FORTRAN 3.0.1 Reference Manual

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-7251-10
Revision A, August 1994

Contents

Tables

Preface

This preface is organized into the following sections.

Purpose and Audience

This manual describes language and routines of SunPro™ FORTRAN 3.0.1.

This is a *reference* manual; and though it has many examples, it is in no way a tutorial. Its function and purpose is solely to help you find features or routines quickly, not to help you learn FORTRAN, programming, or programming style.

This book is for scientists and engineers with the following background:

- Thorough knowledge and experience with FORTRAN programming
- General knowledge and understanding of some operating system
- Particular knowledge of SunOS™ or UNIX[1] commands `cd`, `pwd`, `ls`, `cat`.

1. UNIX is a registered trademark of Novell, Inc., in the United States and other countries.

For help using the compiler, linker, debugger, related utilities, or making or using libraries, refer to the *FORTRAN User's Guide.*

How this Book is Organized

This book is organized as follows:

Related Manuals

The following documents are provided on-line or in hard copy, as indicated.

Title	**Paper**	**AnswerBook**
FORTRAN 3.0.1 User's Guide	X	X
FORTRAN 3.0.1 Reference Manual	X	X
Debugging a Program	X	X
Numerical Computation Guide	X	X
Installing SunSoft Developer Products Software on Solaris	X	X

Conventions in Text

We use the following conventions in this manual to display information.

- We show code listings examples in boxes.

```
      WRITE( *, * ) 'Hello world'
```

- The `plain typewriter font` shows prompts and coding.
- In dialogs, the **`boldface typewriter font`** shows text the user types in.

```
demo$ echo hello
hello
demo$ ■
```

- *Italics* indicate general arguments or parameters that you should replace with the appropriate input. Italics also indicate emphasis.
- For Solaris 2.x, the default shell is `sh` and the default prompt is the dollar sign (`$`). Most systems have distinct host names, and you can read some of our examples more easily if we use a symbol longer than a dollar sign. Examples generally use "`demo$`" as the system prompt; where the `csh` shell is shown, we use "`demo%`" as the system prompt.
- The small clear triangle Δ shows a blank space where that is significant.

```
ΔΔ36.001
```

- We generally tag nonstandard features with a small black diamond (♦). Wherever we indicate that a feature is *nonstandard*, that means a program using it does not conform to the ANSI X3.9-1978 standard, as described in *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, April 1978, American National Standards Institute, Inc., abbreviated as the FORTRAN Standard.
- We usually show FORTRAN examples in tab format, not fixed column. See Section 1.9, "Source Line Formats," for details.
- We usually abbreviate "FORTRAN" as "`f77`".

Elements of FORTRAN 1

This chapter is organized into the following sections.

1.1 Operating Environments

This manual describes FORTRAN 3.0.1 under Solaris® 1.x and 2.x for SPARC, and under Solaris 2.x for x86 operating environments. Most aspects of FORTRAN for 1.x, 2.x, and x86 are the same, including functionality, behavior, and features. Anything unique to one operating environment is tagged “(2.x only)” or sometimes “(1.x)” or “(x86)”.

1

The previous major release was ported to Intel[1] 80386-compatible computers running Solaris 2.x for x86, and some features remain identified as being for x86 only.

1.2 Standards

This FORTRAN is an enhanced FORTRAN 77 development system.

- It conforms to the ANSI X3.9-1978 FORTRAN standard and the corresponding International Standards Organization number is ISO 1539-1980. NIST (formerly GSA and NBS) validates it at appropriate intervals.
- It also conforms to the standards FIPS 69-1, BS 6832, and MIL-STD-1753.
- It provides an IEEE standard 754-1985 floating-point package.
- On SPARC systems it provides support for optimization exploiting features of SPARC V8, including the SuperSPARC™ implementation[2]. These features are defined in the *SPARC Architecture Manual: Version 8*.

1.3 Extensions

This FORTRAN compiler provides iMPact™ multiprocessor FORTRAN and lint-like checking across routines for consistency of arguments, commons, parameters, and so forth. Other extensions include recursion, pointers, double-precision complex, quadruple-precision real, quadruple-precision complex, and many VAX[3] VMS FORTRAN 5.0 extensions, including `NAMELIST`, `DO WHILE`, structures, records, unions, maps, and variable formats. Multiprocessor FORTRAN includes automatic and explicit loop parallelization. You can write FORTRAN programs with many VMS extensions so that these programs run with the same source code on both SPARC and VAX systems.

1. Intel is a registered trademark of Intel Corporation.

2. SuperSPARC is a trademarks of Texas Instruments, Inc.

3. VAX and VMS are a trademarks of Digital Equipment Corporation

1.4 Basic Terms

This section introduces the basic terms and concepts.

- A *program* consists of one or more program units.
- A *program unit* is a sequence of statements, terminated by an `END`.
- A *statement* consists of zero or more key words, symbolic names, literal constants, statement labels, operators, and special characters.
- Each *key word*, *symbolic name*, *literal constant*, and *operator* consists of one or more characters from the FORTRAN character set.
- A *character constant* can include any valid `ASCII` character.
- A *statement label* consists of 1 to 5 digits, with at least one nonzero.

1.5 Character Set

The character set consists of the following.

- Uppercase and lowercase letters `A – Z` and `a – z`
- Numerals `0 – 9`
- The following special characters

Table 1-1 Special Characters

Character	***Name***	***Character***	***Name***
Space	Space	'	Apostrophe
Tab	Tab	"	Quote ♦
=	Equals	$	Dollar sign ♦
+	Plus	_	Underscore ♦
–	Minus	!	Exclamation point ♦
*	Asterisk	:	Colon
/	Slash	?	Question mark ♦
(	Left parenthesis	%	Percent ♦
)	Right parenthesis	&	Ampersand ♦
,	Comma	\	Backslash ♦
.	Period	<	Left angle bracket ♦
		>	Right angle bracket ♦

Usage and Restrictions

- Uppercase and lowercase are not significant in the key words of FORTRAN statements or in symbolic names.

The –U option of f77 makes case significant in symbolic names. ♦

- Control Characters ♦

  Even though they are not in the character set, most control characters are allowed as *data.* The exceptions are:

  Control A, Control B, Control C

  These are not allowed as data.

  While entering a character string, you must not hold down the control key and press the A, or B, or C. Even these characters can be entered other ways, such as with the char() function.

- Special characters used for punctuation:

Table 1-2 Special Character Usage

Character	***Usage***
Space	Ignored in statements, except as part of a character constant
Tab	Establish the line as a tab-format source line ♦
=	Assignment
+	Add, unary operator
–	Subtract, unary operator
*	Multiply, alternate returns, comments, exponentiation, `stdin`, `stdout`, list-directed I/O
/	Divide, delimit data, labeled commons, structures, end-of-record
( )	Enclose expressions, complex constants, equivalence groups, formats, argument lists, subscripts
,	Separator for data, expressions, complex constants, equivalence groups, formats, argument lists, subscripts
.	Radix point, delimiter for logical constants and operators, record fields
'	Quoted character literals
"	Quoted character literals, octal constants ♦
$	Delimit namelist input, edit descriptor, pragmas ♦
!	Comments ♦
:	Array declarators, substrings, edit descriptor
%	Special functions: `%REF, %VAL, %LOC` ♦
&	Continuation, alternate return, delimit namelist input; in column 1: establish the line as a tab-format source line ♦
?	Request names in namelist group ♦
\	Escape character ♦
< >	Enclose variable expressions in formats ♦

- Any ASCII character is valid as literal data in a character string. ♦

  For Control A, Control B, and Control C, do not hold down the control key and press the A, or B, or C; use `char()`, or some other way. For the backslash (\ ) character you may need to use an escape sequence or use the `-xl` compiler option. The backslash (\ ) is also called a reverse solidus, and the slash (/) is also called a solidus. For the newline (\ n) character you must use an escape sequence. See Table 2-5.

1.6 Symbolic Names

The following items can have symbolic names.

Table 1-3 Items with Symbolic Names

Symbolic constants	Labeled commons
Variables	Namelist groups ♦
Arrays	Main programs
Structures ♦	Subroutines
Records ♦	Functions
Record fields ♦	Entry points

Restrictions

Symbolic names have the following restrictions.

- They can have from 1 to 32 characters. (The standard is 6.) ♦
- They consist of letters, digits, the dollar sign ($), and the underscore character (_). The $ and the _ are not standard. ♦
- They generally start with a letter—never with a digit or dollar sign ($). Names that start with an underscore (_) are allowed, ♦ but it is safer to reserve such names for the compiler.
- Uppercase and lowercase are not significant; the compiler converts them all to lowercase. The `-U` option on the `f77` command line overrides this default, thereby preserving any uppercase used in your source file. ♦
- Example: These are equivalent with the default in effect.

```
ATAD = 1.0E-6
Atad = 1.0e-6
```

Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice.

- The space character is not significant.

  Example: These are equivalent.

```
      IF ( X .LT. ATAD ) GO TO 9
      IF ( X .LT. A TAD ) GO TO 9
      IF(X.LT.ATAD)GOTO9
```

- Sample Symbolic Names

Table 1-4 Sample Symbolic Names

Valid	***Invalid***	***Reason***
`X2`	`2X`	Starts with a digit
`DELTA_TEMP`	`_DELTA_TEMP`	Starts with an _ (reserved for compiler)
`Y$Dot`	`Y|Dot`	Invalid character |

- In general, for any single program unit, different entities cannot have the same symbolic name. Exceptions:
  - A variable or array can have the same name as a common block.
  - A field of a record can have the same name as a structure. ♦
  - A field of a record can have the same name as a field at a different level of the structure. ♦
- Throughout any program of more than one programming unit, no two of the following can have the same name.
  - Block data subprograms
  - Common blocks
  - Entry points
  - Function subprograms
  - Main program
  - Subroutines

1.7 Program

A program unit is a sequence of statements, terminated by an END statement. Every program unit is either a main program or a subprogram. If a program is to be executable, it must have a main program.

There are three types of subprograms: subroutines, functions, and block data subprograms. The subroutines and functions are called *procedures*. The procedures are invoked from other procedures or from the main program. The block data subprograms are handled by the loader.

1.8 Statements

A statement consists of one or more key words, symbolic names, literal constants, and operators, with appropriate punctuation. In FORTRAN, no key words are reserved in all contexts. Most statements begin with a key word; the exceptions are the statement function and assignment statements.

Executable Statements

Every statement is either executable or nonexecutable. In general, if a statement specifies an action to be taken at runtime, it is called *executable*. Otherwise, it is called *nonexecutable*.

The nonexecutable statements specify attributes such as type and size; determine arrangement or order; define initial data values; specify editing instructions; define statement functions; classify program units; and define entry points. In general, the nonexecutable statements are completed before execution of the first executable statement.

FORTRAN Statements

Table 1-5 FORTRAN Statements

The asterisk (*) indicates an executable statement.

ACCEPT*	DOUBLE COMPLEX	GOTO (Assigned)*	PRINT*
ASSIGN*	DOUBLE PRECISION	GOTO (Unconditional)*	PRAGMA
Assignment*	ELSE*	IF (Arithmetic)*	PROGRAM
AUTOMATIC	ELSE IF*	IF (Block)*	REAL
BACKSPACE*	ENCODE*	IF (Logical)*	RECORD
BLOCK DATA	END*	IMPLICIT	RETURN*
BYTE	END DO	INCLUDE	REWIND*
CALL*	END FILE*	INQUIRE*	SAVE
CHARACTER	END IF*	INTEGER	Statement Function
CLOSE*	END MAP	INTRINSIC	STATIC*
COMMON	END STRUCTURE	LOGICAL	STOP*
COMPLEX	END UNION	MAP	STRUCTURE
CONTINUE*	ENTRY	NAMELIST	SUBROUTINE*
DATA	EQUIVALENCE	OPEN*	TYPE
DECODE*	EXTERNAL	OPTIONS	UNION
DIMENSION	FORMAT	PARAMETER	VIRTUAL
DO*	FUNCTION	PAUSE*	VOLATILE
DO WHILE*	GOTO*	POINTER	WRITE*

1.9 Source Line Formats

A statement takes one or more lines; the first line is called the *initial line,* and the subsequent lines are called the *continuation lines.*

You can format a source line in either of two ways.

- Standard fixed format
- Tab format ♦

Standard Fixed Format

The standard fixed format source lines are defined as follows.

- The first 72 columns of each line are scanned. See “Extended Lines,” below.
- The first five columns must be blank or contain a numeric label.
- Continuation lines are identified by a nonblank, nonzero in column 6.
- Short lines are padded to 72 characters.
- Long lines are truncated. See “Extended Lines,” below.

Tab-Format

The tab-format source lines are defined as follows. ♦

- A tab in any of columns 1 through 6, or an ampersand in column 1, establishes the line as a tab-format source line.
- If the tab is the first nonblank character, the text following the tab is scanned as if it started in column 7.
- A comment indicator or a statement number may precede the tab.
- Default maximum line length is 72 columns. See “Extended Lines,” below.
- Continuation lines are identified by an ampersand (&) in column 1, or a nonzero digit after the first tab.

Mixing Formats

You can format lines both ways in one program unit, but not in the same line.

Continuation Lines

The default maximum number of continuation lines is 99 ♦ (1 initial and 99 continuation). To change this number of lines, use the `-Nl`*n* option. ♦

Extended Lines

To **e**xtend the source line length to 132 characters, use the `–e` option.♦ Otherwise, by default, `f77` ignores any characters after column 72.
Example: Compile to allow extended lines.

```
demo$ f77 -e prog.f
```

Padding

Padding is significant in lines such as the two in the `DATA` statement below.

```
C        1         2         3         4         5         6         7
C234567890123456789012345678901234567890123456789012345678901234567890123456789012
     DATA SIXTYH/60H
1               /
```

Comments and Blank Lines

- A line with a `c`, `C`, `*`, `d`, `D`, or `!` in column one is a comment line, except that if the `-xld` option is set, then the lines starting with `D` or `d` are compiled as debug lines. The `d`, `D`, and `!` are nonstandard. ♦
- If you put an exclamation point (`!`) in any column of the statement field, except within character literals, then everything after the `!` on that line is a comment. ♦
- A totally blank line is a comment line.

Example: `c`, `C`, `d`, `D`, `*`, `!`, and blank comments.

```
c     Start expression analyzer
      CHARACTER S, STACK*80
      COMMON /PRMS/ N, S, STACK
      …
*     Crack the expression:
      IF ( S .GE. '0' .AND. S .LE. '9' ) THEN ! EoL comment
           CALL PUSH ! Save on stack. EoL comment
d          PRINT *, S! Debug comment & EoL comment
      ELSE
           CALL TOLOWER ! To lowercase EoL comment
      END IF
D     PRINT *, N ! Debug comment & EoL comment
      …
C     Finished
!     expression analyzer
```

Pragmas

A pragma passes information to a compiler in a special form of comment. ♦ Pragmas are also called *compiler directives.* Currently there is a C pragma, but see also "Parallel Pragma," in the next section. The general form is as follows.

```
c$pragma id
or
c$pragma id ( a [ , a ] … ) [ , id ( a [ , a ] … ) ] ,…
```

The variable *id* identifies the kind of pragma, and *a* is an argument.

Syntax

A pragma has the following syntax.

- In column one, any of the comment-indicator characters `c`, `C`, `d`, `D`, `!`, or `*`
- In *any* column, the `!` comment-indicator character
- The next 7 characters (with no blanks) are `$pragma`, in any mix of uppercase and lowercase

Rules and Restrictions for Pragmas

- After the first eight characters, blanks are ignored, and uppercase and lowercase are equivalent, as in FORTRAN text.
- Because it is a comment, a pragma cannot be continued, but you can have many `c$pragma` lines, one after the other, as needed.
- If a comment satisfies the above syntax, it is expected to contain one or more directives recognized by the compiler; if it does not, a warning is issued.

The `C()` *Directive*

The `C()` directive specifies that its arguments are external functions written in the C language. It is equivalent to an `EXTERNAL` declaration with the addition that the FORTRAN compiler does not append an underscore to such names, as it ordinarily does with external names.

The `C()` directive for a particular function must appear before the first reference to that function in each subprogram that contains such a reference. The recommended usage is as follows.

```
      EXTERNAL ABC, XYZ  !$PRAGMA C(ABC, XYZ)
```

Parallel Pragma

A *parallel pragma* is a special comment that directs the compiler to do some parallelizing.

The current parallel pragma for explicit parallelizing is the following.

- `c$par doall`

Syntax

A *parallel* pragma has the following syntax.

- The first character is in column one. Only *parallel* pragmas require this.
- The first character can be any one of `c`, `C`, `d`, `D`, `*`, or `!`.
- The next 4 characters are `$par`, no blanks, any uppercase and lowercase.

For `doall`, the compiler parallelizes the *next loop* it finds after the pragma, if possible.

Before using a parallel pragma, read the *FORTRAN User's Guide*, especially the appendixes on parallelization.

1

Data Types and Data Items 2

This chapter is organized into the following sections.

2.1 Types

Any constant or constant expression usually represents typed data (the exceptions are the typeless constants). Any name of a variable, array, array element, substring, or function usually represents typed data.

The following items have data types.

Constant Expressions
Variables
Arrays
External Functions
Statement Functions

These items do *not* have data types.

Main Programs	Common Blocks
Subroutines	Namelist Groups ♦
Block Data Subprograms	Structured Records ♦

Rules for Data Typing

Name determines type; that is, the name of a datum or function determines its data type, explicitly or implicitly, according to the following rules of data typing.

- A symbolic name of a constant, variable, array, or function has only one data type for each program unit, except for generic functions.
- If you explicitly list a name in a type statement, then that determines the data type.
- If you do not explicitly list a name in a type statement, then the first letter of the name determines the data type implicitly.
- The default implicit typing rule is that if the first letter of the name is `I`, `J`, `K`, `L`, `M`, or `N`, then the data type is integer, otherwise it is real.
- You can change the default implied types by using the `IMPLICIT` statement, even to the extent of turning off all implicit typing with the `IMPLICIT NONE` statement. You can also turn off all implicit typing by specifying the `-u` compiler flag on the command line; this is equivalent to beginning each program unit with the `IMPLICIT NONE` statement.

Array Elements

An array element has the same type as the array name.

Functions

Each intrinsic function has a specified type. An intrinsic function does not require an explicit type statement, but that is allowed. A generic function does not have a predetermined type; the type is determined by the type of the arguments, as shown in the chapter on intrinsic functions.

An external function can have its type specified in any of the following ways.

- Explicitly by putting its name in a type statement
- Explicitly in its `FUNCTION` statement, by preceding the word '`FUNCTION`' with the name of a data type.
- Implicitly by its name, as with variables.

Example: Explicitly by putting its name in a type statement.

```
      FUNCTION F ( X )
      INTEGER F, X
      F = X + 1
      RETURN
      END
```

Example: Explicitly in its `FUNCTION` statement.

```
      INTEGER FUNCTION F ( X )
      INTEGER X
      F = X + 1
      RETURN
      END
```

Example: Implicitly by its name, as with variables.

```
      FUNCTION NXT ( X )
      INTEGER X
      NXT = X + 1
      RETURN
      END
```

Consistent Typing of Functions

Implicit typing can affect the type of a function, either by default implicit typing or by an `IMPLICIT` statement. It is your responsibility to make the data type of the function be the same within the function subprogram as it is in the calling program unit. That is, FORTRAN does no type checking between program units.

2

Properties of Data Types

This section describes the data types, what each is for, the way storage is allocated for each of them, and the alignment of the different types. Storage and alignment are always given in bytes. Values that can fit into a single byte are byte-aligned.

BYTE ♦

The BYTE data type provides a data type that uses only one byte of storage. It is a logical data type, and has the synonym LOGICAL*1.

A variable of type BYTE can hold any of the following.

- One character
- An integer between -128 and 127
- The logical values .TRUE. or .FALSE.

If it is interpreted as a logical value, a value of 0 represents .FALSE., and any other value is interpreted as .TRUE.

f77 allows the BYTE type as an array index (just as it allows the REAL type) but it does not allow BYTE as a DO loop index (where it allows only INTEGER, REAL, and DOUBLE PRECISION). Wherever FORTRAN makes an explicit check for INTEGER, it does not allow BYTE.

Examples

```
      BYTE Bit3 / 8 /, C1 / 'W' /,
&          Counter / 0 /, Switch / .FALSE. /
```

Storage— A BYTE item occupies 1 byte of storage.

Alignment— A BYTE item is aligned on 1-byte boundaries.

CHARACTER

- The character data type, CHARACTER, which has the synonym CHARACTER*1, holds 1 character.
- The character is enclosed in apostrophes (') or quotes (").✦ Allowing quotes (") is nonstandard, and if you compile with the -xl option, quotes mean something else, and you must use apostrophes to enclose a string.
- Data of type CHARACTER is always unsigned.

Storage— A CHARACTER item occupies 1 byte (8 bits) of storage.

Alignment— A CHARACTER item is aligned on 1-byte boundaries.

CHARACTER*n

The character string data type, CHARACTER*n, where *n* > 0, holds a string of *n* characters.

Storage— A CHARACTER*n item occupies *n* bytes of storage.

Alignment— A CHARACTER*n *variable* is aligned on 1-byte boundaries.

Every character string *constant* is aligned on 2-byte boundaries, and if it does not appear in a DATA statement, it is followed by a null character to ease communication with C routines.

COMPLEX

A complex datum is an approximation of a complex number. The complex data type, COMPLEX, which usually has the synonym COMPLEX*8, is a pair of REAL*4 values that represent a complex number. The first element represents the real part and the second represents the imaginary part.

Storage— The usual default size for a COMPLEX item (no size specified) is 8. If the -r8 compiler option is set, then the default size is 16; otherwise it is 8.

Alignment— It is aligned on 4-byte boundaries; except if compiled on a Sun-4 or SPARC computer with the -f option, in which case it is aligned on 8-byte boundaries.

COMPLEX*8 ♦

The complex data type COMPLEX*8 is a synonym for COMPLEX, except that it always has a size of 8 bytes, independent of any compiler options.

COMPLEX*16 *(Double Complex)* ♦

The complex data type COMPLEX*16 is a synonym for DOUBLE COMPLEX, except that it always has a size of 16 bytes, independent of any compiler options.

COMPLEX*32 (*Quad Complex)* ♦

(*SPARC only*) The complex data type COMPLEX*32 is a quadruple-precision complex. It is a pair of REAL*16 elements, where each has a sign bit, a 15-bit exponent, and a 112-bit fraction. These REAL*16 elements in f77 conform to the IEEE standard.

Storage— The size for a COMPLEX*32 item is 32 bytes.

Alignment— It is aligned on 4-byte boundaries; except if compiled on a Sun-4 or SPARC computer with the -f option, in which case it is aligned on 8-byte boundaries.

DOUBLE COMPLEX ♦

The complex data type, DOUBLE COMPLEX, which usually has the synonym COMPLEX*16, is a pair of DOUBLE PRECISION (REAL*8) values that represents a complex number. The first element represents the real part and the second represents the imaginary part.

Storage— The usual default size for DOUBLE COMPLEX (no size specified) is 16.

- If the -r8 compiler option is set, then the default size is 32, otherwise 16.

Alignment— It is aligned on 4-byte boundaries; except if compiled on a Sun-4 or SPARC computer with the -f option, in which case it is aligned on 8-byte boundaries.

DOUBLE PRECISION

A double-precision datum is an approximation of a real number. The double-precision data type, DOUBLE PRECISION, which has the synonym REAL*8, holds one double-precision datum.

Storage— The usual default size for a DOUBLE PRECISION item (no size specified) is 8.

- If the –r8 compiler option is set, then the default size is 16, otherwise 8.

Alignment— It is aligned on 4-byte boundaries.

A DOUBLE PRECISION element has a sign bit, an 11-bit exponent, and a 52-bit fraction. These DOUBLE PRECISION elements in f77 conform to the IEEE standard for double-precision floating-point data. The layout is shown in the appendix on data representation.

INTEGER

The integer data type, INTEGER, holds a signed integer.

Storage— The usual default size for an INTEGER item (no size specified) is 4.

- If the –i2 compiler option is set, then the default size is 2; otherwise it is 4.
- If the –r8 compiler option is set, then the default size is 8; otherwise it is 4.
- If both the –i2 and –r8 options are set, then the results are unpredictable.

Alignment— It is aligned on 4-byte boundaries, unless the –i2 option is set, then it is aligned on 2-byte boundaries.

INTEGER*2 ©

The short integer data type, INTEGER*2, holds a signed integer. An expression involving only objects of type INTEGER*2 is of that type. Using this feature may have adverse performance implications and we do not recommend it.

Generic functions return short or long integers depending on the default integer type. If a procedure is compiled using the –i2 flag, all integer constants that fit and all variables of type INTEGER (no explicit size) are of type INTEGER*2. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one is chosen that returns the

prevailing length (INTEGER*2) when the –i2 command flag is in effect). When the –i2 option is in effect, the default length of LOGICAL quantities is 2 bytes.

Ordinary integers follow the FORTRAN 77 rules about occupying the same space as a REAL variable. They are assumed to be equivalent to the C type long int, and 1-byte integers are of C type short int. These short integer and logical quantities do not obey the standard rules for storage association.

Storage— An INTEGER*2 item occupies 2 bytes.

Alignment— It is aligned on 2-byte boundaries.

INTEGER*4 ♦

The integer data type, INTEGER*4, holds a signed integer.

Storage— An INTEGER*4 item occupies 4 bytes.

Alignment— It is aligned on 4-byte boundaries.

LOGICAL

The logical data type, LOGICAL, holds a logical value .TRUE. or .FALSE. The value 0 represents .FALSE.; any other value represents .TRUE.

Storage— The usual default size for an LOGICAL item (no size specified) is 4.

- If the –i2 compiler option is set, then the default size is 2; otherwise it is 4.
- If the –r8 compiler option is set, then the default size is 8; otherwise it is 4.
- If both the –i2 and –r8 options are set, then the results are unpredictable.

Alignment— It is aligned on 4-byte boundaries, unless the –i2 option is set, then it is aligned on 2-byte boundaries.

If the –i2 compiler flag is set, then LOGICAL (without any size specification) is the same as LOGICAL*2; otherwise it is the same as LOGICAL*4.

LOGICAL*1 ♦

The 1-byte logical data type, LOGICAL*1, which has the synonym BYTE, can hold any of the following:

- One character
- An integer between -128 and 127
- The logical values .TRUE. or .FALSE.

The value is as defined for LOGICAL, but it can hold a character or small integer. Examples

```
      LOGICAL*1  Bit3 / 8 /, C1 / 'W' /,
&               Counter / 0 /, Switch / .FALSE. /
```

Storage— A LOGICAL*1 item occupies 1 byte of storage.

