seconds since midnight, minus the user supplied value of the (REAL or DOUBLE PRECISION) *x*. Elapsed time computations can be performed with the following sequence of calls.

```
X = SECNDS(0.0)
...
...   ! Code to be timed
...
DELTA = SECNDS(X)
```

The accuracy of this call is the same as the resolution of the system clock.

TIME

The TIME subroutine returns the current system time as an ASCII string. It has the form:

```
CALL TIME(buf)
```

where *buf* is an eight-byte variable, array, array element, or character substring. The TIME call returns the time as an eight-byte ASCII character string of the form:

```
hh:mm:ss
```

For example:

```
16:45:23
```

Note that a 24-hour clock is used.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W