Alignment— A LOGICAL*1 item is aligned on 1-byte boundaries.

LOGICAL*2 ♦

The data type, LOGICAL*2, holds logical value .TRUE. or .FALSE. The value is defined as for LOGICAL.

Storage— A LOGICAL*2, item occupies 2 bytes.

Alignment— It is aligned on 2-byte boundaries.

If the –i2 compiler flag is set, then LOGICAL (without any size specification) is the same as LOGICAL*2.

LOGICAL*4 ♦

The logical data type, LOGICAL*4 holds a logical value .TRUE. or .FALSE. The value is defined as for LOGICAL.

Storage— A LOGICAL*4, item occupies 4 bytes.

Alignment— It is aligned on 4-byte boundaries.

REAL

A real datum is an approximation of a real number. The real data type, REAL, which usually has the synonym REAL*4, holds one real datum.

Storage— The usual default size for a REAL item (no size specified) is 4 bytes. If the -r8 compiler option is set, then the default size is 8 bytes; otherwise it is 4 bytes.

Alignment— It is aligned on 4-byte boundaries; except if compiled on a Sun-4 or SPARC computer with the -f option, in which case it is aligned on 8-byte boundaries.

A REAL element has a sign bit, an 8-bit exponent, and a 23-bit fraction. These REAL elements in f77 conform to the IEEE standard.

REAL*4 ♦

The REAL*4 data type is a synonym for REAL, except that it always has a size of 4 bytes, independent of any compiler options.

REAL*8 *(Double-Precision Real)* ♦

The REAL*8, data type is a synonym for DOUBLE PRECISION, except that it always has a size of 8 bytes, independent of any compiler options.

REAL*16 *(Quad Real)* ♦

(*SPARC only*) The REAL*16 data type is a quadruple-precision real.

Storage— The size for a REAL*16 item is 16 bytes.

Alignment— It is aligned on 4-byte boundaries; except if compiled on a Sun-4 or SPARC computer with the -f option, in which case it is aligned on 8-byte boundaries.

A REAL*16 element has a sign bit, a 15-bit exponent, and a 112-bit fraction. These REAL*16 elements in f77 conform to the IEEE standard for extended precision.

Size and Alignment Summary

Size and alignment of types depends on various compiler options. This table summarizes size and alignment, ignoring other aspects of types and options.

Table 2-1 Size and Alignment without `-dalign`, `–f`, `–i2`, or `–r8`

FORTRAN Type	*Size (bytes)*	*Alignment (bytes)*
BYTE	1	1
CHARACTER	1	1
CHARACTER*n	n	1
COMPLEX	8	4
COMPLEX*8	8	4
COMPLEX*16	16	4
DOUBLE COMPLEX	16	4
COMPLEX*32 (*SPARC only*)	32	4
REAL	4	4
REAL*4	4	4
REAL*8	8	4
DOUBLE PRECISION	8	4
REAL*16 (*SPARC only*)	16	4
INTEGER	4	4
INTEGER*4	4	4
INTEGER*2	2	2
LOGICAL	2	4
LOGICAL*4	2	4
LOGICAL*2	2	2
LOGICAL*1	1	1

Synonyms
`COMPLEX` ≡`COMPLEX*8`
`INTEGER` ≡ `INTEGER*4`
`LOGICAL` ≡ `LOGICAL*4`
`REAL` ≡ `REAL*4`
`DOUBLE COMPLEX`≡`COMPLEX*16`
`DOUBLE PRECISION`≡`REAL*8`

These are synonyms in the sense that `COMPLEX` is treated the same as `COMPLEX*8`, `INTEGER` is treated the same as `INTEGER*4`, etc.

`REAL*16` is sometimes called quad real. `COMPLEX*32` is sometimes called quad complex

Note that `-dalign` triggers the `-f` option.

Arrays and structures align according to their elements or fields. An array aligns the same as the array element. A structure aligns the same as the field with the widest alignment.

Table 2-2 Size and Alignment Changed by `-i2`

FORTRAN Type	*Size (bytes)*	*Alignment (bytes)*
INTEGER	2	2
LOGICAL	2	2

Changed synonyms:
INTEGER ≡ INTEGER*2
LOGICAL ≡ LOGICAL*2

Do not use `–i2` with `–i4` or `–r8`.

Table 2-3 Size and Alignment Changed by `-r8` (*SPARC only*)

FORTRAN Type	*Size (bytes)*	*Alignment (bytes)*
COMPLEX	16	4
DOUBLE COMPLEX	32	4
REAL	8	4
DOUBLE PRECISION	16	4
INTEGER	8	4
LOGICAL	8	4

Changed synonyms:
COMPLEX ≡ COMPLEX*16
INTEGER ≡ *INTEGER*8*
LOGICAL ≡ *LOGICAL*8*
REAL ≡ REAL*8
DOUBLE PRECISION ≡ REAL*16
DOUBLE COMPLEX ≡ COMPLEX*32

Do not use `–r8` with `–i2`.

In the margin note, types in *italics* are allocated the larger space indicated. This is done to maintain the FORTRAN requirement that an integer item and a real item have the same amount of storage.

That space is only partially filled, using the largest actual comparable type available, and the appropriate computation. For example, an *integer*8* item gets 8 bytes, but an `integer*4` item is stored in those 8 bytes, and only `integer*4` computations are done. These italicized types cannot be explicitly used by the programmer.

Table 2-4 Size and Alignment Changed by `-dalign` or `-f` (*SPARC only*)

FORTRAN Type	*Size (bytes)*	*Alignment (bytes)*
COMPLEX*8	8	8
COMPLEX*16	16	8
DOUBLE COMPLEX	32	8
COMPLEX*32 (*SPARC only*)	32	8
REAL*8	8	8
REAL*16 (*SPARC only*)	16	8

Note that `-dalign` triggers the `-f` option.

Types in *italics* are allocated the larger space indicated, but that space is only partially filled, using the largest actual comparable type available. For example, an *integer*8* item gets 8 bytes, but an `integer*4` is stored in those 8 bytes. This is done to maintain the FORTRAN requirement that an integer item and a real item have the same amount of storage. These italicized types cannot be explicitly used by the programmer.

2.2 Constants

A *constant* is a datum whose value cannot change throughout the program unit. The form of the string representing a constant determines the value and data type of the constant.

General kinds of constants*:*

Arithmetic
Logical
Character

Blanks— Blank characters within an arithmetic or logical constant do not affect the value of the constant. Within character constants they do affect the value.

Kinds of arithmetic constants:

Typed Constants:	*Typeless Constants:*
Complex	Binary
Double complex	Octal
Double precision	Hexadecimal
Integer	Hollerith
Real	

Sign— A *signed constant* is an arithmetic constant with a leading plus or minus sign. An *unsigned constant* is an arithmetic constant without a leading sign.

Zero— For integer, real, and double- precision data, zero is neither positive nor negative. The value of a signed zero is the same as that of an unsigned zero.

Character Constants

A character-string constant is a string of characters enclosed in apostrophes or quotes. The apostrophes are standard; the quotes are not. ♦

If you compile with the `-xl` option, then the quotes mean something else, and you must use apostrophes to enclose a string.

To include an apostrophe in an apostrophe-delimited string, repeat it. To include a quote in a quote-delimited string, repeat it. Examples

```
'abc'          "abc"
'ain''t'       "in vi type ""h9Y"
```

If a string begins with one kind of delimiter, the other kind can be embedded within it without using the repeated quote or backslash escapes. See Table 2-5.

Examples: Character constants.

```
"abc"     "abc"
"ain't"   'in vi type "h9Y'
```

Null Characters ♦

Each character string constant appearing outside a DATA statement is followed by a null character to ease communication with C routines. You can make character string *constants* consisting of no characters, but only as arguments being passed to a subprogram. Such zero length character string constants are not FORTRAN Standard.

Example. Null character string.

```
demo$ cat NulChr.f
      write(*,*) 'a', '', 'b'
      stop
      end
demo$ f77 NulChr.f
NulChr.f:
 MAIN:
demo$ a.out
ab
demo$ ■
```

However, if you put such a null character constant into a character variable, the variable will contain a blank, and have a length of at least 1 byte.

Example. Length of null character string.

```
demo$ cat NulVar.f
      character*1 x / 'a' /, y / '' /, z / 'c' /
      write(*,*) x, y, z
      write(*,*) len( y )
      end
demo$ f77 NulVar.f
NulVar.f:
 MAIN:
demo$ a.out
a c
  1
demo$ ■
```

Escape Sequences ♦

For compatibility with C usage, the following backslash escapes are recognized. If you include the escape sequence in a character string, then you get the indicated character.

Table 2-5 Backslash Escape Sequences

Escape Sequence	***Character***
\n	Newline
\r	Carriage return
\t	Tab
\b	Backspace
\f	Form feed
\v	Vertical tab
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Quotation mark (does not terminate a string)
****	\
\x	*x*, where *x* is any other character

If you compile with the `-xl` option, then the backslash character (\ ) is treated as an ordinary character. That is, with the `-xl` option, you cannot use these escape sequences to get special characters.

Technically, the escape sequences are not nonstandard but are implementation defined.

Complex Constants

A complex constant is an ordered pair of real or integer constants. The constants are separated by a comma, and the pair is enclosed in parentheses. The first constant is the real part and the second is the imaginary part. A complex constant, `COMPLEX*8`, uses 8 bytes of storage.

Examples: Complex constants.

```
( 9.01, .603 )
( +1.0, -2.0 )
( +1.0, -2 )
( 1, 2 )
( 4.51, )                    Invalid  — need second part
```

COMPLEX*16 *Constants*

A *double-complex* constant, COMPLEX*16, is an ordered pair of real or integer constants where one of the constants is REAL*8, and the other is INTEGER, REAL*4, or REAL*8. ♦

The constants are separated by a comma, and the pair is enclosed in parentheses. The first constant is the real part and the second is the imaginary part. A double-complex constant, COMPLEX*16, uses 16 bytes of storage.

Example: Double-complex constants.

```
( 9.01D6, .603 )
( +1.0, -2.0D0 )
( 1D0, 2 )
( 4.51D6, )                  Invalid  — need second part
( +1.0, -2.0 )               Not  DOUBLE  COMPLEX — need a REAL*8
```

COMPLEX*32 *(Quad Complex) Constants*

(*SPARC only*) A quad complex constant ♦ is an ordered pair of real or integer constants where one of the constants is REAL*16, and the other is INTEGER, REAL*4, REAL*8, or REAL*16.♦

The constants are separated by a comma, and the pair is enclosed in parentheses. The first constant is the real part and the second is the imaginary part. A quad complex constant, COMPLEX*32 ♦, uses 32 bytes of storage.

Examples: Quad complex constants (*SPARC only*).

```
( 9.01Q6, .603 )
( +1.0, -2.0Q0 )
( 1Q0, 2 )
( 3.3Q-4932, 9 )
( 1, 1.1Q+4932 )
( 4.51Q6, )              Invalid — need second part
( +1.0, -2.0 )           Not quad complex — need a REAL*16
```

Integer Constants

An integer constant consists of an optional plus or minus sign, followed by a string of decimal digits.

Restrictions

- No other characters are allowed, except, of course, a space.
- If no sign is present, the constant is assumed to be nonnegative.
- The value must be in the range [-2147483648, 2147483647].

Examples: Integer constants.

```
-2147483648
-2147483649              Invalid — too small, error message
-10
0
+199
29002
2.71828                  Not INTEGER — decimal point not allowed
1E6                      Not INTEGER — E not allowed
29,002                   Invalid — comma not allowed, error message
2147483647
2147483648               Invalid — too large, error message
```

Alternate Octal Notation ♦

You can also specify integer constants with the following alternate octal notation. Precede an integer string with a double quote (") and compile with the `-xl` option. These are octal constants. They are of type `INTEGER`. Example: The following two statements are equivalent.

```
      JCOUNT = ICOUNT + "703
      JCOUNT = ICOUNT + 451
```

You can also specify *typeless* constants as binary, octal, hexadecimal, or Hollerith. See “Typeless Constants (Binary, Octal, Hexadecimal)” on page 37.

Short Integers ♦

If a constant argument is in the range [-32768, 32767], it is usually widened to a 4-byte integer, data type `INTEGER*4`; but if the `–i2` option is set, then it is stored or passed as a 2-byte integer, data type `INTEGER*2`.

Logical Constants

A logical constant is either the logical value true or false. The only logical constants are `.TRUE.` and `.FALSE.`; no others are possible. The period delimiters are necessary.

A logical constant takes 4 bytes of storage. If it is an actual argument, it is passed as 4 bytes, unless the `–i2` option is set, then it is passed as 2.

Real Constants

A real constant is an approximation of a real number. It can be positive, negative, or zero. It has a decimal point and/or an exponent. If no sign is present, the constant is assumed to be nonnegative.

Real constants, `REAL*4`, use 4 bytes of storage.

Basic Real Constant

A *basic real constant* consists of an optional plus or minus sign, followed by an integer part, followed by a decimal point, followed by a fractional part.

The integer part and the fractional part are each strings of digits, and you can omit either of these parts, but not both.

Examples: Basic real constants.

```
+82.
-32.
90.
98.5
```

Real Exponent

A *real exponent* consists of the letter E, followed by an optional plus or minus sign, followed by an integer.

Examples: Real exponents.

```
E+12
E-3
E6
```

Real Constant

A *real constant* has one of these forms.

- Basic real constant
- Basic real constant followed by a real exponent
- Integer constant followed by a real exponent

A real exponent denotes a power of ten. The value of a real constant is the product of that power of ten and the constant that precedes the E.

Examples: Real constants.

```
-32.
-32.18
1.6E-9
7E3
1.6E12
$1.0E2.0              Invalid — $ not allowed, error message
82                    Not REAL — need decimal point or exponent
29,002.0              Invalid — comma not allowed, error message
1.6E39                Invalid — too large, machine infinity is used
1.6E-39               Invalid — too small, some precision is lost
```

Restrictions

- Other than the optional plus or minus sign, a decimal point, the digits 0 through 9, and the letter E, no other characters are allowed.
- The magnitude of a normalized single-precision floating-point value must be in the approximate range [1.175494E-38, 3.402823E+38].

REAL*8 *(Double-Precision Real) Constants*

A double-precision constant is an approximation of a real number. It can be positive, negative, or zero. If no sign is present, the constant is assumed to be nonnegative. A double-precision constant has a double-precision exponent and an optional decimal point. Double-precision constants, REAL*8, use 8 bytes of storage. The REAL*8 notation is nonstandard.♦

Double-Precision Exponent

A *double-precision exponent* consists of the letter D, followed by an optional plus or minus sign, followed by an integer.

A double-precision exponent denotes a power of 10. The value of a double-precision constant is the product of that power of 10 and the constant that precedes the D. The form and interpretation are the same as for a real exponent, except that a D is used instead of an E.

Examples of double-precision constants

```
1.6D-9
7D3
$1.0D2.0                    Invalid — $ not allowed, error message
82                          Not DOUBLE PRECISION — need decimal point or
exponent
29,002.0D0                  Invalid — comma not allowed, error message
1.8D308                     Invalid — too large, machine infinity is used
1.0D-324                    Invalid — too small, some precision is lost
```

Restrictions

- Other than the optional plus or minus sign, a decimal point, the digits 0 through 9, a blank, and the letter D, no other characters are allowed.
- The magnitude of an IEEE normalized double-precision floating-point value must be in the approximate range [2.225074D-308, 1.797693D+308].

REAL*16 *(Quad Real) Constants*

(*SPARC only*) A quadruple-precision constant is a *basic real constant (*see the start of the Section “Real Constants” on page 33) or an integer constant, such that it is followed by a quadruple-precision exponent. ♦

A *quadruple-precision exponent* consists of the letter `Q`, followed by an optional plus or minus sign, followed by an integer.

A quadruple-precision constant can be positive, negative, or zero. If no sign is present, the constant is assumed to be nonnegative.

Examples: Quadruple-precision constants (*SPARC only*).

```
1.6Q-9
7Q3
3.3Q-4932
1.1Q+4932
$1.0Q2.0        Invalid — $ not allowed, error message
82              Not quad — need exponent
29,002.0Q0      Invalid — comma not allowed, error message
1.6Q5000        Invalid — too large, machine infinity is used
1.6Q-5000       Invalid — too small, some precision is lost
```

The form and interpretation are the same as for a real constant, except that a `Q` is used instead of an `E`.

Summary of Restrictions

- Other than the optional plus or minus sign, a decimal point, the digits 0 through 9, a blank, and the letter `Q`, no other characters are allowed.
- The magnitude of an IEEE normalized quadruple-precision floating-point value must be in the approximate range [`3.362Q-4932`, `1.20Q+4932`].
- It occupies 16 bytes of storage.
- Each such datum is aligned on 4-byte boundaries.

Typeless Constants (Binary, Octal, Hexadecimal)

Typeless numeric constants are so named because their expressions assume data types based on how they are used. ♦

They are not converted before use. However, in f77 such constants must be distinguished from character strings.

The general form is to enclose a string of appropriate digits in apostrophes and prefix it with the letter B, O, X, or Z. The B is for binary, the O is for octal, and the X or Z are for hexadecimal.

Example. Binary, octal, and hexadecimal constants, DATA and PARAMETER.

```
      PARAMETER ( P1 = Z'1F' )
      INTEGER*2 N1, N2, N3, N4
      DATA N1 /B'0011111'/, N2/O'37'/, N3/X'1f'/, N4/Z'1f'/
      WRITE ( *, 1 ) N1, N2, N3, N4, P1
1     FORMAT ( 1X, O4, O4, Z4, Z4, Z4 )
      END
```

Above, note the edit descriptors in FORMAT statements: O for octal and Z for hexadecimal. Each of the above integer constants has the value 31 decimal.

Example: Binary, octal, and hexadecimal, other than in DATA and PARAMETER.

```
      INTEGER*4  M, ICOUNT/1/, JCOUNT
      REAL*4  TEMP
      M = ICOUNT + B'0001000'
      JCOUNT = ICOUNT + O'777'
      TEMP = X'FFF99A'
      WRITE(*,*) M, JCOUNT, TEMP
      END
```

Above, the context defines B'0001000' and O'777' as INTEGER*4 and X'FFF99A' as REAL*4. For a real number, using IEEE floating-point, a given bit pattern yields the same value on different architectures.

The above statements are treated as the following.

```
      M = ICOUNT + 8
      JCOUNT = ICOUNT + 511
      TEMP = 2.35076E-38
```

Control Characters

You can enter control characters with typeless constants, although the CHAR function is standard and this way is not.

Example: Control characters with typeless constants.

```
      CHARACTER BELL, ETX / X'03' /
      PARAMETER ( BELL = X'07' )
```

Alternate Notation for Typeless Constants

For compatibility with other versions of FORTRAN, the following alternate notation is allowed for octal and hexadecimal notation.

This alternate does not work for binary, nor does it work in DATA or PARAMETER statements.

Octal — Enclose a string of octal digits in apostrophes and append the letter O.

Examples: Octal alternate notation for typeless constants.

```
      '37'O
      37'O        Invalid — missing initial apostrophe
      '37'        Not numeric — missing letter O
      '397'O      Invalid — invalid digit
```

- *Hexadecimal* — Enclose a string of hex digits in apostrophes and *append* the letter X.
- Examples: Hex alternate notation for typeless constants.

```
      'ab'X
      3fff'X
      '1f'X
      '1fX        Invalid — missing trailing apostrophe
      '3f'        Not numeric — missing X
      '3g7'X      Invalid — invalid digit g
```

Rules and Restrictions for Binary, Octal, and Hexadecimal Constants

- These constants are for use anywhere numeric constants are allowed.
- These constants are typeless; they are stored in the variables without any conversion to match the type of the variable, but they are stored in the appropriate part of the receiving field — low end, high end.
- If the receiving data type has *more* digits than are specified in the constant, zeros are *filled on the left*.
- If the receiving data type has *fewer* digits than are specified in the constant, digits are *truncated on the left*. If nonzero digits are lost, an error message is displayed.
- Specified leading zeros are ignored.
- You can specify up to 8 bytes of data for any one constant, at least that's all that are used.
- If a typeless constant is an actual argument, it has no data type, but it is *always* 4 bytes that are passed.
- For binary constants, each digit must be 0 or 1.
- For octal constants, each digit must be in the range 0 to 7.
- For hexadecimal constants, each digit must be in the range 0 to 9 or in the range `A` to `F`, or `a` to `f`.
- Outside of DATA statements, such constants are treated as the type required by the context. If a typeless constant is used with a binary operator, it gets the data type of the other operand. (`8.0 + '37'O`)
- In `DATA` statements, such constants are treated as typeless binary, hexadecimal, or octal constants.

Hollerith Constants ♦

A Hollerith constant consists of an unsigned, nonzero, integer constant, followed by the letter `H`, followed by a string of printable characters where the integer constant designates the number of characters in the string, including any spaces and tabs.

Storage— A Hollerith constant occupies 1 byte of storage for each character.

Alignment— It is aligned on 2-byte boundaries.

The FORTRAN Standard does not have this old Hollerith notation, although the FORTRAN Standard recommends implementing the Hollerith feature in order to improve compatibility with old programs.

Hollerith data can be used in place of character-string constants. They can also be used in `IF` tests, and to initialize noncharacter variables in `DATA` statements and assignment statements, though none of these are recommended, and none are standard. These are typeless constants.

Example: Typeless constants

```
      CHARACTER C*1, CODE*2
      INTEGER TAG*2
      DATA TAG / 2Hok /
      CODE = 2Hno
      IF ( C .EQ. 1HZ ) CALL PUNT
```

Rules and Restrictions on Hollerith Constants

- The number of characters has no practical limit.
- The characters can continue over to a continuation line, but that gets tricky. Short standard fixed format lines are padded on the right with blanks up to 72 columns, but short tab-format lines stop at the newline.
- If a Hollerith constant is used with a binary operator, it gets the data type of the other operand.
- If you assign a Hollerith constant to a variable, and the length of the constant is less than the length of the data type of the variable, then spaces (ASCII 32) are appended on the right.

  If the length of a Hollerith constant or variable is greater than the length of the data type of the variable, then characters are truncated on the right.
- If a Hollerith constant is used as an actual argument, it is passed as a 4-byte item.
- If a Hollerith constant is used and the context does not determine the data type, then `INTEGER*4` is used.

2.3 Variables

A *variable* is a symbolic name paired with a storage location. A variable has a name, a value, and a type. Whatever datum is stored in the location is the value of the variable. Note that this does not include arrays or array elements, or records, or record fields, so this definition is more restrictive than the usual usage of the word “variable.”

Type— You can specify the type of a variable in a type statement. If the type is not explicitly specified in a type statement, it is implied by the first letter of the variable name: either by the usual default implied typing, or by any implied typing of `IMPLICIT` statements. see Section 2.1, “Types,” for more detail on the rules for data typing.

Defined— At any given time during the execution of a program, a variable is either *defined* or *undefined*. If a variable has a predictable value, it is defined; otherwise, it is undefined. A previously defined variable may become undefined, as when a subprogram is exited.

You can define a variable with an assignment statement, an input statement, or a `DATA` statement. If a variable is assigned a value in a `DATA` statement, then it is *initially defined*.

Associated— Two variables are associated if each is associated with the same storage location. You can associate variables by use of `EQUIVALENCE`, `COMMON`, or `MAP` statements. Actual and dummy arguments can also associate variables.

2.4 Arrays

An *array* is a named collection of elements of the same type. It is a nonempty sequence of data and occupies a group of contiguous storage locations. An array has a name, a set of elements, and a type.

An *array name* is a symbolic name for the whole sequence of data.

An *array element* is one member of the sequence of data. Each storage location holds one element of the array.

An *array element name* is an array name qualified by a subscript. See “Array Subscripts,” for details.

You can declare an array in any of the following.

- `DIMENSION` statement
- `COMMON` statement
- *Type* statements: `BYTE`, `CHARACTER`, `INTEGER`, `REAL`, ...

Array Declarators

An *array declarator* specifies the name and properties of an array.

The syntax of an array declarator is as follows.

```
a ( d [, d ] … )
```

where:

a is the name of the array, and
d is a dimension declarator.

A dimension declarator has the form

```
[ dl:] du
```

where:

dl is the **l**ower dimension bound.
du is the **u**pper dimension bound.

Dimensions— The number of dimensions in an array is the number of dimension declarators. The minimum number of dimensions is one, and the maximum is seven. For an assumed-size array, the last dimension can be an asterisk.

Bounds— The *lower bound* indicates the first element of the dimension, and the *upper bound* indicates the last element of the dimension. In a one-dimensional array, these are the first and last elements of the array.

Example: Array declarator, lower and upper bounds.

```
      REAL V(-5:5)
```

In the above example, `V` is an array of real numbers, with 1 dimension and 11 elements. The first element is `V(-5)` the last element is `V(5)`.

Example: Default lower bound of 1.

```
      REAL V(1000)
```

In the above example, V is an array of real numbers, with 1 dimension and 1000 elements. The first element is V(1) the last element is V(1000).

Example: Arrays can have as many as 7 dimensions.

```
      REAL TAO(2,2,3,4,5,6,10)
```

Example: Lower bounds other than one.

```
      REAL A(3:5, 7, 3:5), B(0:2)
```

Example: Character arrays

```
      CHARACTER M(3,4)*7, V(9)*4
```

The array M has 12 elements, each of which consists of 7 characters.
The array V has 9 elements, each of which consists of 4 characters.

Restrictions on bounds

- Both upper and lower can be negative, zero, or positive.
- The upper must be greater than or equal to the lower.
- If only one bound is specified, it is the upper, and the lower is one.
- In assumed-size arrays, the upper bound of the last dimension is an asterisk.
- Each bound is an integer expression, and each operand of the expression is a constant, a dummy argument, or a variable in a common block. No array references or user-defined functions are allowed.

Adjustable Arrays

An *adjustable array* is an array which is a dummy argument, and which has one or more of its dimensions or bounds as integer variables that are either themselves dummy arguments, or are in a common block.

You can declare adjustable arrays in the usual DIMENSION, COMMON, or type-statements. In f77 you can also declare adjustable arrays in a RECORD statement, if that RECORD statement is not inside a structure declaration block.

Example: Adjustable array bounds with arguments, and variables in common.

```
      SUBROUTINE POPUP ( A, B, N )
      COMMON / DEFS / M, L, K
      REAL A(3:5, 7, M:N), B(N+1:2*N)
```

Restrictions

- The size of an adjustable array cannot exceed the size of the corresponding actual argument.
- In the first caller of the call sequence, the corresponding array must be dimensioned with constants.

Assumed Size Arrays

An *assumed size array* is an array that is a dummy argument, and which has an asterisk as the upper bound of the last dimension.

You can declare assumed-size arrays in the usual `DIMENSION`, `COMMON`, or type-statements.

In `f77` the following extensions are allowed:

- You can declare assumed-size arrays in a `RECORD` statement, if that `RECORD` statement is not inside a structure declaration block.
- You can use an assumed size array as a unit identifier for an internal file in an I/O statement.
- You can use an assumed size array as a runtime format specifier in an I/O statement.

Example: Assumed size with upper bound of last dimension an asterisk.

```
      SUBROUTINE PULLDOWN ( A, B, C )
      INTEGER A(5, *), B(*), C(0:1, (mI2:*)
```

Restriction

An assumed-size array cannot be used in an I/O list.

Array Names with No Subscripts

An array name with no subscripts indicates the entire array, and it can appear in any of the following statements.

- `COMMON`
- `DATA`
- I/O statements
- `NAMELIST`
- `RECORD` statements
- `SAVE`
- Type statements

In an `EQUIVALENCE` statement, the array name without subscripts indicates the first element of the array.

Array Subscripts

An *array element name* is an array name qualified by a subscript.

Form of a Subscript

A subscript is a parenthesized list of subscript expressions. There must be one subscript expression for each dimension of the array.

The form of a subscript is as follows:

```
( s [, s ] … )
```

where *s* is a subscript expression. The parentheses are part of the subscript.

Example: Declare a two-by-three array with the declarator.

```
      REAL M(2,3)
```

With the above, you can assign a value to a particular element as follows:

```
      M(1,2) = 0.0
```

The above assigns 0.0 to the element in row 1, column 2, of array `M`.

Subscript Expressions

Subscript expressions have the following properties and restrictions.

- A subscript expression is an integer, real, or byte expression. (According to the FORTRAN Standard it must be an integer expression.)
- A subscript expression may contain array element references and function references.
- Evaluation of a function reference must not alter the value of any other subscript expression within the same subscript.
- Each subscript expression is an index into the appropriate dimension of the array.
- Each subscript expression must be within the bounds for the appropriate dimension of the array.
- A subscript of the form (*L1*, ..., *Ln*), where each *Li* is the *lower* bound of the respective dimension, references the first element of the array.
- A subscript of the form (*U1*, ..., *Un*), where each *Ui* is the *upper* bound of the respective dimension, references the last element of the array.
- Array element A(n) is not necessarily the n^{th} element of array A.

```
      REAL V(-1:8)
      V(2) = 0.0
```

In the above example, the fourth element of V is set to zero.

Array Ordering

Array elements are usually mentally arranged with the first subscript as the row number and the second subscript as the column number. Example:

```
      INTEGER*4 A(3,2)
```

The elements of A are usually mentally arranged like this in 3 rows and 2 columns.

A(1,1)	A(1,2)
A(2,1)	A(2,2)
A(3,1)	A(3,2)

Array elements are *stored* in column-major order.

Example: For the array A, they are located in memory as follows:

A(1,1)	A(2,1)	A(3,1)	A(1,2)	A(2,2)	A(3,2)

The inner (leftmost) subscript changes more rapidly.

2.5 Substrings

A character datum is a sequence of one or more characters. A character *substring* is a contiguous portion of a character variable or of a character array element or of a character field of a structured record.

A *substring name* can be in either of the following two forms.

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

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

where

v	Character variable name
a(s [, s] …)	Character array element name
e1	Leftmost character position of the substring
e2	Rightmost character position of the substring

Both *e1* and *e2* are integer expressions.

Example. The string with initial character from the *Ith* character of *S* and with the last character from the *Lth* character of *S*.

```
S(I:L)
```

In the above example there are *L-I+1* characters in the substring.

The following string has initial character from the *Mth* character of the array element `A(J,K)` and with the last character from the *Nth* character of that element.

```
A(J,K)(M:N)
```

In the above example there are *N-M+1* characters in the substring.

Rules and Restrictions for Substrings

- Character positions within a substring are numbered from left to right.
- The first character position is numbered 1 (not 0).
- The initial and last character positions must be integer expressions.
- If the *first* expression is omitted, it is 1.
- If the *second* expression is omitted, it is the declared length.
- The result is undefined unless $0 < I \leq L \leq$ *the declared length*, where I is the *initial* position and L is the *last* position.
- Substrings may be used on the left and right sides of assignments and as procedure actual arguments.

Examples: Substrings. The value of the element in column 2, row 3 is `e23`.

```
demo$ cat sub.f
      character  v*8 / 'abcdefgh' /,
&               m(2,3)*3 / 'e11', 'e21',
&               'e12', 'e22',
&               'e13', 'e23' /
      print *, v(3:5)
      print *, v(1:)
      print *, v(:8)
      print *, v(:)
      print *, m(1,1)
      print *, m(2,1)
      print *, m(1,2)
      print *, m(2,2)
      print *, m(1,3)
      print *, m(2,3)
      print *, m(1,3)(2:3)
      end
demo$ ■
```

Substrings example *(continued)*

```
demo$ f77 sub.f
sub.f:
 MAIN:
demo$ a.out
 cde
 abcdefgh
 abcdefgh
 abcdefgh
 e11
 e21
 e12
 e22
 e13
 e23
 13
demo$ ■
```

2.6 Structures

A *structure* is a generalization of an array. ♦

Just as an array is a collection of elements of the same type, so a structure is a collection of elements that are not necessarily of the same type.

As elements of arrays are referenced by using numeric subscripts, so elements of structures are referenced by using element (or field) names.

The structure declaration defines the form of a *record* by specifying the name, type, size, and order of the *fields* that constitute the record. Once a structure is defined and named, it can be used in RECORD statements, as explained below. The structure declaration has the following syntax.

Structure Declaration

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

structure-name	Name of the structure
field-list	List of fields of the specified structure
field-declaration	Defines a field of the record. *field-declaration* is defined below.

Field Declaration

Each field declaration can be one of the following.

- A substructure (either another structure declaration, or a record that has been previously defined)
- A *union* declaration (described below)
- A FORTRAN type declaration

Example: A STRUCTURE declaration.

```
STRUCTURE /PRODUCT/
     INTEGER*4 ID
     CHARACTER*16 NAME
     CHARACTER*8 MODEL
     REAL*4 COST
     REAL*4 PRICE
END STRUCTURE
```

In the above example, a *structure* named PRODUCT is defined to consist of the five fields ID, NAME, MODEL, COST, and PRICE. For an example with a *field-list*, see “Structure within a Structure” on page 56.

Rules and Restrictions for Structures

- The name is enclosed in slashes and is optional only in nested structures.
- If slashes are present, a name must be present.
- You can specify the *field-list* within nested structures only.
- There must be at least one *field-declaration*.
- Each *structure-name* must be unique among structures, although you can use structure names for fields in other structures or as variable names.
- The only statements allowed between the `STRUCTURE` statement and the `END STRUCTURE` statement are *field-declaration* statements and PARAMETER statements. A `PARAMETER` statement inside a structure declaration block is equivalent to one outside.

Rules and Restrictions for Fields

Fields that are type declarations use the identical syntax of normal FORTRAN type statements, and all `f77` types are allowed, subject to the following rules and restrictions:

- Any dimensioning needed must be in the type statement. The `DIMENSION` statement has no effect on field names.
- You can specify the pseudo-name `%FILL` for a field name. The `%FILL` is provided for compatibility with other versions of FORTRAN. It is not needed in `f77` because the alignment problems are taken care of for you. It might be considered a useful feature to anyone who wants to make one or more fields that you cannot reference in some particular subroutine. The only thing that `%FILL` does is provide a field of the specified size and type, and preclude referencing it.
- You must explicitly type all field names. The `IMPLICIT` statement does not apply to statements in a `STRUCTURE` declaration, nor do the implicit `I,J,K,L,M,N` rules apply.
- You cannot use arrays with adjustable or assumed size in field declarations, nor can you include passed-length `CHARACTER` declarations.

Field offsets — In a structure declaration, the offset of field *n* is the offset of the preceding field, plus the length of the preceding field, possibly corrected for any adjustments made to maintain alignment. See Appendix C, "Data Representations," for a summary of storage allocation.

Record Declaration

The RECORD statement declares variables to be records with a specified structure, or declares arrays to be arrays of such records.

The syntax of a RECORD statement is as follows.

```
RECORD /structure-name/ record-list
     [,/structure-name/ record-list]
     …
     [,/structure-name/ record-list]
```

structure-name	Name of a previously declared structure
record-list	List of variables, arrays, or arrays with dimensioning and index ranges, separated by commas.

Example: a RECORD using the previous STRUCTURE example.

```
RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
```

Each of the three variables CURRENT, PRIOR, and NEXT is a record which has the PRODUCT structure, and LINE is an array of 10 such records.

Rules and Restrictions for Records

- Each record is allocated separately in memory.
- Initially, records have undefined values, unless explicitly initialized.
- Records, record fields, record arrays, and record-array elements are allowed as arguments and dummy arguments. When you pass records as arguments, their fields must match in type, order, and dimension. The record declarations in the calling and called procedures must match. Within a union declaration, the order of the map fields is not relevant. See "Unions and Maps" on page 57.

- Records and record fields are allowed in COMMON and DIMENSION statements.
- Records and record fields are not allowed in DATA, EQUIVALENCE, NAMELIST, or SAVE statements.

Record and Field Reference

You can refer to a whole record, or to an individual field in a record, and since structures can be nested, a field can itself be a structure, so you can refer to fields within fields, within fields, and so forth.

The syntax of record and field reference is as follows.

record-name[.field-name] … [.field-name]	
record-name	Name of a previously defined record variable
field-name	Name of a field in the record immediately to the left.

Example: References (based on structure and records of above two examples)

```
…
RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
…
CURRENT = NEXT
LINE(1) = CURRENT
WRITE ( 9 ) CURRENT
NEXT.ID = 82
```

In the above example, the first assignment statement copies one whole record (all five fields) to another record, the second assignment statement copies a whole record into the first element of an array of records, the WRITE statement writes a whole record, and the last statement sets the ID of one record to 82.

Example: Structure and record declarations, record and field assignments.

```
demo$ cat str1.f
* str1.f Simple structure
     STRUCTURE / S /
          INTEGER*4 I
          REAL*4 R
     END STRUCTURE
     RECORD / S / R1, R2
     R1.I = 82
     R1.R = 2.7182818
     R2 = R1
     WRITE ( *, * ) R2.I, R2.R
     STOP
     END
demo$ f77 -silent str1.f
demo$ a.out
82 2.718280
demo$ ■
```

Substructure Declaration

A structure can have a field that is also a structure. Such a field is called a *substructure*. You can declare a substructure in either of two ways.

- A RECORD declaration within a structure declaration
- A structure declaration within a structure declaration (nesting)

Record within a Structure

A nested structure declaration is one that is contained within either a structure declaration or a union declaration. You can use a previously defined record within a structure declaration.
Example: Define structure SALE using previously defined record PRODUCT.

```
     STRUCTURE /SALE/
          CHARACTER*32  BUYER
          INTEGER*2  QUANTITY
          RECORD /PRODUCT/  ITEM
     END STRUCTURE
```

In the above example, the structure SALE contains three fields. BUYER, QUANTITY, and ITEM, where ITEM is a record with the structure /PRODUCT/.

Structure within a Structure

You can nest a declaration within a declaration.

Example: If /PRODUCT/ is *not* declared previously, then you can declare it within the declaration of SALE.

```
      STRUCTURE /SALE/
           CHARACTER*32  BUYER
           INTEGER*2  QUANTITY
           STRUCTURE /PRODUCT/ ITEM
                INTEGER*4  ID
                CHARACTER*16  NAME
                CHARACTER*8  MODEL
                REAL*4  COST
                REAL*4  PRICE
           END STRUCTURE
      END STRUCTURE
```

Here the structure SALE still contains the same three fields as in the prior example: BUYER, QUANTITY, and ITEM. The field ITEM is an example of a *field-list* (in this case, a single-element list), as defined under "Structure Declaration."

The size and complexity of the various structures determine which style of substructure declaration is best to use in a given situation.

Field Reference in Substructures

You can refer to fields within substructures.

Example: Refer to fields of substructures (PRODUCT and SALE, from previous examples, are defined in current program unit).

```
      …
      RECORD /SALE/ JAPAN
      …
      N = JAPAN.QUANTITY
      I = JAPAN.ITEM.ID
      …
```

Rules and restrictions for substructures

- You must define at least one field name for any substructure.
- No two fields at the same nesting level can have the same name. Fields at different levels of a structure can have the same name (although doing so might be questionable programming practice).
- You can use the pseudo-name `%FILL` to align fields in a record. This makes an unnamed empty field.
- You must not include a structure as a substructure of itself, at any level of nesting.

Unions and Maps

A *union* declaration defines groups of fields that share memory at runtime.

The syntax of a union declaration is as follows.

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

The syntax of a map declaration is as follows.

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

Fields in a Map

Each *field-declaration* in a *map* declaration can be one of the following.

- Structure declaration
- Record
- Union declaration
- Declaration of a typed data field

A *map* declaration defines alternate groups of fields in a union. During execution, one map at a time is associated with a shared storage location. When you reference a field in a map, the fields in any previous map become undefined and are succeeded by the fields in the map of the newly referenced field. The amount of memory used by a union is that of its biggest map.

Example: Declare the structure /STUDENT/ to contain either NAME, CLASS, and MAJOR — or NAME, CLASS, CREDITS, and GRAD_DATE.

```
      STRUCTURE /STUDENT/
           CHARACTER*32  NAME
           INTEGER*2  CLASS
           UNION
                MAP
                        CHARACTER*16 MAJOR
                END MAP
                MAP
                        INTEGER*2  CREDITS
                        CHARACTER*8  GRAD_DATE
                END MAP
           END UNION
      END STRUCTURE
```

If you define the variable PERSON to have the structure /STUDENT/ from the above example, then PERSON.MAJOR references a field from the first map, and PERSON.CREDITS references a field from the second map. If the variables of the second map field are initialized and then the program references the variable PERSON.MAJOR, the first map becomes active and the variables of the second map become undefined.

2.7 Pointers

The POINTER statement establishes pairs of variables and pointers. ♦ Each pointer contains the address of its paired variable.

Syntax

The POINTER statement has the following syntax:

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

where

v1, *v2* are pointer-based variables.
p1, *p2* are the corresponding pointers.

A *pointer-based variable* is a variable paired with a pointer in a POINTER statement. A pointer-based variable is usually called just a *based variable*. The *pointer* is the integer variable that contains the address.
Example: A simple POINTER statement.

```
POINTER ( P, V )
```

Here, V is a pointer-based variable, and P is its associated pointer.

Usage of Pointers

Normal use of pointer-based variables involves the following steps (the first two steps can be in either order).

1. Define the pairing of the pointer-based variable and the pointer in a POINTER statement.
2. Define the type of the pointer-based variable. The pointer itself is integer type, but in general, it is safer if you *not* list it in an INTEGER statement.
3. Set the pointer to the address of an area of memory that has the appropriate size and type. You do *not* normally do anything else explicitly with the pointer.
4. Reference the pointer-based variable. Just use the pointer-based variable in normal FORTRAN statements – the address of that variable will always be taken from its associated pointer.

Address and Memory

Note that no storage for the variable is allocated when a pointer-based variable is defined, so it is *your* responsibility to provide an address of a variable of the appropriate type and size, and assign the address to a pointer, usually with the normal assignment statement or data statement.

Address by LOC() *Function*

You can obtain the address from the intrinsic function LOC().

Example: Use the LOC() function to get an address.

```
* ptr1.f: Assign an address via LOC()
      POINTER ( P, V )
      CHARACTER A*12, V*12
      DATA A / 'ABCDEFGHIJKL' /
      P = LOC( A )
      PRINT *, V(5:5)
      END
```

In the above example, the CHARACTER statement allocates 12 bytes of storage for A, but *no* storage for V; it merely specifies the type of V because V is a pointer-based variable. Then assign the address of A to P so now any use of V will refer to A by the pointer P. The program will print an E.

Memory and Address by MALLOC() *Function*

The function MALLOC() allocates an area of memory and returns the address of the start of that area. The argument to the function is an integer specifying the amount of memory to be allocated, in bytes. If successful, it returns a pointer to the first item of the region, otherwise it returns an integer 0. The region of memory is not initialized in any way — assume it is garbage.

Example: Memory allocation for pointers, by MALLOC

```
      COMPLEX Z
      REAL X, Y
      POINTER ( P1, X ), ( P2, Y ), ( P3, Z )
      …
      P1 = MALLOC ( 10000 )
      …
```

In the above example, we get 10,000 bytes of memory from MALLOC() and assign the address of that block of memory to the pointer P1.

Deallocate Memory by FREE() *Function*

The subroutine FREE() deallocates a region of memory previously allocated by MALLOC(). The argument given to FREE() must be a pointer previously returned by MALLOC(), but not already given to FREE(). The memory is returned to the memory manager, making it unavailable to the programmer.

Example: Deallocate via FREE.

```
POINTER ( P1, X ), ( P2, Y ), ( P3, Z )
…
P1 = MALLOC ( 10000 )
…
CALL FREE ( P1 )
…
```

Above, after getting memory via MALLOC(), and after some other instructions, probably using that chunk of memory, we tell FREE() to return those same 10,000 bytes to the memory manager.

Restrictions

- The pointers are of type integer, and are automatically typed that way by the compiler. You must *not* type them yourself.
- A pointer-based variable cannot itself be a pointer.
- The pointer-based variables can be of any type, including structures.
- No storage is allocated when such a pointer-based variable is declared, even if there is a size specification in the type statement.
- You cannot use a pointer-based variable as a dummy argument or in COMMON, EQUIVALENCE, DATA, or NAMELIST statements.
- The dimension expressions for pointer-based variables must be constant expressions in main programs. In subroutines and functions, the same rules apply for pointer-based array variables as for dummy arguments — the expression can contain dummy arguments and variables in common. Any variables in the expressions must be defined with an integer value at the time the subroutine or function is called.

Optimization and Pointers

Pointers have the annoying side effect of reducing the assumptions that the global optimizer can make. For one thing, compare the following:

- Without pointers, if you call a subroutine or function, the optimizer knows that the call will change only variables in common or those passed as arguments to that call.
- With pointers, this is no longer valid, since a routine can take the address of an argument and save it in a pointer in common for use in a subsequent call to itself or to another routine.

Therefore, the optimizer must assume that a variable passed as an argument in a subroutine or function call can be changed by any other call. Such an unrestricted use of pointers would degrade optimization for the vast majority of programs that do *not* use pointers.

General Guidelines

There are two alternatives for optimization with pointers.

- Do not use pointers with optimization level `-O3` or `-O4`.
- Use a pointer only to identify the location of the data for calculations and pass the pointer to a subprogram. Almost anything else you do to the pointer can yield incorrect results.

  The second choice also has a suboption: localize pointers to one routine and do not optimize it, but do optimize the routines that do the calculations. If you put the calling the routines on different files, you can optimize one and not optimize the other.

Example: A relatively safe kind of coding with -O3 or -O4.

```
      REAL A, B, V(100,100) ! Within this programming unit,
      POINTER ( P, V )      ! do nothing else with P
      P = MALLOC(10000)     ! other than getting the address and passing it.
      …
      CALL CALC ( P, A )
      ...
      END

      SUBROUTINE CALC ( ARRAY, X )
      ...
      RETURN
      END
```

If you want to optimize only CALC at level -O3 or -O4, then use no pointers in CALC.

Some of the Many things that Cause Trouble

Any of the following coding practices (and many others) could cause problems with an optimization level of -O3 or -O4.

- A program unit does arithmetic with the pointer.
- A subprogram saves the address of any of its arguments between calls.
- A function returns the address of any of its arguments. (Although it can return the value of a pointer argument.)
- A variable is referenced through a pointer, but the address of the variable is not explicitly taken with the LOC() or MALLOC() functions.

Example: One kind of code that could cause trouble with -O3 or -O4.

```
      COMMON A, B, C
      POINTER ( P, V )
      P = LOC(A) + 4        ! ←possible problems if optimized
      …
```

The compiler will assume that a reference through P may change A, but not B; this assumption could produce incorrect code.

2

Expressions 3

This chapter is organized into the following sections.

3.1 Introduction

An *expression* is a combination of one or more operands, zero or more operators, and zero or more pairs of parentheses.

There are four kinds of expressions:

- Arithmetic
- Character
- Relational
- Logical

3

An *arithmetic expression* evaluates to a single arithmetic value.
A *character expression* evaluates to a single value of type character.
A *logical* or *relational expression* evaluates to a single logical value.

The *operators* indicate what action or operation to perform.

The *operands* indicate what items to apply the action to.
An operand can be any of the following kinds of data items:

- Constant
- Variable
- Array element
- Function
- Substring
- Structured record field (if it evaluates to a scalar data item)

3.2 Arithmetic Expressions

An *arithmetic expressio*n evaluates to a single arithmetic value, and its operands have the following types. The ♦ indicates a nonstandard feature.

- `BYTE` ♦
- `COMPLEX`
- `COMPLEX*32` *(SPARC only)* ♦
- `DOUBLE COMPLEX` ♦
- `DOUBLE PRECISION`
- `INTEGER`
- `LOGICAL`
- `REAL`
- `REAL*16` *(SPARC only)* ♦

The operators for an *arithmetic expression* are any of the following.

Table 3-1 Arithmetic Operators

Operator	***Meaning***
**	Exponentiation
*	Multiplication
/	Division
-	Subtraction or Unary Minus
+	Addition or Unary Plus

If BYTE or LOGICAL operands are combined with arithmetic operators, they are interpreted as integer data.

Each of these operators is a *binary* operator in an expression of the form

a ⊕ *b*

where *a* and *b* are operands, and ⊕ is any one of the **, *, /, -, or + operators.

Examples: Binary operators.

```
A-Z
X*B
```

The operators + and - are *unary* operators in an expression of the form

⊕ *b*

where *b* is an operand, and ⊕ is either of the - or + operators.

Examples: Unary operators.

```
-Z
+B
```

Basic Arithmetic Expressions

Each arithmetic operator is shown in its basic expression below.

Table 3-2 Arithmetic Expressions

Expression	***Meaning***
a ** z	Raise a to the power z
a / z	Divide a by z
a * z	Multiply a by z
a - z	Subtract z from a
-z	Negate z
a + z	Add z to a
+z	Same as z

In the absence of parentheses, if there is more than one operator in an expression, then the operators are applied in the order of precedence. With one exception, if the operators are of equal precedence they are applied left to right.

Table 3-3 Arithmetic Operator Precedence

Operator	***Precedence***
`**`	First
`* /`	Second
`+ -`	Last

For the left-to-right rule, the one exception is shown by the following example.

```
F ** S ** Z
```

The above is evaluated as

```
F ** (S ** Z)
```

Two successive operators

`f77` allows two successive operators. ♦

Example: Two successive operators.

```
      X ** -A * Z
```

The above expression is evaluated as follows.

```
      X ** (-(A * Z))
```

In the above example, the compiler starts to evaluate the `**`, but it needs to know what power to raise `X` to; so it looks at the rest of the expression and must choose between `-` and `*`; so it first does the `*`, then the `-`, then the `**`. Some early releases of this FORTRAN incorrectly interpreted "`X**-A*Z`" as "`(X**(-A))*Z.`" Current releases correctly interpret "`X**-A*Z`" as "`X**(-(A*Z)),`" which is compatible with VMS FORTRAN.

Example: Two successive operators.

```
demo$ cat twoops.f
      REAL X / 2.0 /, A / 1.0 /, Z / -3.0 /
      PRINT *, "X**-A*Z = ", X ** -A*Z
      PRINT *, "X**(-(A*Z)) = ", X ** (-(A*Z))
      PRINT *, "(X**(-A))*Z = ", (X ** (-A))*Z
      PRINT *, "X**-2 = ", X ** -2 !{same in both}
      END
demo$ f77old twoops.f      {Use old}
twoops.f:
 MAIN:
demo$ a.out
X**-A*Z      = -1.50000
X**(-(A*Z))  = 8.00000
(X**(-A))*Z  = -1.50000
X**-2        = 0.250000
demo$ f77new -silent twoops.f      {Use new}
demo$ a.out
X**-A*Z      = 8.00000
X**(-(A*Z))  = 8.00000
(X**(-A))*Z  = -1.50000
X**-2        = 0.250000
demo$ ■
```

Mixed Mode

If both operands have the same type, then the resulting value has that type. If operands have different types, then the weaker of two types is promoted to the stronger type, where the weaker type is the one with less precision or fewer storage units. This is summarized in the following ranking.

Data Type	***Rank***
`BYTE or LOGICAL*1`	`1` (Weakest)
`LOGICAL*2`	`2`
`LOGICAL*4`	`3`
`INTEGER*2`	`4`
`INTEGER*4`	`5`
`REAL*4 (REAL)`	`6`
`REAL*8 (DOUBLE PRECISION)`	`7`
`REAL*16 (QUAD PRECISION)`	`8`
`COMPLEX*8 (COMPLEX)`	`9`
`COMPLEX*16 (DOUBLE COMPLEX)`	`10`
`COMPLEX*32 (QUAD COMPLEX)`	`11` (Strongest)

In the above table, `REAL*16` and `COMPLEX*32` are for *SPARC only.*

Example of mixed mode: If `R` is real, and `I` is integer, then the expression

```
R * I
```

has the type real, because first `I` is promoted to real, and then the multiplication is performed.

Rules for the Data Type of an Expression

- If there is more than one operator in an expression, then the type of the last operation performed becomes the type of the final value of the expression.
- Integer operators apply to only integer operands.

  Example: The expression below evaluates to zero.

```
2/3 + 3/4
```

  There is one extension to this: a logical or byte operand in an arithmetic context is used as an integer.

- Real operators apply to only real operands, or to combinations of byte, logical, integer, and real operands. An integer operand mixed with a real operand is promoted to real; the fractional part of the new real number is zero. For example, if `R` is real and `I` is integer, then `R+I` is real. But note that `(2/3)*4.0 is 0.`
- Double precision operators apply to only double precision operands, and any operand of lower precision is promoted to double precision. The new least significant bits of the new double precision number are set to zero. Promoting a real operand does not increase the accuracy of the operand.
- Complex operators apply to only complex operands. Any integer operands are promoted to real, and they are then used as the real part of a complex operand, with the imaginary part set to zero.
- Numeric operations are allowed on logical variables. ♦
  You can use a logical value any place where the FORTRAN Standard requires a numeric value. The numeric can be `integer`, `real`, `complex`, `double precision`, `double complex`, or `real*16` *(SPARC only)*. The compiler implicitly converts the logical to the appropriate numeric. Logical operations are allowed on integers, bytes, and characters. If you use these features, your program may not be portable.

  Example: Some combinations of both `integer` and `logical` types.

```
      COMPLEX  C1 / ( 1.0, 2.0 ) /
      INTEGER*2 I1, I2, I3
      LOGICAL L1, L2, L3, L4, L5
      REAL R1 / 1.0 /
      DATA I1 / 8 /, I2 / 'W' /, I3 / 0 /
      DATA L1/.TRUE./, L2/.TRUE./, L3/.TRUE./, L4/.TRUE./,
&           L5/.TRUE./
      L1 = L1 + 1
      I2 = .NOT. I2
      L2 = I1 .AND. I3
      L3 = I1 .OR. I2
      L4 = L4 + C1
      L5 = L5 + R1
```

Resultant Type

- For integer operands with a logical operator, the operation is done bit-by-bit. The result is an integer.
- If the operands are mixed integer and logical, then the logicals are converted to integers and the result is an integer.

Arithmetic Assignment

The arithmetic assignment statement assigns a value to a variable, array element, record, or record field. The syntax is:

```
v = e
```

e	Arithmetic expression, a character constant, or a logical expression
v	Numeric variable, array element, record, or record field

Assigning logicals to numerics is allowed, but nonstandard, and may not be portable. The resultant data type is, of course, the data type of *v*. ♦

Execution of an arithmetic assignment statement causes the evaluation of the expression *e*, and conversion to the type of *v* (if types differ), and assignment of *v* with the resulting value typed according to the table below.

Character constants can be assigned to variables of type integer or real. Such a constant can be a Hollerith constant or a string in apostrophes or quotes. The characters are transferred to the variables without any conversion of data. This is nonstandard and may not be portable. ♦

Type of v	***Type of e***
`INTEGER*2 or INTEGER*4`	`INT (e)`
`REAL`	`REAL (e)`
`REAL*8`	`DBLE (e)`
`REAL*16 (SPARC only)`	`QREAL (e) (SPARC only)`
`DOUBLE PRECISION`	`DBLE (e)`
`COMPLEX*8`	`CMPLX (e)`
`COMPLEX*16`	`DCMPLX (e)`
`COMPLEX*32 (SPARC only)`	`QCMPLX (e) (SPARC only)`

Example: Arithmetic assignment.

```
INTEGER I2*2, J2*2, I4*4
LOGICAL L1, L2
REAL R4*4, R16*16 ! (The *16 is for SPARC only)
DOUBLE PRECISION DP
COMPLEX C8, C16*16
J2 = 29002
I2 = J2
I4 = (I2 * 2) + 1
DP = 6.4D0
QP = 9.8Q1
R4 = DP
R16 = QP
C8 = R1
C8 = ( 3.0, 5.0 )
I2 = C8
C16 = C8
C8 = L1
R4 = L2
```

3.3 Character Expressions

A *character expression* is an expression whose operands have the character type. A character expression evaluates to a single value of type character, with a size of one or more characters. The only character operator is the concatenation operator `//`.

Expression	***Meaning***
a `//` *z*	Concatenate *a* with *z*

The result of *concatenating* two strings is a third string containing the characters of the left operand followed immediately by the characters of the right operand. The value of a concatenation operation `a//z` is a character string whose value is the value of `a` concatenated on the right with the value of `z`, and whose length is the sum of the lengths of `a` and `z`.

The operands can be any of the following kinds of data items.

- Character constant
- Character variable
- Character array element
- Character function
- Substring
- Structured record field (if it evaluates to a scalar character data item)

Examples: Character expressions (assumes `C`, `S`, and `R.C` are character).

```
'wxy'
'AB' // 'wxy'
C
C // S
C(4:7)
R.C
```

- Control Characters ♦

  One way to enter control characters is to hold down the control key and press another key. Most control characters can be entered this way, but not control-A, control-B, control-C, or control-J.

  Example: A valid way to enter a control-C.

```
CHARACTER etx
etx = CHAR(3)
```

- Multiple Byte Characters ♦

  Multiple byte characters (such as Kanji) are allowed in comments and strings.

Character String Assignment

The form of the character string assignment is

v = *e*	
e	Expression giving the value to be assigned
v	Variable, array element, or substring

The meaning of character assignment is to copy characters from the right to the left side.

Execution of a character assignment statement causes evaluation of the character expression and assignment of the resulting value to *v*.

- If *e* is longer than *v*, characters on the right are truncated.
- If *e* is shorter than *v*, blank characters are padded on the right.

Example: The program below displays "`joined`ΔΔ".

```
      CHARACTER A*4, B*2, C*8
      A = 'join'
      B = 'ed'
      C = A // B
      PRINT *, C
      END
```

Also, the program below displays the "`equal`" string.

```
      IF ( ('ab' // 'cd') .EQ. 'abcd' ) PRINT *, 'equal'
      END
```

Example: Character assignment.

```
      CHARACTER BELL*1, C2*2, C3*3, C5*5, C6*6
      REAL Z
      C2 = 'z'
      C3 = 'uvwxyz'
      C5 = 'vwxyz'
      C5(1:2) = 'AB'
      C6 = C5 // C2
      I = 'abcd'
      Z = 'wxyz'
      BELL = CHAR(7)   ! Control Character (^G)
```

The results of the above are as follows.

C2	gets	'zΔ'	A trailing blank
C3	gets	'uvw'	
C5	gets	'ABxyz'	
C6	gets	'ABxyzz'	That is, the 'z' from C2
I	gets	'abcd'	
Z	gets	'wxyz'	
BELL	gets	07 hex	Control-G, a bell

Example 4: Hollerith assignment. ♦

```
      CHARACTER S*4
      INTEGER I2*2, I4*4
      REAL R
      S = 4Hwxyz
      I2 = 2Hyz
      I4 = 4Hwxyz
      R = 4Hwxyz
```

Rules for character assignment

- If the left side is longer than the right, it is padded with trailing blanks.
- If the left side is shorter than the right, trailing characters are discarded.
- The left and right sides of a character assignment may share storage. ♦

Example: The following program displays abcefggh.

```
      CHARACTER S*8
      S = 'abcdefgh'
      S(4:6) = S(5:7)
      WRITE(*,*) S
      END
```

3.4 Logical Expressions

A *logical expression* is a sequence of one or more logical operands and logical operators. It evaluates to a single logical value.The operators can be any of the following.

Table 3-4 Logical Operators

Operator	***Standard Name***
`.AND.`	Logical conjunction
`.OR.`	Logical disjunction (Inclusive OR)
`.NEQV.`	Logical nonequivalence
`.XOR.`	Logical exclusive OR
`.EQV.`	Logical equivalence
`.NOT.`	Logical negation

The period delimiters are necessary.

Two logical operators cannot appear consecutively, unless the *first* one is the `.NOT.` operator.

Logical operators are evaluated according to the following precedence.

Table 3-5 Logical Operator Precedence

Operator	***Precedence***
`.NOT.`	First
`.AND.`	
`.OR.`	
`.NEQV.,.XOR., .EQV.`	Last

If they are of equal precedence, they are evaluated left to right.

If they appear along with the various other operators in a logical expression, the precedence is as follows.

Table 3-6 Operator Precedence

Operator	***Precedence***
Arithmetic	First
Character	Second
Relational	Third
Logical	Last

The meaning of simple expressions is as follows.

Table 3-7 Logical Expression Meanings

Expression	***Meaning***
`X .AND. Y`	Both X and Y are true.
`X .OR. Y`	Either X or Y, or both, are true.
`X .NEQV. Y`	X and Y are not both true and not both false.
`X .XOR. Y`	Either X or Y is true, but not both.
`X .EQV. Y`	X and Y are both true or both false.

Logical Assignment

This assigns the value of a logical expression to a logical variable.

```
v = e
```

where

e	A logical expression, or an integer between –128 and 127, or a single character constant
v	A logical variable, array element, record, or record field

Execution of a logical assignment statement causes evaluation of the logical expression *e* and assignment of the resulting value to *v*. Note that if *e* is a logical expression (rather than an integer between -128 and 127, or a single character constant), then *e* must have a value of either true or false.

Logical expressions of any size can be assigned to logical variables of any size.

Assigning numerics to logicals is allowed, but nonstandard, and may not be portable. ♦

Example: Logical assignment.

```
      LOGICAL B1*1, B2*1
      LOGICAL L3, L4
      B2 = B1
      B1 = L3
      L4 = .TRUE.
```

3.5 Relational Operator

A *relational operator* compares two arithmetic expressions, or two character expressions, and evaluates to a single logical value. The operators can be any of the following.

Table 3-8 Relational Operators

Operator	***Meaning***
`.LT.`	Less than
`.LE.`	Less than or equal
`.EQ.`	Equal
`.NE.`	Not equal
`.GT.`	Greater than
`.GE.`	Greater than or equal

The period delimiters are necessary.

All relational operators have equal precedence. Character and arithmetic operators have higher precedence than relational operators.

For a relational expression, first each of the two operands is evaluated, and then the two values are compared. If the specified relationship holds, then the value is true, otherwise it is false.

Example: Relational operators.

```
NODE .GE. 0
X .LT. Y
U*V .GT. U-V
M+N .GT. U-V        Mixed mode: integer M+N is promoted to real
STR1 .LT. STR2      where STR1 and STR2 are type character
S .EQ. 'a'          where S is type character
```

For character relational expressions:

- "Less than" means "precedes in the ASCII collating sequence."
- If one operand is shorter than the other, the shorter one is padded on the right with blanks to the length of the longer.

3.6 Constant Expressions

A *constant expression* is made up of explicit constants and parameters and the FORTRAN operators. Each operand is either itself another constant expression, a constant, a symbolic name of a constant, or one of the following intrinsic functions.

```
LOC, CHAR,
AND, OR, NOT, XOR, LSHIFT, RSHIFT, LGE, LGT, LLE, LLT,
MIN, MAX, ABS, MOD, ICHAR, NINT, DIM,
DPROD, CMPLX, CONJG, AIMAG
```

The functions IAND, IOR, IEOR, and ISHFT are also available, or you can use the corresponding AND, OR, XOR, LSHIFT, or RSHIFT.

Examples: Constant expressions.

```
PARAMETER (L=29002), (P=3.14159), (C='along the ')
PARAMETER ( I=L*2, V=4.0*P/3.0, S=C//'riverrun' )
PARAMETER ( M=MIN(I,L), IA=ICHAR('A') )
PARAMETER ( Q=6.4Q6, D=2.3D9 )
K = 66 * 80
VOLUME = V*10**3
DO I = 1, 20*3
```

There are few restrictions on constant expressions.

- Constant expressions are permitted wherever a constant is allowed, except they are *not* allowed in DATA or standard FORMAT statements.
- Constant expressions are permitted in variable format expressions. ♦
- Exponentiation to a floating-point power is not allowed, and a warning is issued.

  Example: Exponentiation to a floating-point power not allowed.

```
demo$ cat ConstExpr.f
     parameter (T=2.0*(3.0**2.5))
     write(*,*) t
     end
demo$ f77 ConstExpr.f
ConstExpr.f:
 MAIN:
"ConstExpr.f", line 1: Warning:
          parameter t set to a nonconstant
demo$ a.out
    31.1769
demo$ ■
```

3.7 Record Assignment

The general form of record assignment is as follows. ♦

```
     v =  e
```

where

e	A record or record field
v	A record or record field

Both *e* and *v* must have the same structure. That is, each must have the same number of fields, and corresponding fields must be of the same type and size.

Example: Record assignment and record-field assignment.

```
      STRUCTURE /PRODUCT/
           INTEGER*4 ID
           CHARACTER*16 NAME
           CHARACTER*8 MODEL
           REAL*4 COST
           REAL*4 PRICE
      END STRUCTURE
      RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
      …
      CURRENT = NEXT
      LINE(1) = CURRENT
      WRITE ( 9 ) CURRENT
      NEXT.ID = 82
```

In the above example, the first assignment statement copies one whole record (all five fields) to another record, the second assignment statement copies a whole record into the first element of an array of records, the `WRITE` statement writes a whole record, and the last statement sets the `ID` of one record to 82.

3.8 Evaluation of Expressions

The following restrictions apply to all arithmetic, character, relational, and logical expressions.

- If you reference any one of these items in an expression, variable, array element, character substring, record field, pointer, or function, then that item must be defined at the time the reference is executed.
- An integer operand must be defined with an integer value, and not with a statement label value by an `ASSIGN` statement.
- All the characters of a substring that are referenced must be defined at the time the reference is executed.
- The execution of a function reference must not alter the value of any other entity within the same statement.
- The execution of a function reference must not alter the value of any entity in common that effects the value of any other function reference in the same statement.

Statements 4

This chapter describes the FORTRAN statements. The nonstandard statements are indicated with a small black diamond (♦).

4.1 ACCEPT

The ACCEPT ♦ statement reads from standard input.

Syntax

ACCEPT *f* [, *iolist*]	
ACCEPT *grname*	
f	Format identifier
iolist	List of variables, substrings, arrays, and records
grname	Name of the namelist group

Description

ACCEPT *f* [,*iolist*] is equivalent to READ *f* [,*iolist*] and is for compatibility with older versions of FORTRAN. Example: List-directed input.

```
      REAL VECTOR(10)
      ACCEPT *, NODE, VECTOR
```

4.2 ASSIGN

The ASSIGN statement assigns a statement label to a variable.

Syntax

ASSIGN *s* TO *i*	
s	Statement label
i	Integer variable

Description

The label *s* is the label of an executable statement or a FORMAT statement.

The statement label must be the label of a statement that is defined in the same program unit as the ASSIGN statement.

The integer variable *i*, once assigned a statement label, can be reassigned the same statement label, a different label, or an integer.

Once a variable is defined as a statement label, you can reference in:

- An assigned GO TO statement
- An input/output statement, as a format identifier
- A routine call, as a nonstandard return

Restrictions

- Define a variable with a statement label *before* you reference it as a label.
- While *i* is defined with a statement label value, do *no* arithmetic with *i*.

Examples

Example 1: Assign the statement number of an executable statement.

```
      ASSIGN 9 TO K
      GO TO K
      …
 9    WRITE (*,*) 'Assigned ', K, ' to K'
```

Above, the output shows the address, not 9.

Example 2: Assign the statement number of a format statement.

```
      INTEGER PHORMAT
 2    FORMAT ( A80 )
      ASSIGN 2 TO PHORMAT
      …
      WRITE ( *, PHORMAT ) 'Assigned a FORMAT statement no.'
```

4.3 Assignment

The assignment statement assigns a value to a variable, substring, array element, record, or record field.

Syntax

v = e	
e	Expression giving the value to be assigned
v	Variable, substring, array element, record, or record field

Description

The value can be a constant or the result of an expression. The kinds of assignment statements are arithmetic, logical, character, and record assignments.

Arithmetic Assignment

v is of numeric type and is the name of a variable, array element, or record field.

e is an arithmetic expression, a character constant, or a logical expression. Assigning logicals to numerics is nonstandard, and may not be portable; the resultant data type is, of course, the data type of *v*. ♦

Execution of an arithmetic assignment statement causes the evaluation of the expression *e*, and conversion to the type of *v* (if types differ), and assignment of *v* with the resulting value typed according to the table below.

Table 4-1 Arithmetic Assignment Conversion Rules

Type of v	***Type of e***
`INTEGER*2 or INTEGER*4`	`INT(`*e*`)`
`REAL`	`REAL(`*e*`)`
`DOUBLE PRECISION`	`DBLE(`*e*`)`
`REAL*16` *(SPARC only)*	`QREAL(`*e*`)` *(SPARC only)*
`COMPLEX*8`	`CMPLX(`*e*`)`
`COMPLEX*16`	`DCMPLX(`*e*`)`
`COMPLEX*32` *(SPARC only)*	`QCMPLX(`*e*`)` *(SPARC only)*

Example: Assignment statement

```
      REAL A, B
      DOUBLE PRECISION V
      V = A * B
```

The above code is compiled exactly as if it were the following:

```
      REAL A, B
      DOUBLE PRECISION V
      V = DBLE( A * B )
```

Logical Assignment

v is the name of a variable, array element, or record field of type logical.

e is a logical expression, or an integer between -128 and 127, or a single character constant.

Execution of a logical assignment statement causes evaluation of the logical expression *e* and assignment of the resulting value to *v*. Note that if *e* is a logical expression (rather than an integer between -128 and 127, or a single character constant), then *e* must have a value of either true or false.

Logical expressions of any size can be assigned to logical variables of any size. The section on the `LOGICAL` statement has more on the size of logical variables.

Character Assignment

Character constants can be assigned to variables of the following type:

- `BYTE` ♦
- `CHARACTER`
- `INTEGER` ♦
- `REAL` ♦
- `DOUBLE PRECISION` ♦
- quadruple precision ♦
- `LOGICAL` ♦

The constant can be a Hollerith constant or a string of characters delimited by apostrophes (') or quotes ("). The character string *cannot* include the control characters control-A, control-B, or control-C; that is, you cannot hold down the control key and press the A, B, or C keys. If you need those control characters, use the `char()` function.

If you use quotes to delimit a character constant, then you cannot compile with the `-xl` option, because in that case a quote introduces an octal constant. The characters are transferred to the variables without any conversion of data. This may not be portable.

Character expressions which include the `//` operator can be assigned only to items of type `CHARACTER`. Here the *v* is the name of a variable, substring, array element, or record field of type `CHARACTER`. *e* is a character expression.

Execution of a character assignment statement causes evaluation of the character expression and assignment of the resulting value to *v*. If the length of *e* is more than that of *v*, characters on the right are truncated. If the length of *e* is less than that of *v*, blank characters are padded on the right.

Record Assignment

v and *e* are each a record or record field. ♦

The *e* and *v* must have the same structure. They have the same structure if any of the following occur:

1. Both *e* and *v* are fields with the same elementary data type.
2. Both *e* and *v* are structures with the same number of fields such that corresponding fields are the same elementary data type.

3. Both *e* and *v* are structures with the same number of fields such that corresponding fields are substructures with the same structure as defined in 2, above.

The sections on the RECORD and STRUCTURE statements have more on the structure of records.

Examples

Example 1: Arithmetic assignment.

```
      INTEGER I2*2, J2*2, I4*4
      REAL R1, QP*16 ! (The *16 is for SPARC only)
      DOUBLE PRECISION DP
      COMPLEX C8, C16*16, QC*32  ! (The *32 is for SPARC only)
      J2 = 29002
      I2 = J2
      I4 = (I2 * 2) + 1
      DP = 6.4D9
      QP = 6.4Q9
      R1 = DP
      C8 = R1
      C8 = ( 3.0, 5.0 )
      I2 = C8
      C16 = C8
      C32 = C8
```

Example 2: Logical assignment.

```
      LOGICAL B1*1, B2*1
      LOGICAL L3, L4
      L4 = .TRUE.
      B1 = L4
      B2 = B1
```

Example 3: Hollerith assignment.

```
      CHARACTER S*4
      INTEGER I2*2, I4*4
      REAL R
      S = 4Hwxyz
      I2 = 2Hyz
      I4 = 4Hwxyz
      R = 4Hwxyz
```

Example 4: Character assignment.

```
CHARACTER BELL*1, C2*2, C3*3, C5*5, C6*6
REAL Z
C2 = 'z'
C3 = 'uvwxyz'
C5 = 'vwxyz'
C5(1:2) = 'AB'
C6 = C5 // C2
I = 'abcd'
Z = 'wxyz'
BELL = CHAR(7)   ! Control Character (^G)
```

The results of the above are

C2	gets	'zΔ'	*That is, a trailing blank*
C3	gets	'uvw'	
C5	gets	'ABxyz'	
C6	gets	'ABxyzz'	*That is, an extra* z *left over from* C5
I	gets	'abcd'	
Z	gets	'wxyz'	
BELL	gets	07 hex	*That is, control-*G*, a bell*

Example 5: Record assignment and record-field assignment.

```
STRUCTURE /PRODUCT/
     INTEGER*4 ID
     CHARACTER*16 NAME
     CHARACTER*8 MODEL
     REAL*4 COST
     REAL*4 PRICE
END STRUCTURE
RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
…
CURRENT = NEXT      ! record to record
LINE(1) = CURRENT   ! record to array element
WRITE ( 9 ) CURRENT ! write whole record
NEXT.ID = 82        ! assign a value to a field
```

4

4.4 AUTOMATIC

The AUTOMATIC ♦ statement makes each recursive invocation of the subprogram have its own copy of the specified items. It also makes the specified items become undefined outside the subprogram when the subprogram exits through a RETURN statement.

Syntax

AUTOMATIC *vlist*	
vlist	List of variables and arrays

Description

For automatic variables, there is one copy for each invocation of the procedure. To avoid local variables becoming undefined between invocations, f77 classifies every variable as either static or automatic with all *local* variables being static by default. For other than the default, you can declare variables as static or automatic in a STATIC, AUTOMATIC, or IMPLICIT statement. Compare with -stackvar option in the FORTRAN User's Guide.

One usage of AUTOMATIC is to declare all automatic at the start of a function.

Example: Recursive function with implicit automatic.

```
      INTEGER FUNCTION NFCTRL( I )
      IMPLICIT AUTOMATIC (A-Z)
      ...
      RETURN
      END
```

Save/Static—Local variables and arrays are static by default, so in general, this eliminates the need for SAVE. You can still use SAVE to insure portability. Also, SAVE is safer if you leave a subprogram by some way other than a RETURN.

Restrictions

- Arguments and function values are automatic.
- Automatic variables and arrays cannot appear in DATA, EQUIVALENCE, NAMELIST, RECORD, or SAVE statements.

Examples

Example: Some other uses of AUTOMATIC.

```
      AUTOMATIC A, B, C
      REAL P, D, Q
      AUTOMATIC P, D, Q
      IMPLICIT AUTOMATIC (X-Z)
```

Example: Structures are unpredictable if AUTOMATIC.

```
demo$ cat autostru.f
      AUTOMATIC X
      STRUCTURE /ABC/
        INTEGER I
      END STRUCTURE
      RECORD /ABC/ X       ! X is automatic. It cannot be a structure
      X.I = 1
      PRINT '(I2)', X.I
      END
demo$ f77 -silent autostru.f
demo$ a.out
*** TERMINATING a.out
*** Received signal 10 (SIGBUS)
Bus Error (core dumped)
demo$
```

Note – Sometimes an automatic structure works; sometimes it core dumps.

Remark

An AUTOMATIC statement and a type statement cannot be combined to make an AUTOMATIC *type* statement. For example, the statement

```
      AUTOMATIC REAL X
```

does *not* declare the variable X to be both AUTOMATIC and REAL; it declares the variable REALX to be AUTOMATIC.

4.5 BACKSPACE

The BACKSPACE statement positions the specified file to just before the preceding record.

Syntax

BACKSPACE *u*	
BACKSPACE([UNIT=] *u* [, IOSTAT= *ios*] [, ERR= *s*])	
u	Unit identifier of the external unit connected to the file
ios	I/O status specifier, integer variable, or an integer array element
s	Error specifier: *s* must be the label of an executable statement in the same program unit in which the BACKSPACE statement occurs. Program control is transferred to the label in case of an error during the execution of the BACKSPACE statement.

Description

BACKSPACE in a terminal file has no effect.

u must be connected for *sequential* access. Execution of a BACKSPACE statement on a direct-access file is not defined in the FORTRAN Standard, and is unpredictable. We do not recommend using a BACKSPACE statement on a *direct-access* file, nor do we recommend using a BACKSPACE statement on an *append access* file.

Execution of the BACKSPACE statement modifies the file position as follows.

Prior to Execution	***After Execution***
Beginning of the file	Remains unchanged
Beyond endfile record	Before endfile record
Middle of a record	Start of the same record

Examples

Example 1: Simple backspace.

```
      BACKSPACE 2
      LUNIT = 2
      BACKSPACE LUNIT
```

Example 2: Backspace with error trap.

```
      INTEGER  CODE
      BACKSPACE ( 2, IOSTAT=CODE, ERR=9 )
      …
 9    WRITE (*,*) 'Error during BACKSPACE'
      STOP
```

4.6 BLOCK DATA

The BLOCK DATA statement identifies a subprogram that initializes variables and arrays in labeled common blocks.

Syntax

BLOCK DATA [*name*]	
name	Symbolic *name* of the block data subprogram in which the BLOCK DATA statement appears. This parameter is optional. It is a global name.

Description

A block data subprogram can contain as many labeled common blocks and data initializations as desired.

The BLOCK DATA statement must be the first statement in a block data subprogram.

The only other statements that can appear in a block data subprogram are:

- COMMON
- DATA
- DIMENSION
- END

- EQUIVALENCE
- IMPLICIT
- PARAMETER
- RECORD
- SAVE
- STRUCTURE
- *type* statements

Only an entity defined in a labeled common block can be initially defined in a block data subprogram.

If an entity in a labeled common block is initially defined, all entities having storage units in the common block storage sequence must be specified even if they are not all initially defined.

Restrictions

- Only one unnamed block data subprogram can appear in the executable program.
- The same labeled common block cannot be specified in more than one block data subprogram in the same executable program.
- The optional parameter *name* must not be the same as the name of an external procedure, main program, common block, or other block data subprogram in the same executable program. The name must not be the same as any local name in the subprogram.

Example

```
      BLOCK DATA INIT
      COMMON /RANGE/ X0, X1
      DATA X0, X1 / 2.0, 6.0 /
      END
```

4.7 BYTE

The BYTE ♦ statement specifies the type to be 1-byte integer.
It optionally specifies array dimensions and initializes with values.

Syntax

BYTE *v* [/*c*/] …	
v	Name of a symbolic constant, variable, array, array declarator, function, or dummy function
c	List of constants for the immediately preceding name

Description

This is a synonym for LOGICAL*1. A BYTE type item can hold the logical values .TRUE., .FALSE., one character, or an integer between –128 and 127.

Example

```
      BYTE BIT3 / 8 /, C1 / 'W' /,
 &          COUNTER /0/, M /127/, SWITCH / .FALSE. /
```

4.8 CALL

The CALL statement branches to the specified subroutine, executes the subroutine, and returns to the calling program after finishing the subroutine.

Syntax

CALL *sub* [([*ar* [, *ar*] …])]	
sub	Name of the subroutine to be called
ar	Actual argument to be passed to the subroutine

Description

Arguments are separated by commas.

The FORTRAN Standard requires that actual arguments in a CALL statement must agree in order, number, and type with the corresponding formal arguments of the referenced subroutine; the compiler does not check this.

Recursion is allowed. A subprogram can call itself directly, or indirectly by calling another subprogram that in turns calls this subroutine. Such recursion is nonstandard. ♦

An actual argument, *ar*, must be one of the following:

- An expression
- An intrinsic function permitted to be passed as an argument
- An external function name
- A subroutine name
- An alternate return specifier, "*" or "&" followed by a statement number The "&" is nonstandard. ♦

The simplest expressions, and most frequently used, include such things as

- Variable name
- Array name
- Formal argument (if the CALL statement is inside a subroutine)
- Record name

If a subroutine has no arguments, then a CALL statement referencing that subroutine must not have any actual arguments. A pair of empty matching parentheses can follow the subroutine name.

Execution of the CALL statement proceeds as follows:

1. All expressions (arguments) are evaluated.
2. All actual arguments are associated with the corresponding formal arguments and the body of the subroutine is executed.
3. Normally the control is transferred back to the statement following the CALL statement upon executing a RETURN statement or an END statement in the subroutine. If an alternate return in the form of RETURN *n* is executed, then control is transferred to the statement specified by the *n* alternate return specifier in the CALL statement.

Examples

Example 1: Character string.

```
      CHARACTER *25 TEXT
      TEXT = 'Some kind of major screwup'
      CALL OOPS ( TEXT )
      SUBROUTINE OOPS ( S )
      CHARACTER S*(*)
      WRITE (*,*) S
      END
```

Example 2: Alternate return.

```
      CALL RANK ( N, *8, *9 )
      WRITE (*,*) 'OK - Normal Return'
      STOP
8     WRITE (*,*) 'Minor - 1st alternate return'
      STOP
9     WRITE (*,*) 'Major - 2nd alternate return'
      STOP
      END

      SUBROUTINE RANK ( N, *, * )
      IF ( N .EQ. 0 ) RETURN
      IF ( N .EQ. 1 ) RETURN 1
      RETURN 2
      END
```

Example 3: Another form of alternate return. The "&" is nonstandard. ♦

```
      CALL RANK ( N, &8, &9 )
```

Example 4: Array, array element, and variable. In this example, the real array M matches the real array A, and the real array element Q(1,2) matches the real variable D.

```
      REAL M(100,100), Q(2,2), Y
      CALL SBRX ( M, Q(1,2), Y )
      …
      END
      SUBROUTINE SBRX ( A, D, E )
      REAL A(100,100), D, E
      …
      RETURN
      END
```

Example 5: A structured record and field. The record is nonstandard. ♦

```
      STRUCTURE /PRODUCT/
           INTEGER*4 ID
           CHARACTER*16 NAME
           CHARACTER*8 MODEL
           REAL*4 COST
           REAL*4 PRICE
      END STRUCTURE
      RECORD /PRODUCT/ CURRENT, PRIOR
      CALL SBRX ( CURRENT, PRIOR.ID )
      …
      END
      SUBROUTINE SBRX ( NEW, K )
      STRUCTURE /PRODUCT/
           INTEGER*4 ID
           CHARACTER*16 NAME
           CHARACTER*8 MODEL
           REAL*4 COST
           REAL*4 PRICE
      END STRUCTURE
      RECORD /PRODUCT/ NEW
      …
      RETURN
      END
```

In the above example, the record NEW matches the record CURRENT, and the integer variable K matches the record field PRIOR.OLD.

4.9 CHARACTER

The CHARACTER statement specifies the type of a symbolic constant, variable, array, function, or dummy function to be character.

Optionally, it initializes any of the items with values and specifies array dimensions.

Syntax

CHARACTER [* *len* [,]] *v* [* *len* /*c*/]] …	
v	Name of a symbolic constant, variable, array, array declarator, or function
len	Length in characters of the symbolic constant, variable, array element, or function
c	List of constants for the immediately preceding name

Description

Each character occupies 8 bits of storage, aligned on a character boundary. Character arrays and common blocks containing character variables are packed in an array of character variables. The first character of one element follows the last character of the preceding element, without holes.

The length, *len* must be greater than 0.If *len* is omitted, it is assumed equal to 1.

For local and common character variables, symbolic constants, dummy arguments, or function names, *len* can be an integer constant, or a parenthesized integer constant expression.

For dummy arguments or function names, *len* can have another form: a parenthesized asterisk, that is CHARACTER*(*), which denotes that the function name, or dummy argument has the length of the actual function or argument.

For symbolic constants, *len* can also be a parenthesized asterisk, which indicates that the name is defined as having the length of the constant. This is shown below in Example 5.

The list *c* of constants can be used only for a variable, array, or array declarator. There can be only one constant for the immediately preceding variable, and one constant for each element of the immediately preceding array.

Examples

Example 1: Character strings and arrays of character strings.

```
      CHARACTER*17 A, B(3,4), V(9)
      CHARACTER*(6+3) C
```

The above code is exactly equivalent to the following:

```
      CHARACTER A*17, B(3,4)*17, V(9)*17
      CHARACTER C*(6+3)
```

Both of the above are equivalent to nonstandard variation: ♦

```
      CHARACTER A*17, B*17(3,4), V*17(9)! nonstandard
```

Example 2: No null character-string variables.

There are no null (zero-length) character-string variables. You might expect that a one-byte character string assigned a null constant would have length zero.

```
      CHARACTER S*1
      S = ''
```

During execution of the assignment statement, the variable S is precleared to blank, and then zero characters are moved into S, so S contains one blank; and because of the declaration, LEN( ) will return a length of 1. You cannot declare a size of less than 1, so this is the smallest length string variable you can get.

Example 3: Dummy argument character string with constant length.

```
      SUBROUTINE SCHLEP ( A )
      CHARACTER A*32
```

Example 4: Dummy argument character string with length the same as corresponding actual argument.

```
      SUBROUTINE SCHLEP ( A )
      CHARACTER A*(*)
      …
```

Example 5: Symbolic constant with parenthesized asterisk.

```
      CHARACTER *(*) INODE
      PARAMETER ( INODE = 'Warning: INODE clobbered!' )
```

Example 6: The LEN function. The intrinsic function LEN that returns the actual declared length of a character string. This is mainly for use with CHAR*(*) dummy arguments.

```
      CHARACTER A*17
      A = "xyz"
      PRINT *, LEN( A )
      END
```

The above program will display 17, not 3.

4.10 CLOSE

The CLOSE statement disconnects a file from a unit.

Syntax

CLOSE([UNIT=] *u* [, STATUS= *sta*] [, IOSTAT= *ios*] [, ERR= *s*])	
u	Unit identifier for an external unit. If "UNIT=" is not used, then *u* must be first.
sta	Determines the disposition of the file - *sta* is a character expression whose value, when trailing blanks are removed, can be KEEP or DELETE. The default value for the status specifier is KEEP. For temporary (scratch) files, *sta* is forced to DELETE always. For other files besides scratch files, default *sta* is KEEP.
ios	I/O status specifier - *ios* must be an integer variable or an integer array element.
s	Error specifier - *s* must be the label of an executable statement in the same program containing the CLOSE statement. The program control is transferred to this statement in case an error occurs while executing the CLOSE statement.

Description

For tape it is more reliable to use the `TOPEN()` routines.

The options can be specified in any order.

The `DISP=` and `DISPOSE=` options are allowable alternates for `STATUS=`, with a warning, if the `-ansi` flag is set.

Execution of `CLOSE` proceeds as follows.

1. The specified unit is disconnected.
2. If *sta* is `DELETE`, the file connected to the specified unit is deleted.
3. If an `IOSTAT` argument is specified, *ios* is set to zero if no error was encountered; otherwise it is set to a positive value.

Comments

All open files are closed with default *sta* at normal program termination. Regardless of the specified *sta*, scratch files, when closed, are always deleted.

Execution of a `CLOSE` statement specifying a unit that does not exist or a unit that has no file connected to it, has no effect.

Execution of a `CLOSE` statement specifying a unit zero (standard error) is not allowed, but you can reopen it to some other file.

The unit/file disconnected by the execution of a `CLOSE` statement can be connected again to the same file/unit or to a different file/unit.

Examples

Example 1: Close and keep.

```
      CLOSE ( 2, STATUS='KEEP')
```

Example 2: Close and delete.

```
      CLOSE ( 2, STATUS='DELETE', IOSTAT=I )
```

Example 3: Close and delete a scratch file even though status is KEEP.

```
      OPEN ( 2, STATUS='SCRATCH')
      …
      CLOSE ( 2, STATUS='KEEP', IOSTAT=I )
```

4.11 COMMON

The COMMON statement defines a block of main memory storage so that different program units can share the same data without using arguments.

Syntax

COMMON [/[*cb*]/] *nlist* [[,]/[*cb*] / *nlist*] …	
cb	Common block name
nlist	List of variable names, array names, and array declarators

Description

Common Block Name

If the common block name is omitted, then blank common block is assumed.

Any common block name including blank common can appear more than once in COMMON statements in the same program unit. The list *nlist* following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

Size of a Common Block

The size of a common block is the sum of the sizes of all the entities in the common block, plus space for alignment.

Within a program, all common blocks in different program units that have the same name must be of the same size.

Restrictions

- Formal argument names and function names cannot appear in a COMMON statement.
- An EQUIVALENCE statement must not cause the storage sequences of two different common blocks in the same program unit to be associated. This is shown in example 2.
- An EQUIVALENCE statement must not cause a common block to be extended on the left-hand side. This is shown in Example 4.

Examples

Example 1: Unlabeled common and labeled common.

```
      DIMENSION V(100)
      COMMON V, M
      COMMON / LIMITS / I, J
      …
```

Above, V and M are in the unlabeled common block, while I and J are defined in the named common block, LIMITS.

Example 2: You cannot associate storage of two different common blocks in the same program unit.

```
      COMMON /X/ A
      COMMON /Y/ B
      EQUIVALENCE ( A, B)   ! ← not allowed
```

Example 3: An EQUIVALENCE statement can extend a common block on the right-hand side.

```
      DIMENSION A(5)
      COMMON /X/ B
      EQUIVALENCE ( B, A)
```

Example 4: An EQUIVALENCE statement must not cause a common block to be extended on the left-hand side.

```
      COMMON /X/ A
      REAL B(2)
      EQUIVALENCE ( A, B(2))   ! ← not allowed
```

4.12 COMPLEX

The COMPLEX statement specifies the type of a symbolic constant, variable, array, function, or dummy function to be complex, and optionally specifies array dimensions and size, and initializes with values.

Syntax

COMPLEX [*len[,]] *v* [* *len* [/*c*/]] [, *v* [* *len* [/*c*/]] …	
v	Name of a symbolic constant, variable, array, array declarator, or function
len	Either 8, 16, or 32, the length in bytes of the symbolic constant, variable, array element, or function (32 is *SPARC only*)
c	List of constants for the immediately preceding name

Description

COMPLEX

For a declaration such as COMPLEX W, the variable W is usually two REAL*4 elements contiguous in memory, interpreted as a complex number. Details are in the ”Default Size” section, below.)

COMPLEX*8 ♦

For a declaration such as COMPLEX*8 W, the variable W is always two REAL*4 elements contiguous in memory, interpreted as a complex number.

COMPLEX*16 ♦

For a declaration such as COMPLEX*16 W, W is always two REAL*8 elements contiguous in memory, interpreted as a double-width complex number.

COMPLEX*32 ♦

(SPARC only) For a declaration such as COMPLEX*32 W, the variable W is always two REAL*16 elements contiguous in memory, interpreted as a quadruple-width complex number.

Default Size

If you specify the size as 8, 16, or 32, `COMPLEX*8`, `COMPLEX*16`, `COMPLEX*32`, you get what you specify; if you do *not* specify the size, you get the default size. *(The *32 is for SPARC only.)*

The default size, for a declaration such as `COMPLEX Z`, depends on `-r8`.

- If the `-r8` option is on the `f77` command line, then the compiler allocates 16 bytes, and does 16-byte arithmetic.

  If `-r8` is *not* on the command line, the compiler allocates 8 bytes.

  Similarly, the default size, for such a declaration as `DOUBLE COMPLEX Z`, depends on the `-r8` option.

- If `-r8` is on the `f77` command line, then the compiler allocates 32 bytes, and does 32-byte arithmetic (*SPARC only*).

  If `-r8` is *not* on the command line, the compiler allocates 16 bytes.

- If you put both `-i2` and `-r8` on the `f77` command line, the results are unpredictable.

Comments

Specifying the size is nonstandard. ♦

Double-Complex Functions — There is a double-complex version of each complex built-in function. Generally the specific function names begin with `Z` or `CD` instead of `C`, except for the two functions `DIMAG` and `DREAL`, which return a real value.

Quad-Precision Complex Functions (SPARC only) — There are specific complex functions for quad precision. In general, where there is a specific `REAL` and a corresponding `COMPLEX` with a `C` prefix, and a corresponding `COMPLEX DOUBLE` with a `CD` prefix, there is also a quad-precision `COMPLEX` function with a `CQ` prefix. Examples: `SIN()`, `CSIN()`, `CDSIN()`, `CQSIN()`.

Examples

Example 1: Complex scalars. Styles.

Each of these statements is equivalent to the others. (Don't use all three statements in the same program unit — you cannot declare anything more than once in the same program unit.)

```
COMPLEX U, V
COMPLEX*8 U, V
COMPLEX U*8, V*8
```

Example 2: Initialize complex scalars.

```
COMPLEX U / (1, 9.0) /, V / (4.0, 5 ) /
```

Note that a complex constant is a pair of numbers, either integers or reals.

Example 3: Double complex, some initialization.

```
COMPLEX R*16, V*16
COMPLEX U*16 / (1.0D0, 9 ) /, V*16 / (4.0, 5.0D0) /
COMPLEX*16 X / (1.0D0, 9.0) /, Y / (4.0D0, 5 ) /
```

Note that a double-complex constant is a pair of numbers, and at least one number of the pair must be double precision.

Example 4: Quadruple complex, some initialization *(SPARC only)*.

```
COMPLEX R*32, V*32
COMPLEX U*32 / (1.0Q0, 9 ) /, V*32 / (4.0, 5.0Q0) /
COMPLEX*32 X / (1.0Q0, 9.0) /, Y / (4.0Q0, 5 ) /
```

Note that a quadruple complex constant is a pair of numbers, and at least one number of the pair must be quadruple precision.

Example 5: Complex arrays. All of these are nonstandard.

```
COMPLEX R*16(5), S(5)*16 ! (SPARC only)
COMPLEX U*32(5), V(5)*32 ! (SPARC only)
COMPLEX X*8(5), Y(5)*8
```

4.13 CONTINUE

The CONTINUE statement is a do nothing statement.

Syntax

[*label*] CONTINUE	
label	External statement number

Description

The CONTINUE statement is often used as a place to hang a statement label, usually it is the end of a DO loop.

Label Definition

The CONTINUE statement is used primarily as a convenient point for placing a statement label, particularly as the terminal statement in a DO loop.Execution of a CONTINUE statement has no effect.

If the CONTINUE statement is used as the terminal statement of a DO loop, the next statement executed depends on the DO loop exit condition.

Example

```
      DIMENSION U(100)
      S = 0.0
      DO 1 J = 1, 100
           S = S + U(J)
           IF ( S .GE. 1000000 ) GO TO 2
1     CONTINUE
      STOP
2     CONTINUE
      …
```

4.14 DATA

The DATA statement initializes variables, substrings, arrays, array elements, records, and record fields.

Syntax

DATA *nlist* / *clist* / [[,] *nlist* / *clist* /] …	
nlist	List of variables, arrays, array elements, substrings, and implied DO lists separated by commas
clist	List of the form: *c* [, *c*] …
c	One of the forms: *c* or *r*c*, and *c* is a constant or the symbolic name of a constant.
r	Nonzero, unsigned integer constant or the symbolic name of such constant

Description

All initially defined items are defined with the specified values when an executable program begins running.

The *r*c* is equivalent to *r* successive occurrences of the constant *c*.

A DATA statement is a nonexecutable statement and must appear after all specification statements but can be interspersed with statement functions and executable statements.

The number of constants (taking into account the repeat factor) in *clist* must be equal to the number of items in the *nlist*. The appearance of an array in *nlist* is equivalent to specifying a list of all elements in that array. Array elements can be indexed by constant subscripts only.

Normal type conversion takes place for each noncharacter member of the clist.

Character Constants in the DATA Statement

If the length of a character item in *nlist* is greater than the length of the corresponding constant in *clist*, it is padded with blank characters on the right.

If the length of a character item in *nlist* is less than that of the corresponding constant in *clist*, the additional rightmost characters are ignored.

If the constant in *clist* is of integer type and the item of *nlist* is of character type, they must conform to the following rules.

- The character item must have a length of one character.
- The constant must be of type integer and have a value in the range 0 through 255. For ^A, ^B, ^C, do not hold down the control key and press A, B, or C; use the `char()` function.

If the constant of *clist* is a character constant or a Hollerith constant, and the item of *nlist* is of type `INTEGER`, then the number of characters that can be assigned is 2 or 4 for `INTEGER*2` and `INTEGER*4` respectively. If the character constant or the Hollerith constant has fewer characters than the capacity of the item, the constant is extended on the right with spaces. If the character or the Hollerith constant contains more characters than can be stored, the constant is truncated on the right.

Implied DO Lists

An *nlist* can specify an implied `DO` list for initialization of array elements.

The form of implied `DO` list is:

`(dlist, iv=m1, m2 [,m3 ])`	
`dlist`	List of array element names and implied `DO` lists
`iv`	Integer variable, called the implied `DO` variable
`m1`	Integer constant expression specifying the initial value of *iv*
`m2`	Integer constant expression specifying the limit value of *iv*
`m3`	Integer constant expression specifying the increment value of *iv*. If *m3* is omitted, then a default value of 1 is assumed.

The range of an implied `DO` loop is *dlist*. The iteration count for the implied `DO` is computed from *m1*, *m2*, and *m3*, and it must be positive.

Comments

Variables can also be initialized in type statements. This is an extension of the FORTRAN Standard. Examples are given under each of the individual type statements and under the general *type* statement. ♦

Examples

Example 1: Character, integer, and real scalars. Real arrays.

```
      CHARACTER TTL*16
      REAL VEC(5), PAIR(2)
      DATA TTL / 'Arbitrary Titles' /,
&           M / 9 /, N / 0 /,
&           PAIR(1) / 9.0 /,
&           VEC / 3*9.0, 0.1, 0.9 /
      …
```

Example 2: Arrays — implied DO.

```
      REAL R(3,2), S(4,4)
      DATA ( S(I,I), I=1,4) / 4*1.0 /,
&          (( R(I,J), J=1,3), I=1,2) / 6*1.0 /
      …
```

Example 3: Mixing integer and character.

```
      CHARACTER CR*1
      INTEGER I*2, N*4
      DATA I / 'oy' /, N / 4Hs12t /, CR / 13 /
      …
```

4.15 DECODE/ENCODE

`ENCODE` writes to a character variable, array, or array element. ♦
`DECODE` reads from a character variable, array, or array element. ♦
Data is edited according to the format identifier.

Similar functionality can be accomplished using internal files with formatted sequential `WRITE` statements and `READ` statements. `ENCODE` and `DECODE` are not in the FORTRAN Standard, and are provided for compatibility with older versions of FORTRAN.

Syntax

ENCODE(*size, f, buf* [, IOSTAT= *ios*] [, ERR= *s*]) [*iolist*]	
DECODE(*size, f, buf* [, IOSTAT= *ios*] [, ERR= *s*]) [*iolist*]	
size	Number of characters to be translated, an integer expression
f	Format identifier, either the label of a FORMAT statement, or a character expression specifying the format string, or an asterisk. If *f* specifies more than one record, it is an error.
buf	Variable, array, or array element
ios	I/O status specifier, *ios* must be an integer variable or an integer array element.
s	Error specifier (statement label) *s* must be the label of executable statement in the same program unit in which the ENCODE and DECODE statement occurs.
iolist	List of input/output items.

Description

The entities in the I/O list must be one of the following:

- Variables
- Substrings
- Arrays
- Array elements
- Records
- Record fields

A simple unsubscripted array name specifies all of the elements of the array in memory storage order, with the leftmost subscript increasing more rapidly.

Execution proceeds as follows.

1. The ENCODE statement translates the list items to character form according to the format identifier, and stores the characters in *buf*. A WRITE operation on internal files does the same.

2. The DECODE statement translates the character data in *buf* to internal (binary) form according to the format identifier, and stores the items in the list. A READ statement does the same.
3. If *buf* is an array, its elements are processed in the order of subscript progression, with the leftmost subscript increasing more rapidly.
4. The number of characters that an ENCODE or a DECODE statement can process depends on the data type of *buf*. For example, an INTEGER*2 array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array. A character variable or character array element can contain characters equal in number to its length. A character array can contain characters equal in number to the length of each element multiplied by the number of elements.
5. The interaction between the format identifier and the I/O list is the same as for a formatted I/O statement.

Example

Program using DECODE/ENCODE.

```
      CHARACTER S*6 / '987654' /, T*6
      INTEGER V(3)*4
      DECODE( 6, '(3I2)', S ) V
      WRITE( *, '(3I3)') V
      ENCODE( 6, '(3I2)', T ) V(3), V(2), V(1)
      PRINT *, T
      END
```

The above program has this output.

```
   98 76 54
  547698
```

The DECODE reads the characters of S as 3 integers, and stores them into V(1), V(2), and V(3).

The ENCODE statement writes the values V(3), V(2), and V(1) into T as characters; T then contains '547698'.

4.16 DIMENSION

The DIMENSION statement specifies the number of dimensions for an array, including the number of elements in each dimension.

Optionally, it initializes items with values.

Syntax

DIMENSION *a* (*d*) [,*a* (*d*)] …	
a	Name of an array
d	Specifies the dimensions of the array. It is a list of 1 to 7 declarators separated by commas.

Description

Dimension Declarator

The lower and upper limits of each dimension are designated by a dimension declarator. The form of a dimension declarator is:

```
[ dd1 :] dd2
```

dd1 and *dd2* are dimension bound expressions specifying the lower- and upper-bound values. They can be arithmetic expressions of type integer or real. They can be formed using constants, symbolic constants, formal arguments, or variables defined in the COMMON statement. Array references and references to user-defined functions cannot be used in the dimension bound expression. *dd2* can also be an asterisk. If *dd1* is not specified, a value of one is assumed. The value of *dd1* must be less than or equal to *dd2*.

Nonconstant dimension bound expressions can be used in a subprogram to define adjustable arrays, but not in a main program.

Noninteger dimension bound expressions are converted to integers before use. Any fractional part is truncated.

Adjustable Array

If the dimension declarator is an arithmetic expression that contains formal arguments or variables defined in the COMMON statement, then the array is called an adjustable array. In such cases the dimension is equal to the initial value of the argument upon entry into the subprogram.

Assumed-Size Array

The array is called an assumed-size array when the dimension declarator contains an asterisk. In such cases the upper bound of that dimension is not stipulated. An asterisk can only appear for formal arrays and as the upper bound of the last dimension in an array declarator.

Examples

Example 1: Arrays in a main program.

```
      DIMENSION M(4,4), V(1000)
      …
      END
```

In the above example, M is specified as an array of dimensions 4 ×4 and V is specified as an array of dimension 1000.

Example 2**:** Adjustable array in a subroutine.

```
      SUBROUTINE INV( M, N )
      DIMENSION M( N, N )
      …
      END
```

In the above example, the formal arguments are an array, M, and a variable N. M is specified to be a square array of dimensions N× N.

Example 3: Lower and upper bounds.

```
      DIMENSION HELIO (-3:3, 4, 3:9)
      …
      END
```

Above, HELIO is a 3-dimensional array. The first element is HELIO(-3,1,3) and the last element is HELIO(3,4,9).

Example 4: Dummy array with lower and upper bounds.

```
      SUBROUTINE ENHANCE( A, NLO, NHI )
      DIMENSION A(NLO : NHI)
      …
      END
```

Example 5: Noninteger bounds.

```
      PARAMETER ( LO = 1, HI = 9.3 )
      DIMENSION A(HI, HI*3 + LO )
      …
      END
```

In the above example, A is an array of dimension 9×28.
Example 6: Adjustable array with noninteger bounds.

```
      SUBROUTINE ENHANCE( A, X, Y )
      DIMENSION A(X : Y)
      …
      END
```

4.17 DO

The DO statement repeatedly executes a set of statements.

Syntax

```
DO s [,] loop-control
  or
DO  loop-control ♦
```

s is a statement number.
The form of *loop-control* is:

variable = *e1, e2* [, *e3*]	
variable	Variable of type integer, real, or double precision.)
e1, e2, e3	Expressions of type integer, real or double precision, specifying initial, limit, and increment values respectively.

Description

Labeled DO *Loop*

A labeled DO loop consists of the following:

- DO statement
- Set of executable statements called a block
- Terminal statement, usually a CONTINUE statement

Terminal Statement

The statement identified by *s* is called the *terminal statement*. It must follow the DO statement in the sequence of statements within the same program unit as the DO statement.

The terminal statement should *not* be one of the following statements:

- Unconditional GO TO
- Assigned GO TO
- Arithmetic
- IF
- Block IF
- ELSE IF
- ELSE
- END IF
- RETURN
- STOP
- END DO

If the terminal statement is a logical IF statement, it can contain any executable statement *except:*

- DO
- DO WHILE
- Block IF
- ELSE IF
- ELSE
- END IF
- END
- Logical IF statement

DO Loop Range

The range of a `DO` loop consists of all of the executable statements that appear following the `DO` statement, up to and including the terminal statement.

If a `DO` statement appears within the range of another `DO` loop, its range must be entirely contained within the range of the outer `DO` loop. More than one labeled `DO` loop can have the same terminal statement.

If a `DO` statement appears within an `IF`, `ELSE IF`, or `ELSE` block, the range of the associated `DO` loop must be contained entirely within that block.

If a Block `IF` statement appears within the range of a `DO` loop, the corresponding `END IF` statement must also appear within the range of that `DO` loop.

Block DO Loop ♦

A block `DO` loop consists of:

- `DO` statement
- Set of executable statements called a block
- Terminal statement, an `END DO` statement

This is nonstandard.

Execution proceeds as follows.

1. The expressions *e1*, *e2*, and *e3* are evaluated. If *e3* is not present, its value is assumed to be one.
2. The `DO` variable is initialized with the value of *e1*.
3. The iteration count is established as the value of the expression.

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

   Note that the iteration count is zero if either of the following is true.

   e1 > *e2* and *e3* > zero.

   e1 < *e2* and *e3* < zero.

   If the `-onetrip` compile time option is specified, then the iteration count is never less than one.

4. The iteration count is tested, and, if it is greater than zero, the range of the DO loop is executed.

Terminal Statement Processing

After the terminal statement of a DO loop is executed, the following steps are performed.

1. The value of the DO variable, if any, is incremented by the value of *e3* that was computed when the DO statement was executed.
2. The iteration count is decreased by one.
3. The iteration count is tested, and if it is greater than zero, the statements in the range of the DO loop are executed again.

Restrictions

- The DO variable must not be modified in any way within the range of the DO loop.
- You must not jump into the range of a DO loop from outside its range.

Comments

In some cases, the DO variable can overflow as a result of an increment that is performed prior to testing it against the final value. When this happens, your program has an error, and neither the compiler nor the runtime system detects it. In this situation, though the DO variable wraps around, the loop can terminate properly.

Examples

Example 1: Nested DO's.

```
      N = 0
      DO 210 I = 1, 10
           J = I
           DO 200 K = 5, 1
                L = K
                N = N + 1
200        CONTINUE
210   CONTINUE
      WRITE(*,*)'I =',I, ', J =',J, ', K =',K, ', N =',N, ', L =',L
      END
demo % f77 -silent DoNest1.f
"DoNest1.f", line 4: Warning: DO range never executed
demo % a.out
I =  11, J =  10, K =  5, N =  0, L =  0
demo %
```

The inner loop is not executed, and at the `WRITE`, `L` is undefined. Here `L` is shown as 0, but that is implementation dependent; do not rely on it.

Example 2: The program `DoNest2.f` (`DO` variable always defined).

```
      INTEGER COUNT, OUTER
      COUNT = 0
      DO OUTER = 1, 5
           NOUT = OUTER
           DO INNER = 1, 3
                NIN = INNER
                COUNT = COUNT+1
           END DO
      END DO
      WRITE(*,*) OUTER, NOUT, INNER, NIN, COUNT
      END
```

The above program prints out:

```
      6 5 4 3 15
```

4.18 DO WHILE

The DO WHILE ♦ statement repeatedly executes a set of statements while the specified condition is true.

Syntax

DO [*s* [,]] WHILE (*e*)	
s	Label of an executable statement
e	Logical expression

Description

Execution proceeds as follows.

1. The specified expression is evaluated.
2. If the value of the expression is true, the statements in the range of the DO WHILE loop are executed.
3. If the value of the expression is false, control is transferred to the statement following the DO WHILE loop.

Terminal Statement

If *s* is specified, the statement identified by it is called the terminal statement and it must follow the DO WHILE statement. The terminal statement must *not* be one of the following statements:

- Unconditional GO TO
- Assigned GO TO
- Arithmetic IF
- Block IF ELSE IF
- ELSE
- END IF
- RETURN
- STOP
- END
- DO
- DO WHILE

If the terminal statement is a logical IF statement, it can contain any executable statement *except*:

- DO
- DO WHILE
- Block IF
- ELSE IF
- ELSE
- END IF
- END
- Logical IF

If *s* is not specified, the DO WHILE loop must end with an END DO statement.

DO WHILE *Loop Range*

The range of a DO WHILE loop consists of all the executable statements that appear following the DO WHILE statement, up to and including the terminal statement.

If a DO WHILE statement appears within the range of another DO WHILE loop, its range must be entirely contained within the range of the outer DO WHILE loop. More than one DO WHILE loop can have the same terminal statement.

If a DO WHILE statement appears within an IF, ELSE IF, or ELSE block, the range of the associated DO WHILE loop must be entirely within that block.

If a Block IF statement appears within the range of a DO WHILE loop, the corresponding END IF statement must also appear within the range of that DO WHILE loop.

Terminal Statement Processing

After the terminal statement of a DO WHILE loop is executed, control is transferred back to the corresponding DO WHILE statement.

Restriction

If you jump into the range of a DO WHILE loop from outside its range, then the results are unpredictable.

Comments

The variables used in the *e* can be modified in any way within the range of the DO WHILE loop.

Examples

Example 1: A DO WHILE *without* a statement number.

```
      INTEGER A(4,4), C, R
      …
      C = 4
      R = 1
      DO WHILE ( C .GT. R )
           A(C,R) = 1
           C = C - 1
      END DO
```

Example 2: A DO WHILE *with* a statement number.

```
      INTEGER A(4,4), C, R
      …
      DO 10 WHILE ( C .NE. R )
           A(C,R) = A(C,R) + 1
           C = C+1
10    CONTINUE
```

4.19 DOUBLE COMPLEX

The DOUBLE COMPLEX ♦ statement specifies the type to be double complex. It optionally specifies array dimensions and size, and initializes with values.

Syntax

DOUBLE COMPLEX *v* [/*c*/] [, *v* [/*c*/] …	
v	Name of a symbolic constant, variable, array, array declarator, function, or dummy function
c	List of constants for the immediately preceding name

Description

DOUBLE COMPLEX ♦

For a declaration such as DOUBLE COMPLEX Z, the variable Z is usually two REAL*8 elements contiguous in memory, interpreted as one double-width complex number. There is more detail in the *Default Size* section, below.

COMPLEX*16 ♦

For a declaration such as COMPLEX*16 Z, the variable Z is always two REAL*8 elements contiguous in memory, interpreted as one double-width complex number.

Default Size

If you explicitly specify the size as 16, COMPLEX*16, you get what you specify; if you *do not* specify the size, you get the default size. Default size, for such a declaration as DOUBLE COMPLEX Z, depends on –r8.

- If –r8 is on the f77 command line, then the compiler allocates 32 bytes, and does 32-byte arithmetic (*SPARC only*).
  If –r8 is *not* on the command line, then the compiler allocates 16 bytes, and does 16-byte arithmetic.
- If you put both –i2 and –r8 on the f77 command line, the results are unpredictable.

Comments

Double-Complex Functions

There is a double-complex version of each complex built-in function. Generally the specific function names begin with Z or CD instead of C, except for the two functions DIMAG and DREAL, which return a real value. Examples: SIN(), CSIN(), CDSIN().

Example: Double-complex scalars and arrays.

```
      DOUBLE COMPLEX U, V
      DOUBLE COMPLEX W(3,6)
      COMPLEX*16 X, Y(5,5)
      COMPLEX  U*16(5), V(5)*16
```

4.20 DOUBLE PRECISION

The DOUBLE PRECISION statement specifies the type to be double precision, and optionally specifies array dimensions and initializes with values.

Syntax

DOUBLE PRECISION *v* [/*c*/] [, *v* [/*c*/] …	
v	Name of a symbolic constant, variable, array, array declarator, function, or dummy function
c	List of constants for the immediately preceding name

Description

DOUBLE PRECISION

For a declaration such as DOUBLE PRECISION X, the variable X is usually a REAL*8 element in memory, interpreted as one double-width real number. (There is more detail under *Default Size*, below.)

REAL*8 ♦

For a declaration such as `REAL*8 X`, the variable `X` is always an element of type `REAL*8` in memory, interpreted as a double-width real number.

Default Size

If you explicitly specify the size as 8, `REAL*8`, you get what you specify; if you *do not* specify the size, you get the default size.

The default size, for such a declaration as "`DOUBLE PRECISION X`" depends on the `-r8` option as follows:

- If `-r8` is on the `f77` command line, then the compiler allocates 16 bytes, and does 16-byte arithmetic (*SPARC only*).
- If `-r8` is *not* on the command line, then the compiler allocates 8 bytes, and does 8-byte arithmetic.
- If you put both `-i2` and `-r8` on the `f77` command line, the results are unpredictable.

Example

Example: Double-precision scalars and arrays.

```
      DOUBLE PRECISION R, S
      DOUBLE PRECISION T(3,6)
      REAL*8 U(3,6)
      REAL V*8(6), W(6)*8
```

4.21 ELSE

The ELSE statement indicates the beginning of an ELSE block.

Syntax

```
IF ( e ) THEN
     ...
ELSE
     ...
END IF
```

e	Logical expression

Description

Execution of an ELSE statement has no effect on the program.

ELSE Block

An ELSE block consists of all the executable statements following the ELSE statements, up to but not including the next END IF statement at the same IF level as the ELSE statement. See Section 4.40, “IF (Block),” for more detail.

An ELSE block can be empty.

Restrictions

- You cannot jump into an ELSE block from outside the ELSE block.
- The statement label, if any, of an ELSE statement cannot be referenced by any statement.
- A matching END IF statement of the same IF level as the ELSE must appear before any ELSE IF or ELSE statement at the same IF level.

Examples

Example 1: ELSE.

```
      CHARACTER S
      …
      IF ( S .GE. ’0’ .AND. S .LE. ’9’ ) THEN
           CALL PUSH
      ELSE
           CALL TOLOWER
      END IF
      …
```

Example 2: An invalid ELSE IF where an END IF is expected.

```
      IF ( K .GT. 5 ) THEN
           N = 1
      ELSE
           N = 0
      ELSE IF ( K .EQ. 5 ) THEN ←incorrect
      …
```

4.22 ELSE IF

The ELSE IF provides a multiple alternative decision structure.

Syntax

IF (*e1*) THEN ELSE IF (*e2*) THEN END IF…	
e1 and *e2*	Logical expressions

Description

You can make a series of independent tests, and each test can have its own sequence of statements.

An ELSE IF block consists of all the executable statements following the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement at the same IF level as the ELSE IF statement.

An ELSE IF block can be empty.

Restrictions

- You cannot jump into an ELSE IF block from outside the ELSE IF block.
- The statement label, if any, of an ELSE IF statement cannot be referenced by any statement.
- A matching END IF statement of the same IF level as the ELSE IF must appear before any ELSE IF or ELSE statement at the same IF level.

Execution of the ELSE IF proceeds as follows:

1. *e* is evaluated.
2. If *e* is true, execution continues with the first statement of the ELSE IF block. If *e* is true and the ELSE IF block is empty, control is transferred to the next END IF statement at the same IF level as the ELSE IF statement.
3. If *e* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement at the same IF level as the ELSE IF statement.

Example

Example: ELSE IF.

```
      READ (*,*) N
      IF ( N .LT. 0 ) THEN
           WRITE(*,*) 'N<0'
      ELSE IF ( N .EQ. 0) THEN
           WRITE(*,*) 'N=0'
      ELSE
           WRITE(*,*) 'N>0'
      END IF
```

4.23 ENCODE/DECODE

The ENCODE ♦ statement writes data from a list to memory.

Syntax

ENCODE(*size, f, buf* [, IOSTAT= *ios*] [, ERR= *s*]) [*iolist*]	
size	Number of characters to be translated
f	Format identifier
buf	Variable, array, or array element
ios	I/O status specifier
s	Error specifier (statement label)
iolist	List of I/O items, each a character variable, array, or array element

Description

ENCODE is provided for compatibility with older versions of FORTRAN. Similar functionality can be accomplished using internal files with a formatted sequential WRITE statement. ENCODE is not in the FORTRAN Standard.

Data are edited according to the format identifier.

Example

```
      CHARACTER S*6, T*6
      INTEGER V(3)*4
      DATA S / '987654' /
      DECODE( 6, 1, S ) V
1     FORMAT( 3 I2 )
      ENCODE( 6, 1, T ) V(3), V(2), V(1)
```

The DECODE reads the characters of S as 3 integers, and stores them into V(1), V(2), and V(3). The ENCODE statement writes the values V(3), V(2), and V(1), into T as characters; T then contains '547698'.

See Section 4.15, "DECODE/ENCODE," for more detail and a full example.

4.24 END

The END statement indicates the end of a program unit.

Syntax

```
END
```

Description

The END statement:

- Must be the last statement in the program unit.
- Must be the only statement in a line.
- Can have a label.

No other statement such as an END IF statement, can have an initial line that appears to be an END statement.

In a main program an END statement terminates the execution of the program.

In a function or subroutine, it has the effect of a RETURN. ♦

In the FORTRAN Standard the END statement cannot be continued, but f77 allows this. ♦

Example

Example: END.

```
      PROGRAM MAIN
      WRITE( *, * ) 'Very little'
      END
```

4.25 END DO

The END DO statement terminates a DO loop. ♦

Syntax

```
END DO
```

Description

The END DO statement is the delimiting statement of a Block DO statement. If the statement label is not specified in a DO statement, the corresponding terminating statement must be an END DO statement. You can branch to an END DO statement only from within the range of the DO loop that it terminates.

Examples

Example 1: A DO with a statement number.

```
      DO 10 N = 1, 100
           …
10    END DO
```

Example 2: A DO without statement number.

```
      DO N = 1, 100
           …
      END DO
```

4.26 END FILE

The END FILE statement writes an end-of-file record as the next record of the file connected to the specified unit.

Syntax

END FILE *u*	
END FILE ([UNIT=] *u* [, IOSTAT= *ios*] [, ERR= *s*])	
u	Unit identifier of an external unit connected to the file, The options can be specified in any order, but if "UNIT=" is omitted, then *u* must be first.
ios	I/O status specifier, an integer variable or an integer array element.
s	Error specifier, *s* must be the label of an executable statement in the same program in which the END FILE statement occurs. The program control is transferred to the label in the event of an error during the execution of the END FILE statement.

Description

Tape

If you are using the ENDFILE statement and other standard FORTRAN I/O for tapes, we recommend that you use the TOPEN() routines instead, because they are more reliable.

Two endfile records signify the end-of-tape mark. When writing to a tape file, ENDFILE writes two endfile records, then the tape backspaces over the second one. If the file is closed at this point, both end-of-file and end-of-tape are marked. If more records are written at this point (either by continued write statements or by another program if you are using no-rewind magnetic tape), the first tape mark stands (endfile record), and is followed by another data file, then by more tape marks, and so on.

Comments

u must be connected for *sequential* access. Execution of an `END FILE` statement on a direct-access file is not defined in the FORTRAN Standard, and is unpredictable. Do not use an `END FILE` statement on a direct-access file.

Examples

Example 1: Constants.

```
      END FILE 2
      END FILE ( 2 )
      END FILE ( UNIT=2 )
```

Example 2: Variables.

```
      LOGUNIT = 2
      END FILE LOGUNIT
      END FILE ( LOGUNIT )
      END FILE ( UNIT=LOGUNIT )
```

Example 3: Error trap.

```
      NOUT = 2
      END FILE ( UNIT=NOUT, IOSTAT=KODE, ERR=9)
      …
 9    WRITE(*,*) 'Error at END FILE, on unit', NOUT
      STOP
```

4.27 END IF

The END IF statement ends the Block IF that the IF began.

Syntax

```
END IF
```

Description

For each Block IF statement there must be a corresponding END IF statement in the same program unit. An END IF statement matches if it is at the same IF level as the Block IF statement.

Examples

Example 1: IF/END IF.

```
      IF ( N .GT. 0 )THEN
           N = N+1
      END IF
```

Example 2: IF/ELSE/END IF.

```
      IF ( N .EQ. 0 ) THEN
           N = N+1
      ELSE
           N = N-1
      END IF
```

4.28 END MAP

The END MAP ♦ statement terminates the MAP declaration.

Syntax

```
END MAP
```

Description

See Section 4.70, "UNION and MAP," for further information.

Restriction

The MAP statement must be within a UNION statement.

Example

```
      …
      MAP
           CHARACTER *16 MAJOR
      END MAP
      …
```

4.29 END STRUCTURE

The END STRUCTURE ♦ statement terminates the STRUCTURE statement.

Syntax

```
END STRUCTURE
```

Description

See Section 4.66, "STRUCTURE," for further information.

Example

```
      STRUCTURE /PROD/
           INTEGER*4     ID
           CHARACTER*16  NAME
           CHARACTER*8   MODEL
           REAL*4        COST
           REAL*         PRICE
      END STRUCTURE
```

4.30 END UNION

The END UNION ♦ statement terminates the UNION statement.

Syntax

```
END UNION
```

Description

For END UNION, see Section 4.70, “UNION and MAP,” for further information.

Example

```
      UNION
           MAP
                CHARACTER*16
           END MAP
           MAP
                INTEGER*2    CREDITS
                CHARACTER *8 GRAD_DATE
           END MAP
      END UNION
```

4

4.31 ENTRY

The ENTRY statement defines an alternate entry point within a subprogram.

Syntax

ENTRY *en* [([*fa* [, *fa*] …])]	
en	Symbolic name of an entry point in a function or subroutine subprogram
fa	Formal argument. It can be a variable name, array name, formal procedure name, or an asterisk specifying an alternate return label.

Description

Referencing Procedures by Entry Names

An ENTRY name used in a subroutine subprogram is treated like a subroutine and can be referenced with a CALL statement. Similarly, the ENTRY name used in a function subprogram is treated like a function and can be referenced as a function reference.

An entry name can be specified in an EXTERNAL statement and used as an actual argument. It cannot be used as a dummy argument.

Execution of an ENTRY subprogram (subroutine or function) begins with the first executable statement after the ENTRY statement.

The ENTRY statement is a nonexecutable statement.

The entry name cannot be used in the executable statements that physically precede the appearance of the entry name in an ENTRY statement.

Parameter Correspondence

The formal arguments of an ENTRY statement need not be the same in order, number, type, and name as those for FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. Each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, type, and name with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

Alternate return arguments in ENTRY statements can be specified by placing asterisks in the dummy argument list. Ampersands are valid alternates. ♦ ENTRY statements that specify alternate return arguments can be used only in subroutine subprograms, not functions.

Restrictions

- An ENTRY statement cannot be used within a block IF construct or a DO loop.
- If an ENTRY statement appears in a character function subprogram, it must be defined as type CHARACTER with the same length as that of a function subprogram.

Examples

Example 1: Multiple entry points in a subroutine.

```
      SUBROUTINE FINAGLE( A, B, C )
      INTEGER A, B
      CHARACTER C*4
      …
      RETURN

      ENTRY SCHLEP( A, B, C )
      …
      RETURN

      ENTRY SHMOOZ
      …
      RETURN
      END
```

In the above example, the subroutine FINAGLE has two alternate entries; the entry SCHLEP has an argument list; the entry SHMOOZ has no argument list.

Example 2: Calling entry points in a subroutine. In the calling routine you can call the above subroutine and entries as follows.

```
      INTEGER A, B
      CHARACTER C*4
      …
      CALL FINAGLE( A, B, C )
      …
      CALL SHMOOZ
      …
      CALL SCHLEP( A, B, C )
      …
```

Above, the order of the call statements need not match the order of the entry statements.

Example 3: Multiple entry points in a function.

```
      REAL FUNCTION F2 ( X )
      F2 = 2.0 * X
      RETURN

      ENTRY F3 ( X )
      F3 = 3.0 * X
      RETURN

      ENTRY FHALF ( X )
      FHALF = X / 2.0
      RETURN
      END
```

4.32 EQUIVALENCE

The EQUIVALENCE statement specifies that two or more variables or arrays in a program unit share the same memory.

Syntax

EQUIVALENCE (*nlist*) [, (*nlist*)] …	
nlist	List of variable names, array element names, array names, and character substring names separated by commas

Description

Equivalence Association

An EQUIVALENCE statement stipulates that the storage sequence of the entities whose names appear in the list *nlist* must have the same first memory location.

An EQUIVALENCE statement can cause association of entities other than specified in the *nlist*.

Array Names and Array Element Names

An array name, if present, refers to the first element of the array.

If an array element name appears in an EQUIVALENCE statement, the number of subscripts can be different from the number of dimensions specified in the array declarator for the array name.

Restrictions

- In *nlist*, names of substrings, dummy arguments and functions are not permitted.
- Subscripts of array elements must be integer constants greater than the lower bound and less than or equal to the upper bound.
- An EQUIVALENCE statement can associate an element of type character with a noncharacter element. ♦

- An EQUIVALENCE statement cannot specify that the same storage unit is to occur more than once in a storage sequence. For example the following is not allowed.

```
DIMENSION A (2)
EQUIVALENCE (A(1),B), (A(2),B)
```

- An EQUIVALENCE statement cannot specify that consecutive storage units are to be nonconsecutive. For example, the following is not allowed.

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A(1), D(1)), (A(2), D(2))
```

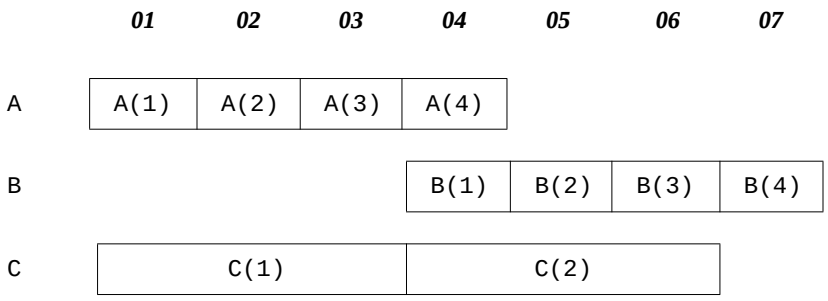
Comments

When COMMON statements and EQUIVALENCE statements are used together, several additional rules can apply. For such rules, refer to the notes on the COMMON statement.

Example

```
CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A,C(1)),(B,C(2))
```

The association of A, B, and C can be graphically illustrated as follows.



4.33 EXTERNAL

The EXTERNAL statement specifies procedures or dummy procedures as external, and allows their symbolic names to be used as actual arguments.

Syntax

EXTERNAL *proc* [, *proc*] …	
proc	Name of external procedure, dummy procedure, or block data routine.

Description

Subroutine or Function Name

If an external procedure or a dummy procedure is an actual argument, it must be in an EXTERNAL statement in the same program unit.

If an intrinsic function name appears in an EXTERNAL statement, that name refers to some external subroutine or function. The corresponding intrinsic function is not available in the program unit.

Restrictions

- A subroutine or function name can appear in only one of the EXTERNAL statements of a program unit.
- A statement function name must not appear in an EXTERNAL statement.

Examples

Example 1: Use your own version of TAN.

```
      EXTERNAL TAN
      T = TAN( 45.0 )
      …
      END
      FUNCTION TAN( X )
      …
      RETURN
      END
```

Example 2: Pass a user-defined function name as an argument.

```
      REAL AREA, LOW, HIGH
      EXTERNAL FCN
      …
      CALL RUNGE ( FCN, LOW, HIGH, AREA )
      …
      END

      FUNCTION FCN( X )
      …
      RETURN
      END

      SUBROUTINE RUNGE ( F, X0, X1, A )
      …
      RETURN
      END
```

4.34 FORMAT

The FORMAT statement specifies the layout of the input or output records.

Syntax

s FORMAT (*f*)	
s	Statement label
f	Format specification list

The items in *f* have the form:

[*r*] *d*	
[*r*] (*f*)	
r	A repeat factor
d	An edit descriptor (repeatable or nonrepeatable), but if *r* is present, then *d* must be repeatable.

Repeatable Edit Descriptors

I I*w* I*w.m* O O*w* O*w.m* Z Z*w* Z*w.m*	F F*w* F*w.m* A A*w* L L*w*	E E*w* E*w.m* E*w.m.e* E*w.m*E*e*	D D*w* D*w.m* D*w.m.e* D*w.m*E*e*	G G*w* G*w.m* G*w.m.e* G*w.m*E*e*

Summary

- I, O, Z are for integers (decimal, octal, hex)
- F, E, D, G are for reals (fixed-point, exponential, double, general)
- A is for characters
- L is for logicals

See Section 5.4, “Formatted I/O,” for full details of these edit descriptors.

Nonrepeatable Edit Descriptors

'a1a2 … an'	[*k*]R	*k* defaults to 10
"*a1a2 … an*"	[*k*]P	*k* defaults to 0
*n*H*a1a2 … an*	S	
$	SU	
/	SP	
:	SS	
B	T*n*	
BN	*n*T	
BZ	TL[*n*]	*n* defaults to 1
	TR[*n*]	*n* defaults to 1
	[*n*]X	*n* defaults to 1

Variable Format Expressions ♦

In general, any integer constant in a format can be replaced by an arbitrary expression enclosed in angle brackets.

```
1    FORMAT(  … < e > … )
```

Restriction

The “*n*” in an “*n*H...” edit descriptor cannot be a variable format expression.

Description

The FORMAT statement includes the explicit editing directives to produce or use the layout of the record. It is used with formatted input/output statements and ENCODE/DECODE statements.

Repeat Factor

r must be a nonzero, unsigned, integer constant.

Repeatable Edit Descriptors

The descriptors I, O, Z, F, E, D, G, L, and A indicate the manner of editing and are repeatable.

w and *e* are nonzero, unsigned integer constants.

d and *m* are unsigned integer constants.

Nonrepeatable Edit Descriptors

The descriptors are the following:

("), ($), (′), (/), (:), B, BN, BZ, H, P, R, Q, S, SU, SP, SS, T, TL, TR, X

These descriptors indicate the manner of editing and are *not* repeatable.

- Each *ai* is any ASCII character.
- *n* is a nonzero, unsigned integer constant.
- *k* is an optionally signed integer constant.

Item Separator

Items in the format specification list are separated by commas. A comma can be omitted before or after the slash and colon edit descriptors, and between a P edit descriptor, and the immediately following F, E, D, or G edit descriptors.

In some sense, the comma can be omitted anywhere the meaning is clear without it, but, other than those cases listed above, this is nonstandard. ◆

Restriction

The FORMAT statement label cannot be used in a GO TO or alternate return.

Warnings

- For *constant* formats, invalid format strings cause warnings and/or error messages at compile time.
- For formats in variables, invalid format strings cause warnings and/or error messages at runtime.
- For variable format expressions, of the form <e>, invalid format strings cause warnings and/or error messages at compile time or runtime.

See Chapter 5, “Input and Output,” for more details and more examples.

Examples

Example 1: Some A, I, and F formats.

```
      READ( 2, 1 ) PART, ID, HEIGHT, WEIGHT
1     FORMAT( A8, 2X, I4, F8.2, F8.2 )
      WRITE( 9, 2 ) PART, ID, HEIGHT, WEIGHT
2     FORMAT( 'Part:', A8, ' Id:', I4, ' Height:', F8.2,
&               ' Weight:', F8.2 )
```

Example 2: Variable format expressions.

```
      DO 100 N = 1, 50
            …
1     FORMAT( 2X, F<N+1>.2 )
```

4.35 FUNCTION (External)

The FUNCTION statement identifies a program unit as a function subprogram.

Syntax

```
[ type ] FUNCTION  fun ( [ ar [, ar ] … ] )
```

type is one of the following:

BYTE ◆ CHARACTER CHARACTER*n CHARACTER*(*) COMPLEX COMPLEX*8 ◆ COMPLEX*16 ◆	COMPLEX*32 ◆ *(SPARC only)* DOUBLE COMPLEX ◆ DOUBLE PRECISION INTEGER INTEGER*2 ◆ INTEGER*4 ◆ LOGICAL	LOGICAL*1 ◆ LOGICAL*2 ◆ LOGICAL*4 ◆ REAL REAL*4 ◆ REAL*8 ◆ REAL*16 ◆*(SPARC only)*
n *(*as in CHARACTER**n)*	Must be greater than zero	
fun	Symbolic name assigned to function	
ar	Formal argument name	

An alternate nonstandard syntax for length specifier is as follows. ◆

```
[ type ] FUNCTION name [* m]([ ar [,ar] …])
```

m	Unsigned, nonzero integer constant specifying length of the data type.

Description

Type of Function

The function statement involves type, name, and formal parameter(s).

If *type* is not present in the FUNCTION statement, then the type of the function is determined by default and by any subsequent IMPLICIT or type statement. If *type* is present, then the function name cannot appear in other type statements.

Value of Function

The symbolic name of the function must appear as a variable name in the subprogram. The value of this variable, at the time of execution of the RETURN or END statement in the function subprogram, is the value of the function.

Formal Arguments

The list of arguments defines the number of formal arguments. The type of these formal arguments is defined by some combination of default, type statements, IMPLICIT statements, and DIMENSION statements.

The number of formal arguments must be the same as the number of actual arguments at the invocation of this function subprogram.

A function can assign values to formal arguments. These values are returned to the calling program when the RETURN or END statements are executed in the function subprogram.

Restrictions

Alternate return specifiers are not allowed in FUNCTION statements.

f77 provides recursive calls. A function or subroutine is *called recursively* if it calls itself directly. If it calls another function or subroutine which in turn calls this function or subroutine before returning, then it is also called recursively.

Examples

Example 1: Character function**.**

```
      CHARACTER*5 FUNCTION BOOL(ARG)
      BOOL = 'TRUE'
      IF (ARG .LE. 0) BOOL = 'FALSE'
      RETURN
      END
```

In the above example, BOOL is defined as a function of type CHARACTER with a length of 5 characters. This function when called returns the string "TRUE" or "FALSE" depending on the value of the variable ARG.

Example 2: Real function.

```
      FUNCTION SQR (A)
      SQR = A*A
      RETURN
      END
```

In the above example, the function SQR is defined as function of type REAL (by default) and returns the square of the number passed to it.

Example 3: Size of function, alternate syntax. ♦

```
      INTEGER FUNCTION FCN*2 ( A, B, C )
```

The above nonstandard form is treated as:

```
      INTEGER*2 FUNCTION FCN ( A, B, C )
```

4.36 GO TO (Assigned)

The *assigned* GO TO statement branches to a statement label identified by the assigned label value of a variable.

Syntax

GO TO *i* [[,] (*s* [, *s*] …)]	
i	Integer variable name
s	Statement label of an executable statement

Description

Execution proceeds as follows:

1. At the time an assigned GO TO statement is executed, the variable *i* must have been assigned the label value of an executable statement in the same program unit as the assigned GO TO statement.
2. If an assigned GO TO statement is executed, control transfers to a statement identified by *i*.

3. If a list of statement labels is present, the statement label assigned to *i* must be one of the labels in the list.

Restrictions

- *i* must be assigned by an ASSIGN statement in the same program unit as the GO TO statement.
- *s* must be in the same program unit as the GO TO statement.
- The same statement label can appear more than once in a GO TO statement.
- The statement you jump to must be *executable*, not DATA, ENTRY, FORMAT, or INCLUDE.
- You cannot jump into a DO, IF, ELSE IF, or ELSE block from outside the block.

Example

Example: Assigned GO TO.

```
      ASSIGN 10 TO N
      …
      GO TO N ( 10, 20, 30, 40 )
      …
10    CONTINUE
      …
40    STOP
```

4.37 GO TO (Computed)

The *computed* GO TO statement selects one statement label from a list, depending on the value of an integer or real expression, and transfers control to the selected one.

Syntax

GO TO (*s* [, *s*] …) [,] *e*	
s	Statement label of an *executable* statement
e	Expression of type integer or real

Description

Execution proceeds as follows

1. *e* is evaluated first. It is converted to integer, if required.
2. If $1 \le e \le n$, where *n* is the number of statement labels specified, then the e^{th} label is selected from the specified list and control is transferred to it.
3. If the value of *e* is outside the range, that is, $e < 1$ or $e > n$, then the computed GO TO statement serves as a CONTINUE statement.

Restrictions

- *s* must be in the same program unit as the GO TO statement.
- The same statement label can appear more than once in a GO TO statement.
- The statement you jump to must be *executable*, not DATA, ENTRY, FORMAT, or INCLUDE.
- You cannot jump into a DO, IF, ELSE IF, or ELSE block from outside the block.

Example

Example: Computed GO TO.

```
      …
      GO TO ( 10, 20, 30, 40 ), N
10    CONTINUE
      …
20    CONTINUE
      …
40    CONTINUE
```

In the above example:

- If N=1 then go to 10.
- If N=2 then go to 20.
- If N=3 then go to 30.
- If N=4 then go to 40.
- If N<1 or N>4 then fall through to 10.

4.38 GO TO (Unconditional)

The *unconditional* GO TO statement transfers control to a specified statement.

Syntax

GO TO *s*	
s	Statement label of an *executable* statement

Description

Execution of the GO TO statement transfers control to the statement labeled *s*.

Restrictions

- *s* must be in the same program unit as the GO TO statement.
- The statement you jump to must be executable, not DATA, ENTRY, FORMAT, or INCLUDE.
- You cannot jump into a DO, IF, ELSE IF, or ELSE block from outside the block.

Example

```
      A = 100.0
      B = 0.01
      GO TO 90
      …
 90   CONTINUE
```

4.39 IF (Arithmetic)

The *arithmetic* IF statement branches to one of three specified statements, depending on the value of an arithmetic expression.

Syntax

IF (*e*) *s1, s2, s3*	
e	Arithmetic expression (integer, real, double precision, or quadruple precision)
s1, s2, s3	Labels of *executable* statements

Description

The IF statement transfers control to the first, second, or third label if the value of the arithmetic expression is less than zero, equal to zero, or greater than zero, respectively.

Restrictions

- The *s1, s2, s3* must be in the same program unit as the IF statement.
- The same statement label can appear more than once in a IF statement.
- The statement you jump to must be *executable*, not DATA, ENTRY, FORMAT, or INCLUDE.
- You cannot jump into a DO, IF, ELSE IF, or ELSE block from outside the block.

Example

```
      N = 0
      IF ( N ) 10, 20, 30
```

Since the value of N is zero, control is transferred to statement label 20.

4.40 IF (Block)

The *block* IF statement executes one of two or more sequences of statements, depending on the value of a logical expression.

Syntax

```
IF (  e ) THEN
 …
END IF
```

e	A logical expression

Description

The *block* IF statement evaluates a logical expression and, if the logical expression is true, it executes a set of statements called the IF block. If the logical expression is false, control transfers to the next ELSE, ELSE IF, or END IF statement at the same IF-level.

IF-level

The IF-level of a statement *S* is the value *n1–n2*, where *n1* is the number of block IF statements from the beginning of the program unit up to the end, including *S*; and *n2* is the number of END IF statements in the program unit up to but not including *S*.

Example: In the following program, The IF-level of statement 9 is 2-1, or, 1.

```
      IF ( X .LT. 0.0 ) THEN
           MIN = NODE
      END IF
      …
 9    IF ( Y .LT. 0.0 ) THEN
           MIN = NODE - 1
      END IF
```

The IF-level of every statement must be zero or positive. The IF-level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The if-level of the END statement of each program unit must be zero.

IF-block

An IF block consists of all the executable statements following the block IF statement, up to, but not including, the next ELSE, ELSE IF, or END IF statement having the same if-level as the block IF statement. An IF block can be empty. In the following example, the two assignment statements form an IF block.

```
IF ( X .LT. Y ) THEN
     M = 0
     N = N+1
END IF
```

Execution proceeds as follows:

1. The logical expression *e* is evaluated first. If *e* is true, execution continues with the first statement of the IF block.
2. If *e* is true and the IF block is empty, control is transferred to the next END IF statement with the same if-level as the block IF statement.
3. If *e* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement with the same if-level as the Block IF statement.
4. If the last statement of the IF block does not result in a branch to a label, control is transferred to the next END IF statement that has the same if-level as the Block IF statement preceding the IF block.

Restrictions

You cannot jump into an IF block from outside the IF block.

Examples

Example 1: If-then-else.

```
IF ( L ) THEN
     N=N+1
     CALL CALC
ELSE
     K=K+1
     CALL DISP
END IF
```

Example 2: If-then-else-if with else-if.

```
      IF ( C .EQ. 'a' ) THEN
           NA=NA+1
           CALL APPEND
      ELSE IF ( C .EQ. 'b' ) THEN
           NB=NB+1
           CALL BEFORE
      ELSE IF ( C .EQ. 'c' ) THEN
           NC=NC+1
           CALL CENTER
      END IF
```

Example 3: Nested If-then-else.

```
      IF ( PRESSURE .GT 1000.0 ) THEN
           IF ( N .LT. 0.0 ) THEN
                X = 0.0
                Y = 0.0
           ELSE
                Z = 0.0
           END IF
      ELSE IF ( TEMPERATURE .GT. 547.0 ) THEN
           Z = 1.0
      ELSE
           X = 1.0
           Y = 1.0
      END IF
```

4.41 IF (Logical)

The *logical* `IF` statement executes one single statement, or does not execute it, depending on the value of a logical expression.

Syntax

`IF ( e ) st`	
e	Logical expression
st	Executable statement

Description

The *logical* `IF` statement evaluates a logical expression and executes the specified statement if the value of the logical expression is true. The specified statement is not executed if the value of the logical expression is false and execution continues as though a `CONTINUE` statement had been executed.

st can be any *executable* statement except a `DO` block, `IF`, `ELSE IF`, `ELSE`, `END IF`, `END`, or another logical `IF` statement.

Example

```
      IF ( VALUE .LE. ATAD ) CALL PUNT  ! Note that there is no THEN.
      IF ( TALLY .GE. 1000 ) RETURN
```

4.42 IMPLICIT

The `IMPLICIT` statement confirms or changes the default type of names.

Syntax

```
IMPLICIT  type  ( a [, a ] … ) [, type ( a [, a ] … ) ]
 or
IMPLICIT NONE ♦
 or
IMPLICIT UNDEFINED(A-Z) ♦
```

type is one of the following permitted types:

BYTE ◆	INTEGER	AUTOMATIC ◆
CHARACTER	INTEGER*2 ◆	STATIC ◆
CHARACTER*n	INTEGER*4 ◆	
CHARACTER*(*)	LOGICAL	
COMPLEX	LOGICAL*1 ◆	
COMPLEX*8 ◆	LOGICAL*2 ◆	
COMPLEX*16 ◆	LOGICAL*4 ◆	
COMPLEX*32 ◆ *(SPARC only)*	REAL	
DOUBLE COMPLEX ◆	REAL*4 ◆	
DOUBLE PRECISION	REAL*8 ◆	
	REAL*16 ◆ *(SPARC only)*	

n must be greater than 0

a is either a single letter or a range of single letters in alphabetical order. A range of letters can be specified by the first and last letters of the range, separated by a minus sign.

Description

Implicit Typing

The `IMPLICIT` statement can also indicate that no implicit typing rules apply in a program unit.

An `IMPLICIT` statement specifies a type and size for all user-defined names that begin with any letter, either a single letter or in a range of letters, appearing in the specification.

An `IMPLICIT` statement does not change the type of the intrinsic functions.

An `IMPLICIT` statement applies only to the program unit that contains it.

A program unit can contain more than one `IMPLICIT` statement.

`IMPLICIT` types for particular user names are overridden by a *type* statement.

No Implicit Typing

The second form of `IMPLICIT` specifies that no implicit typing should be done for user-defined names, and all user-defined names shall have their types declared explicitly.

If either `IMPLICIT NONE` or `IMPLICIT UNDEFINED (A-Z)` is specified, there cannot be any other `IMPLICIT` statement in the program unit.

Restrictions

- `IMPLICIT` statements must precede all other specification statements.
- The same letter can appear *more than once* as a single letter, or in a range of letters in all `IMPLICIT` statements of a program unit. ♦
  The FORTRAN Standard restricts this to *only once*. But for `f77`, if a letter is used twice, each usage is declared in order. Note example 4.

Examples

Example 1: IMPLICIT, everything is integer.

```
      IMPLICIT INTEGER (A-Z)
      X = 3
      K = 1
      STRING = 0
```

Example 2: Complex if it starts with U, V, or W; character if it starts with C or S.

```
      IMPLICIT COMPLEX (U,V,W), CHARACTER*4 (C,S)
      U1 = ( 1.0, 3.0)
      STRING = 'abcd'
      I = 0
      X = 0.0
```

Example 3: All items must be declared. ♦

```
      IMPLICIT NONE
      CHARACTER STR*8
      INTEGER N
      REAL Y
      N = 100
      Y = 1.0E5
      STR = 'Length'
```

In the above example, once IMPLICIT NONE is specified in the beginning, all the variables *must* be declared explicitly.

Example 4: A letter used twice. ♦

```
      IMPLICIT INTEGER (A-Z)
      IMPLICIT REAL (A-C)
      C = 1.5E8
      D = 9
```

In the above example, D through Z implies INTEGER, and A through C implies REAL.

4.43 INCLUDE

The `INCLUDE` ♦ statement inserts a file into the source program.

Syntax

`INCLUDE '`*`file`*`'`
or
`INCLUDE "`*`file`*`"`	

`file`	Name of the file to be inserted

Description

The contents of the named file replace the `INCLUDE` statement.

Searchpath

If the name referred to by the `INCLUDE` statement begins with the character '/', then it is taken by `f77` to mean the absolute path name of the include file. Otherwise, `f77` looks for the file in the following directories, in this order:

1. The directory containing the source file with the `INCLUDE` statement
2. The current directory in which the `f77` command was issued
3. The default list. This is different in Solaris 1.x and 2.x.

   Solaris 2.x:

   If you installed into the *standard* directory, the default list is

   `/opt/SUNWspro/SC3.0.1/include/f77 /usr/include`

   If you installed into *nonstandard* directory */mydir/*, then it is

   /mydir`/SUNWspro/SC3.0.1/include/f77 /usr/include`

   Solaris 1.x:

   If you installed in the *standard* directory, then the default list is

   /usr/lang/SC3.0.1/include/f77 /usr/include

If you installed into *nonstandard* directory */mydir/*, then it is

/mydir/SC3.0.1/include/f77 /usr/include

Remarks:

- The "SC3.0.1" varies with the release of the set of compilers.
- These INCLUDE statements can be nested ten deep.

Preprocessor #include

The paths and order searched for the INCLUDE statement are not the same as those searched for the preprocessor #include directive, described under -I in the *User's Guide*. Files included by the preprocessor #include directive can contain #defines and the like, while files included with the compiler INCLUDE statement must contain only FORTRAN statements.

VMS Logical File Names in the INCLUDE *statement*

f77 interprets VMS logical file names on the INCLUDE statement if:

1. The -xl[d] compiler option is set
2. The environment variable LOGICALNAMEMAPPING is there to define the mapping between the logical names and the UNIX path name.

f77 uses the following rules for the interpretation:

- The environment variable should be set to a string with the syntax:

```
"lname1=path1; lname2=path2; …"
```

  where each *lname* is a logical name and each *path1*, *path2*, and so forth. is the path name of a directory (without a trailing '/').

- All blanks are ignored when parsing this string. It strips any trailing "/[no]list" from the file name in the INCLUDE statement.
- Logical names in a file name are delimited by the first ":" in the VMS file name, so f77 converts file names of the "lname1:file" form to the "path1/file" form.
- For logical names, uppercase/lowercase is significant. If a logical name is encountered on the INCLUDE statement which is not specified in the LOGICALNAMEMAPPING, the file name is used unchanged.

Examples

Example 1: `INCLUDE`, simple case.

```
INCLUDE 'stuff'
```

The above line is replaced by the contents of the file `stuff`.

Example 2: `INCLUDE`, search paths.

For the following conditions:

- Your source file has the line

```
INCLUDE 'ver1/const.h'
```

- Your current working directory is `/usr/ftn`
- Your source file is `/usr/ftn/projA/myprg.f`

In this example, `f77` seeks `const.h` in these directories, in the order shown.

If you installed into the *standard* directory, then `f77` searches these:

Solaris 2.x

```
/usr/ftn/projA/ver1/
/usr/ftn/ver1/
/opt/SUNWspro/SC3.0.1/include/f77/ver1/
/usr/include
```

If you installed into *nonstandard* directory /*mydir*/, it searches these:

```
/usr/ftn/projA/ver1/
/usr/ftn/ver1/
/mydir/SUNWspro/SC3.0.1/include/f77/ver1/
/usr/include
```

If you installed into the *standard* directory, then `f77` searches these:

Solaris 1.x

```
/usr/ftn/projA/ver1/
/usr/ftn/ver1/
/usr/lang/SC3.0.1/include/f77/ver1/
/usr/include
```

If you installed into *nonstandard* directory /*mydir*/, it searches these:

```
/usr/ftn/projA/ver1/
/usr/ftn/ver1/
/mydir/SC3.0.1/include/f77/ver1/
/usr/include
```

4.44 INQUIRE

The INQUIRE statement returns information about a unit or file.

Syntax

An inquire by *unit* has the general form:

```
INQUIRE( [ UNIT=] u, slist )
```

An inquire by *file* has the general form:

INQUIRE(FILE=*fn*, *slist*)	
fn	Name of the file being queried
u	Unit of the file being queried
slist	Specifier list

The INQUIRE *slist* can include one or more of the following, in any order:

- ERR = *s*
- EXIST = *ex*
- OPENED = *od*
- NAMED = *nmd*
- ACCESS = *acc*
- SEQUENTIAL = *seq*
- DIRECT = *dir*
- FORM = *fm*
- FORMATTED = *fmt*
- UNFORMATTED = *unf*
- NAME = *fn*
- BLANK = *blnk*

- OSTAT = *ios*
- NUMBER = *num*
- RECL = *rcl*
- NEXTREC = *nr*

Description

You can determine such things about a file as whether it exists, is opened, or is connected for sequential I/O. That is, files have such attributes as name, existence (or nonexistence), and the ability to be connected in certain ways (FORMATTED, UNFORMATTED, SEQUENTIAL, or DIRECT).

You can inquire either by unit or by file, but not by both in the same INQUIRE statement.

In this system environment, the only way to discover what permissions you have for a file is to use the ACCESS (3F) function. The INQUIRE statement does not determine permissions.

The specifiers for INQUIRE are described as follows.

FILE = *fn*

n is a character expression or * with the name of the file. Trailing blanks in the file name are ignored. If the file name is all blanks, that means the current *directory*. The file need not be connected to a unit in the current program.

UNIT = *u*

u is an integer expression or * with the value of the unit.

Exactly one of FILE or UNIT must be used.

IOSTAT = *ios*

ios is as in the OPEN statement.

ERR = *s*

s is a statement label of a statement to branch to if an error occurs during the execution of the INQUIRE statement.

EXIST = *ex*

ex is a logical variable that is set to .TRUE. if the file or unit exists and .FALSE. otherwise.

OPENED = *od*

od is a logical variable that is set to .TRUE. if the file is connected to a unit or the unit is connected to a file, and .FALSE. otherwise.

NUMBER = *num*

num is an integer variable that is assigned the number of the unit connected to the file, if any. If no file is connected, the variable is unchanged.

NAMED = *nmd*

nmd is a logical variable that is assigned .TRUE. if the file has a name, .FALSE. otherwise.

NAME = *fn*

fn is a character variable that is assigned the name of the file connected to the unit. If you do an inquire-by-unit, the name parameter is undefined unless both the values of the OPENED and NAMED variables are both true. If you do an inquire-by-file, the name parameter is returned, even though the FORTRAN Standard leaves it undefined.

ACCESS = *acc*

acc is a character variable that is assigned the value 'SEQUENTIAL' if the connection is for sequential I/O and 'DIRECT' if the connection is for direct I/O. The value is undefined if there is no connection.

SEQUENTIAL = *seq*

seq is a character variable that is assigned the value 'YES' if the file could be connected for sequential I/O, 'NO' if the file could not be connected for sequential I/O, and 'UNKNOWN' if the system can't tell.

DIRECT = *dir*

dir is a character variable that is assigned the value 'YES' if the file could be connected for direct I/O, 'NO' if the file could not be connected for direct I/O, and 'UNKNOWN' if the system can't tell.

FORM = *fm*

fm is a character variable which is assigned the value 'FORMATTED' if the file is connected for formatted I/O and 'UNFORMATTED' if the file is connected for unformatted I/O.

FORMATTED = *fmt*

fmt is a character variable that is assigned the value 'YES' if the file could be connected for formatted I/O, 'NO' if the file could not be connected for formatted I/O, and 'UNKNOWN' if the system can't tell.

UNFORMATTED = *unf*

unf is a character variable that is assigned the value 'YES' if the file could be connected for unformatted I/O, 'NO' if the file could not be connected for unformatted I/O, and 'UNKNOWN' if the system can't tell.

RECL = *rcl*

rcl is an integer variable that is assigned the record length of the records in the file if the file is connected for direct access. f77 does not ever adjust the *rcl* returned by INQUIRE. The OPEN statement does just such an adjustment if the -xl[d] option is set.

NEXTREC = *nr*

nr is an integer variable that is assigned one more than the number of the last record read from a file connected for direct access.

BLANK = *blnk*

blnk is a character variable that is assigned the value 'NULL' if null blank control is in effect for the file connected for formatted I/O and 'ZERO' if blanks are being converted to zeros and the file is connected for formatted I/O.

Example: An OPEN statement in which declarations are *omitted.*

```
      OPEN( 1, FILE='/dev/console' )
```

For f77 this statement opens the console for formatted sequential I/O. An INQUIRE for either unit 1 or file /dev/console would reveal that the file has the following aspects. It has the following aspects:

- Exists
- Is connected to unit 1

- Has the name `/dev/console`
- Is opened for sequential I/O
- Could be connected for sequential I/O
- Can't be connected for direct I/O (can't seek)
- Is connected for formatted I/O
- Can be connected for formatted I/O
- Can't be connected for unformatted I/O (can't seek)
- Has neither a record length nor a next record number
- Is ignoring blanks in numeric fields

Table 4-2 INQUIRE Options Summary

Form: SPECIFIER = Variable

SPECIFIER	*Value of Variable*	*Data type of Variable*
ACCESS	'DIRECT' 'SEQUENTIAL'	CHARACTER
BLANK	'NULL','ZERO'	CHARACTER
DIRECT *	'YES' 'NO' 'UNKNOWN'	CHARACTER
ERR	Statement number	INTEGER
EXIST	.TRUE., .FALSE.	LOGICAL
FORM	'FORMATTED' 'UNFORMATTED'	CHARACTER
FORMATTED *	'YES' 'NO' 'UNKNOWN'	CHARACTER
IOSTAT	Error number	INTEGER
NAME †	Name of the file	CHARACTER
NAMED †	.TRUE., .FALSE.	LOGICAL
NEXTREC	Next record number	INTEGER
NUMBER *	Unit number	INTEGER
OPENED	.TRUE., .FALSE.	LOGICAL
RECL	Record length	INTEGER
SEQUENTIAL *	'YES' 'NO' 'UNKNOWN'	CHARACTER
UNFORMATTED *	'YES' 'NO' 'UNKNOWN'	CHARACTER

The asterisk (*) indicates the returned value is undefined for inquire-by-unit in the FORTRAN Standard, but is defined in f77.

The † indicates the returned value is undefined for inquire-by-file in the FORTRAN Standard, but is defined in f77.

- If a file is scratch, then NAMED and NUMBER are not returned.
- If there is no file with the specified name, then these are not returned: DIRECT, FORMATTED, NAME, NAMED, SEQUENTIAL, and UNFORMATTED.
- If OPENED=.FALSE., then these are not returned: ACCESS, BLANK, FORM, NEXTREC, and RECL.
- If no file is connected to the specified unit, then these are not returned: ACCESS, BLANK, DIRECT, FORM, FORMATTED, NAME, NAMED, NEXTREC, NUMBER, RECL, SEQUENTIAL, and UNFORMATTED.
- If ACCESS='SEQUENTIAL', then these are not returned: RECL and NEXTREC.
- If FORM='UNFORMATTED', then BLANK is not returned.

Examples

Example 1: Inquire by *unit*.

```
      LOGICAL OK
      INQUIRE( UNIT=3, OPENED=OK )
      IF ( OK ) CALL GETSTD ( 3, STDS )
```

Example 2: Inquire by *unit* — omit the UNIT=.

```
      LOGICAL OK
      INQUIRE( 3, OPENED=OK )
      IF ( OK ) CALL GETSTD ( 3, STDS )
```

Example 3: Inquire by *file.*

```
      LOGICAL THERE
      INQUIRE( FILE='.profile', EXIST=THERE )
      IF ( THERE ) CALL GETPROFILE( FC, PROFILE )
```

Example 4: More than one answer.

```
      CHARACTER FN*32
      LOGICAL HASNAME, OK
      INQUIRE ( UNIT=3, OPENED=OK, NAMED=HASNAME, NAME=FN )
      IF ( OK .AND. HASNAME ) PRINT *, 'Filename="', FN, '"'
```

4.45 INTEGER

The `INTEGER` statement specifies the type to be integer for a symbolic constant, variable, array, function, or dummy function.

Optionally, it specifies array dimensions and size and initializes with values.

Syntax

`INTEGER [* len[,]] v [* len [/c/]] [, v [* len [/c/]] …`	
v	Name of a symbolic constant, variable, array, array declarator, function, or dummy function
len	Either 2 or 4, the length in bytes of the symbolic constant, variable, array element, or function
c	List of constants for the immediately preceding name

Description

If you specify the size as 2 or 4, you get what you specify; if you do *not* specify the size, you get the default size.

Default Size

The default size depends on `-i2` and `-r8`.

- If the `-i2` option is on the `f77` command line, then the default length is 2; otherwise, the default is 4.
- If the `-r8` option is on the `f77` command line, then the compiler allocates 8 bytes, but still does only 4-byte arithmetic. This is done to satisfy the requirements of the FORTRAN Standard that an integer and a real datum are allocated the same amount of storage.
- If you put both `-i2` and `-r8` on the `f77` command line, the results are unpredictable.

Examples

Example 1: Integer scalars. Each of these is equivalent to the others, if there is no `-i2`. (Don't use all three lines in the same program unit — you cannot declare anything more than once in the same program unit.)

```
      INTEGER   U, V
      INTEGER*4 U, V
      INTEGER   U*4, V*4
```

Example 2: Initialize.

```
      INTEGER U / 1 /, V / 4 /, W*2 / 1 /, X*2 / 4 /
```

Example 3: Integer arrays. Use any one of these lines; they are equivalent.

```
      INTEGER   U(9), V(9)
      INTEGER*4 U(9), V(9)
      INTEGER   U*4(9), V(9)*4
```

4.46 INTRINSIC

The `INTRINSIC` statement lists intrinsic functions that can be passed as actual arguments.

Syntax

`INTRINSIC` *fun* [, *fun*] …	
fun	Function name

Description

If the name of an intrinsic function is used as an actual argument, it must appear in an `INTRINSIC` statement in the same program unit.
Example.

The following example shows intrinsic functions passed as actual arguments:

```
      INTRINSIC SIN, COS
      X = CALC ( SIN, COS )
```

Restrictions

- A symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in the same program unit.
- The *actual* argument must be a *specific* name. Most generic names are also specific, but a few are not: IMAG, LOG, and LOG10.
- A symbolic name can appear *more than once* in an INTRINSIC statement.♦ In the FORTRAN Standard a symbolic name can appear *only once* in an INTRINSIC statement.
- Because they are in-line, the following *cannot* be passed as *actual* arguments.

Table 4-3 Intrinsics That Cannot Be Actual Arguments

LOC	IIQINT	QEXTD	MIN	LOG
AND	JIQINT	QFLOAT	MIN0	LOG10
IAND	IFIX	CMPLX	AMIN0	QREAL
IIAND	IIFIX	DCMPLX	AIMIN0	QCMPLX
JIAND	JIFIX	ICHAR	AJMIN0	
OR	IDINT	IACHAR	IMIN0	
IOR	IIDINT	ACHAR	JMIN0	
IIOR	JIDINT	CHAR	MIN1	
IEOR	FLOAT	MAX	AMIN1	
IIEOR	FLOATI	MAX0	DMIN1	
JIOR	FLOATJ	AMAX0	IMIN1	
JIEOR	DFLOAT	AIMAX0	JMIN1	
NOT	DFLOATI	AJMAX0	QMIN1	
INOT	DFLOATJ	IMAX0	IMAG	
JNOT	SNGL	JMAX0	EPBASE	
XOR	SNGLQ	MAX1	EPEMAX	
LSHIFT	REAL	AMAX1	EPEMIN	
RSHIFT	DREAL	DMAX1	EPHUGE	
INT	DBLE	IMAX1	EPMRSP	
IINT	DBLEQ	JMAX1	EPPREC	
JINT	QEXT	QMAX1	EPTINY	
IQINT				

4.47 LOGICAL

The LOGICAL statement specifies the type to be logical for a symbolic constant, variable, array, function, or dummy function.

Optionally, it specifies array dimensions and initializes with values.

Syntax

LOGICAL [*len[,]] v[* len [/c/]] [, v [* len [/c/]] …	
v	Name of a symbolic constant, variable, array, array declarator, function, or dummy function
len	Either 1, 2, or 4, the length in bytes of the symbolic constant, variable, array element, or function
c	List of constants for the immediately preceding name

Description

If you specify the size as 1, 2, or 4, then you get what you specify; but if you do *not* specify the size, you get the default size.

Default Size

The default size depends on –i2 and –r8.

- If the –i2 option is on the f77 command line, then the default length is 2; otherwise, the default is 4.
- If the –r8 option is on the f77 command line, then the compiler allocates 8 bytes, but still does only 4-byte arithmetic. This is done to satisfy the requirements of the FORTRAN Standard that an integer and a real datum are allocated the same amount of storage.
- If you put both –i2 and –r8 on the f77 command line, the results are unpredictable.

Examples

Example 1: Each of these statements is equivalent to the others, if there is no `-i2`. (Don't use all three statements in the same program unit — you cannot declare anything more than once in the same program unit.)

```
      LOGICAL U, V
      LOGICAL*4 U, V
      LOGICAL U*4, V*4
```

Example 2: Initialize.

```
      LOGICAL U /.false./, V /0/, W*4 /.true./, X*4 /'z'/
```

4.48 MAP

The `MAP` ♦ declaration defines alternate groups of fields in a *union*.

Syntax

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

Description

Each field declaration can be one of the following:

- Type declaration (can include initial values)
- Substructure (either another structure declaration, or a record that has been previously defined)
- Union declaration (See Section 4.70, "UNION and MAP," for more detail.)

Example

Example: MAP (See Section 4.70, "UNION and MAP," for details.)

```
      STRUCTURE /STUDENT/
           CHARACTER*32  NAME
           INTEGER*2  CLASS
           UNION
                MAP
                        CHARACTER*16 MAJOR
                END MAP
                MAP
                        INTEGER*2  CREDITS
                        CHARACTER*8  GRAD_DATE
                END MAP
           END UNION
      END STRUCTURE
```

4.49 NAMELIST

The NAMELIST ♦ statement defines a list of variables or array names, and associates it with a unique group name.

Syntax

NAMELIST / *grname* / *namelist* [[,] / *grname* / *namelist*] …	
grname	Symbolic name of the group
namelist	List of variables and arrays

Description

Group Name

The group name is used in the namelist-directed I/O statement to identify the list of variables or arrays that are to be read or written. This name is used by namelist-directed I/O statements instead of an input/output list. The group name must be unique and identify a list whose items can be read or written.

A group of variables can be defined through several NAMELIST statements with the same group name. Together, these definitions are taken as defining one NAMELIST group.

Namelist Items

The namelist items can be of any data type. The items in the namelist can be variables or arrays and can appear in more than one namelist. Only the items specified in the namelist can be read or written in namelist-directed I/O, but it is not necessary to specify data in the input record for every item of the namelist.

The order of the items in the namelist controls the order in which the values are written in namelist-directed output. The items in the input record can be in any order.

Restrictions

- Input data can assign values to the elements of arrays or to substrings of strings that appear in a namelist.
- The following *cannot* appear in a NAMELIST statement:
  - Constants (parameters),
  - Array elements
  - Records and record fields
  - Character substrings
  - Dummy arrays (with nonconstant dimension specifiers)
  - Automatic variables and arrays

See Chapter 5, “Input and Output,” for more on namelist.

Example

Example: The NAMELIST statement.

```
      CHARACTER*16 SAMPLE
      LOGICAL*4 NEW
      REAL*4 DELTA
      NAMELIST /CASE/ SAMPLE, NEW, DELTA
```

In this example, the group CASE has three variables SAMPLE, NEW, and DELTA.

4

4.50 OPEN

The OPEN statement connects an existing external file to a unit, or creates a file and connects it to a unit, or changes some specifiers of the connection.

Syntax

OPEN(*KEYWORD1=value1, KEYWORD2=value2, …*)	
KEYWORDn	A valid keyword specifier, as listed below

Description

For tape, it is more reliable to use the TOPEN() routines. The OPEN statement determines the type of file named, whether the connection specified is legal for the file type (for instance, DIRECT access is illegal for tape and tty devices), and allocates buffers for the connection if the file is on tape or if the subparameter FILEOPT='BUFFER=*n*' is specified. Existing files are never truncated on opening. The options can be specified in any order.

Table 4-4 OPEN Keyword Specifier Summary

Standard Form	Alternate Form
[UNIT=] *u*	
FILE = *fin*	NAME = *fin*
ACCESS = *acc*	
BLANK = *blnk*	
ERR = *s*	
FORM = *fm*	
IOSTAT = *ios*	
RECL = *rl*	RECORDSIZE = *rl*
STATUS = *sta*	TYPE = *sta*
FILEOPT = *fopt* ♦	

The alternate forms give a warning if the –ansi flag is set.

Details of the OPEN keyword specifier are listed in the following table.

Table 4-5 OPEN Keyword Specifier Details

[UNIT=] *u*	
	u is an integer expression or an asterisk (*) that specifies the unit number. The *u* is required. If the *u* is first in the parameter list, then the "UNIT=" can be omitted.
FILE=*fin*	
	fin is a character expression or * naming the file to open. An OPEN statement need not specify a file name. If not specified, a default file name is created.
	Reopen
	If you open a unit that's already open without specifying a file name (or with the previous file name), FORTRAN thinks you are reopening the file to change parameters. The file position is not changed. The only parameters you are allowed to change are BLANK (NULL or ZERO) and FORM (FORMATTED or PRINT). To change any other parameters, you must close, then reopen the file.
	Switch Files
	If you open a unit that's already open, but you specify a different file name, it is as if you closed with the old file name before the open.
	Switch Units
	If you open a file that's already open, but you specify a different unit, that is an error. This error is *not* caught by the ERROR= option, and the program will *not* terminate abnormally.
	Scratch
	If a file is opened with STATUS='SCRATCH', a temporary file is created and opened. See STATUS=*sta*
ACCESS=*acc*	
	The ACCESS=*acc* clause is optional. *acc* is a character expression. Possible values are: APPEND, DIRECT, or SEQUENTIAL. The default is SEQUENTIAL.
	If ACCESS='APPEND':
	SEQUENTIAL and FILEOPT='EOF' are assumed. This is for opening a file to append records to an existing sequential-access file. Only WRITE operations are allowed . This is an extension. ♦

Table 4-5 OPEN Keyword Specifier Details *(Continued)*

	If ACCESS='DIRECT':
	RECL must also be given, since all I/O transfers are done in multiples of fixed-size records. Only directly accessible files are allowed; thus, tty, pipes, and magnetic tape are not allowed. If you build a file as sequential, then you cannot access it as direct. If FORM is not specified, unformatted transfer is assumed. If FORM='UNFORMATTED', the size of each transfer depends upon the data transferred.
	If ACCESS='SEQUENTIAL':
	RECL is ignored. ♦ The FORTRAN Standard prohibits RECL for sequential access. No padding of records is done. If you build a file as direct, then you cannot access it as sequential. Files don't have to be randomly accessible, in the sense that tty, pipes, and tapes can be used. But for tapes we recommend the TOPEN() routines because they are more reliable. If FORM is not specified, formatted transfer is assumed. If FORM='FORMATTED', each record is terminated with a newline (\n) character. This means that each record actually has one extra character. If FORM='PRINT', the file acts like a FORM='FORMATTED' file, except for interpretation of column-1 characters on output (0 = double space, 1 = form feed, and blank = single space). If FORM='UNFORMATTED', each record is preceded and terminated with an INTEGER*4 count, making each record 8 characters longer than normal. This convention is not shared with other languages, so it is useful only for communicating between FORTRAN programs.
FORM=*fm*	
	The FORM=*fm* clause is optional. The *fm* is a character expression. Possible values are 'FORMATTED', 'UNFORMATTED', or 'PRINT'. ♦ The default is 'FORMATTED'. This option interacts with ACCESS. The 'PRINT' makes it a *print* file. See Chapter 5, "Input and Output," for details.

Table 4-5 OPEN Keyword Specifier Details *(Continued)*

RECL=*rl*	
	The RECL=*rl* clause is required if ACCESS='DIRECT' and ignored otherwise. The *rl* is an integer expression for the length in characters of each record of a file. *rl* must be positive. If the record length is unknown, you can use RECL=1; see *Direct Access I/O* on page 289. If -xl[d] is *not* set, *rl* is number of characters, and record length is *rl*. If -xl[d] is set, *rl* is number of words, and record length is *rl**4. ♦ There are more details in the ACCESS='SEQUENTIAL' section, above. Each WRITE defines one record and each READ reads one record (unread characters are flushed). The default buffer size for tape is 64K characters. But for tapes we recommend the TOPEN() routines because they are more reliable.
ERR=*s*	
	The ERR=*s* clause is optional. The *s* is a statement label of a statement to branch to if an error occurs during execution of the OPEN statement.
IOSTAT=*ios*	
	The IOSTAT=*ios* clause is optional. The *ios* is an integer variable that receives the error status from an OPEN. After the execution of the OPEN, if no error condition exists, then *ios* is zero, otherwise it is some positive number. If you want to avoid aborting the program when an error occurs on an OPEN, include "ERR=*s*" o "IOSTAT=*ios*".
BLANK=*blnk*	
	The BLANK=*blnk* clause is optional. The *blnk* is a character expression that indicates how blanks are treated. Possible values are: 'ZERO' (blanks are treated as zeroes) 'NULL' (blanks are ignored during numeric conversion) Default: 'NULL' This clause is for formatted input only.

Table 4-5 OPEN Keyword Specifier Details *(Continued)*

STATUS=*sta*	
	The STATUS=*sta* clause is optional. The *sta* is a character expression. Possible values are: 'OLD', 'NEW', 'UNKNOWN', or 'SCRATCH'. The default is 'UNKNOWN'. 'OLD' — The file already exists (nonexistence is an error). For example: STATUS='OLD' 'NEW' — The file doesn't exist (existence is an error). If 'FILE=*name*' is not specified, then a file named 'fort.*n*' is opened, where *n* is the specified logical unit. 'UNKNOWN' — Existence is unknown (the default). 'SCRATCH' — For a file opened with STATUS='SCRATCH', a temporary file with a name of the form tmp.F*AAAxnnnnn* is opened. Any other STATUS specifier without an associated file name results in opening a file named 'fort.*n*', where *n* is the specified logical unit number. By default, a scratch file is deleted when closed or during normal termination. If the program aborts, then the file may not get deleted. To prevent deletion, CLOSE with STATUS='KEEP'. The FORTRAN Standard prohibits opening a named file as scratch: if OPEN has a FILE=*name* option, then it cannot have a STATUS='SCRATCH' option. This Fortran extends the standard by allowing opening named files as scratch. ♦ Such files are normally deleted when closed or at normal termination. TMPDIR: Fortran programs normally put *scratch* files in the current working directory. If the TMPDIR environment variable is set to a writeable directory, then the program puts sc*ratch* files there. ♦
FILEOPT=*fopt* ♦	
	The FILEOPT=*fopt* clause is optional. The *fopt* is a character expression. Possible values are:
	'NOPAD' — Don't extend records with blanks if you read past the end-of-record (formatted input only). That is, a *short* record causes an abort with an error message, rather than just filling with trailing blanks and continuing. 'BUFFER=*n*' — This suboption is for either disk or magnetic tape. But for tapes we recommend the TOPEN() routines because they are more reliable. It sets the size in bytes of the I/O buffer to use. It is necessary only when writing to a new file, since the I/O system defaults to 64K-character buffers for tape, allowing reads to anything smaller than that. For writes, larger buffers yield faster I/O. For good performance, make the buffer a multiple of the largest record size. This can be larger than actual physical memory, and probably the very best performance is obtained by making the record size equal to the entire file size. Larger buffer sizes can cause extra paging. For tape only, it must be at least 8 characters greater than the largest record you write; this is to avoid spanning tape blocks.
	'EOF' —This opens a file at end-of-file rather than at the beginning (useful for appending data to file). Example: FILEOPT='EOF'. Unlike ACCESS='APPEND', in this case both READ and BACKSPACE are allowed.

Examples

Example 1: Open a file and connect it to unit 8. Either of the following forms of the OPEN statement will open the file projectA/data.test and connect it to FORTRAN unit 8.

```
      OPEN( UNIT=8, FILE='projectA/data.test' )
      OPEN( 8, FILE='projectA/data.test' )
```

In the above example, the following properties are established by *default*: sequential access, formatted file, and (unwisely) no allowance for error during file open.

Example 2: Explicitly specify properties.

```
      OPEN( UNIT=8, FILE='projectA/data.test',
&             ACCESS='SEQUENTIAL', FORM='FORMATTED' )
```

Example 3: Either of these opens file fort.8 and connects it to unit 8.

```
      OPEN( UNIT=8 )
      OPEN( 8 )
```

In the above example, you get sequential access, formatted file, and no allowance for error during file open. If the file fort.8 does not exist before execution, it is created. The file remains after termination.

Example 4: Allowing for open errors.

```
      OPEN( UNIT=8, FILE='projectA/data.test', ERR=99 )
```

The above statement branches to 99 if an error occurs during the OPEN.

Example 5: Allowing for variable-length records.

```
OPEN( 1, ACCESS='DIRECT', recl=1 )
```

For more on variable-length records, see *Direct Access I/O* on page 289.

Example 6: Scratch file.

```
OPEN( 1, STATUS='SCRATCH' )
```

This opens a temporary file with a name such as tmp.FAAAa003zU. The file is usually in the current working directory, or in TMPDIR if that variable is set.

4

4.51 OPTIONS

The OPTIONS♦ statement overrides compiler command-line options.

Syntax

```
OPTIONS /qualifier  [/qualifier …]
```

Description

The OPTIONS statement qualifiers are:

Table 4-6 OPTIONS Statement Qualifiers

Qualifier	***Action Taken***
/[NO]G_FLOATING	None (not implemented)
/[NO]I4	Enables/Disables the -i2 option
/[NO]F77	None (not implemented)
/CHECK=ALL	Enables the -C option
/CHECK=[NO]OVERFLOW	None (not implemented)
/CHECK=[NO]BOUNDS	Disables/Enables the -C option
/CHECK=[NO]UNDERFLOW	None (not implemented)
/CHECK=NONE	Disables the -C option
/NOCHECK	Disables the -C option
/[NO]EXTEND_SOURCE	Disables/enables the -e option

Restrictions

- The OPTIONS statement must be the *first* statement in a program unit. Note that this means it must be before the BLOCK DATA, FUNCTION, PROGRAM, and SUBROUTINE statements.
- Options set by the OPTIONS statement override those of the command line.
- Options set by the OPTIONS statement endure for that program unit only.

- A qualifier can be abbreviated to four or more characters.
- Uppercase or lowercase is not significant.

Example

For the following source, integer variables declared with no explicit size will occupy 4 bytes rather than 2, with or without the `-i2` option on the command line. This does *not* change the size of integer constants, only variables.

```
OPTIONS /I4
PROGRAM FFT
…
END
```

By way of contrast, if you use `/NOI4`, then all integer variables declared with no explicit size occupy 2 bytes rather than 4, with or without the `-i2` option on the command line. But integer constants occupy 2 bytes with `-i2`, and 4 bytes otherwise.

4.52 PARAMETER

The `PARAMETER` statement assigns a symbolic name to a constant.

Syntax

`PARAMETER ( p=e [, p=e ] … )`	
p	Symbolic name
e	Constant expression

An alternate syntax is allowed, if the `-xl` flag is set. ♦

```
PARAMETER p=e [, p=e ] …
```

In this alternate form, the type of the constant expression determines the type of the name; no conversion is done.

Description

e can be of any type and the type of symbolic name and the corresponding expression must match.

A symbolic name can be used to represent the real part, imaginary part, or both parts of a complex constant.

A constant expression is made up of explicit constants and parameters and the FORTRAN operators. See Section 3.6, "Constant Expressions," for more detail.

No structured records or record fields are allowed in a constant expression.

Exponentiation to a floating-point power is not allowed, and a warning is issued.

If the type of the data expression does not match the type of the symbolic name, then the type of the name must be specified by a type statement or `IMPLICIT` statement prior to its first appearance in a `PARAMETER` statement, otherwise conversion will be performed.

If a `CHARACTER` statement explicitly specifies the length for a symbolic name, then the constant in the `PARAMETER` statement can be no longer than that length. Longer constants are truncated, and a warning is issued. The `CHARACTER` statement must appear before the `PARAMETER` statement.

If a `CHARACTER` statement uses `*(*)` to specify the length for a symbolic name, then the data in the `PARAMETER` statement are used to determine the length of the symbolic constant. The `CHARACTER` statement must appear before the `PARAMETER` statement.

Any symbolic name of a constant that appears in an expression *e* must have been defined previously in the same or a different `PARAMETER` statement in the same program unit.

Restrictions

- A symbolic constant must not be defined more than once in a program unit.
- If a symbolic name appears in a `PARAMETER` statement, then it cannot represent anything else in that program unit.
- A symbolic name cannot be used in a constant format specification, but it can be used in a variable format specification.

- If you pass a parameter as an argument, and the subprogram tries to change it, you may get a runtime error.

Examples

Example 1: Some real, character, and logical parameters.

```
      CHARACTER HEADING*10
      LOGICAL T
      PARAMETER ( EPSILON=1.0E-6, PI=3.141593,
&                 HEADING='IO Error #',
&                 T=.TRUE. )
      …
```

Example 2: Let the compiler count the characters.

```
      CHARACTER HEADING*(*)
      PARAMETER ( HEADING='I/O Error Number' )
      …
```

Example 3: The alternate syntax, if the `-xl` flag is set.

```
      PARAMETER FLAG1 = .TRUE.
```

The above statement is treated as:

```
      LOGICAL FLAG1
      PARAMETER (FLAG1 = .TRUE.)
```

Note that an ambiguous statement that could be interpreted as either a PARAMETER statement or an assignment statement is always taken to be the former, as long as either the `-xl` or `-xld` option is set.

Example: The following statement is ambiguous.

```
      PARAMETER S = .TRUE.
```

With `-xl`, the above statement is a PARAMETER statement about variable S.

```
      PARAMETER S = .TRUE.
```

It is *not* an assignment statement about the variable PARAMETERS.

```
      PARAMETERS = .TRUE.
```

4.53 PAUSE

The PAUSE statement suspends execution and waits for you to type go.

Syntax

PAUSE [*str*]	
str	String of not more than 5 digits or a character constant

Description

The PAUSE statement suspends program execution temporarily and waits for acknowledgment. On acknowledgment, execution continues.

The argument *string*, if present, is displayed on the screen. Then the following message is displayed on the screen:

```
     PAUSE. To resume execution, type: go
     Any other input will terminate the program.
```

After you type go, execution continues as if a CONTINUE statement is executed.

```
demo$ cat p.f
     PRINT *, "Start"
     PAUSE 1
     PRINT *, "Ok"
     END
demo$ f77 p.f
p.f:
 MAIN:
demo$ a.out
Start
PAUSE: 1
To resume execution, type: go
Any other input will terminate the program.
go
Execution resumed after PAUSE.
Ok
demo$ ■
```

If stdin is not a tty io device, PAUSE displays a message of the form:

```
PAUSE: To resume execution, type: kill -15 pid
```

where *pid* is the process ID.

Example: stdin not a tty I/O device.

```
demo$ a.out < mydatafile
PAUSE: To resume execution, type: kill -15 20537
demo$ n
```

For the above example, type the following at a shell prompt in some other window. (The window displaying the message cannot accept command input.)

```
demo$ kill -15 20537
```

4.54 POINTER

The POINTER ♦ statement establishes pairs of variables and pointers.

Syntax

POINTER (*p1*, *v1*) [, (*p2*, *v2*) …]	
v1, v2	Pointer-based variables
p1, p2	Corresponding pointers

Description

Each pointer contains the address of its paired variable.

A *pointer-based variable* is a variable paired with a pointer in a POINTER statement. A pointer-based variable is usually called just a *based variable.* The *pointer* is the integer variable that contains the address.

Usage

Normal use of pointer-based variables involves the following steps (the first two steps can be in either order).

1. Define the pairing of the pointer-based variable and the pointer in a `POINTER` statement.
2. Define the type of the pointer-based variable. The pointer itself is integer type, but in general, it is safer if you *not* list it in an `INTEGER` statement.
3. Set the pointer to the address of an area of memory that has the appropriate size and type. You do *not* normally do anything else with the pointer explicitly.
4. Reference the pointer-based variable. Just use the pointer-based variable in normal FORTRAN statements – the address of that variable will always be taken from its associated pointer.

Address and Memory

Note that no storage for the variable is allocated when a pointer-based variable is defined, so it is *your* responsibility to provide an address of a variable of the appropriate type and size, and assign the address to a pointer, usually with the normal assignment statement or data statement.

There are three procedures used to manage memory with pointers

- `LOC` — You can obtain the address from the intrinsic function `LOC()`.
- `MALLOC` — You can obtain both the area of memory and the address from the function `MALLOC()`.
- `FREE` — You can *deallocate* a region of memory previously allocated by `MALLOC()` by using the subroutine `FREE()`.

Subroutine FREE()

The subroutine `FREE()` deallocates a region of memory previously allocated by `MALLOC()`. The argument given to `FREE()` must be a pointer previously returned by `MALLOC()`, but not already given to `FREE()`. The memory is returned to the memory manager, making it unavailable to the programmer.

Function MALLOC( )

The function MALLOC( ) allocates an area of memory and returns the address of the start of that area. The argument to the function is an integer specifying the amount of memory to be allocated, in bytes. If successful, it returns a pointer to the first item of the region; otherwise, it returns an integer 0. The region of memory is not initialized in any way — assume it is garbage.

Restrictions

- The pointers are of type integer and are automatically typed that way by the compiler. You must *not* type them yourself.
- A pointer-based variable cannot itself be a pointer.
- The pointer-based variables can be of any type, including structures.
- No storage is allocated when such a pointer-based variable is defined, even if there is a size specification in the type statement.
- You can't use a pointer-based variable as a dummy argument or in COMMON, EQUIVALENCE, DATA, or NAMELIST statements.
- The dimension expressions for pointer-based variables must be constant expressions in main programs. In subroutines and functions, the same rules apply for pointer-based array variables as for dummy arguments — the expression can contain dummy arguments and variables in common. Any variables in the expressions must be defined with an integer value at the time the subroutine or function is called.
- This implementation of POINTER follows more along the line of Cray, and not Fortran 90, although it does not follow Cray exactly.

Optimization and Pointers

Pointers have the annoying side effect of reducing the assumptions that the global optimizer can make.

Compare:

- Without pointers, if you call a subroutine or function, the optimizer knows that the call will change only variables in common or those passed as arguments to that call.

- With pointers, this is no longer valid, since a routine can take the address of an argument and save it in a pointer in common for use in a subsequent call to itself or to another routine.

Therefore, the optimizer must assume that a variable passed as an argument in a subroutine or function call can be changed by any other call. Such an unrestricted use of pointers would degrade optimization for the vast majority of programs that *don't* use pointers.

Restrictions

If you use an optimization level greater than `-O2`, you must write your programs with the following restrictions on the use of pointers:

- Subroutines and functions are not permitted to save the address of any of their arguments between calls.
- A function can't return the address of any of its arguments, although it can return the value of a pointer argument.
- Only those variables whose addresses are explicitly taken with the `LOC()` or `MALLOC()` functions can be referenced through a pointer.

Example: One kind of code that could cause trouble if you optimize at a level greater than `-O2`.

```
      COMMON A, B, C
      POINTER ( P, V )
      P = LOC(A) + 4        ! ←possible problems if optimized
      …
```

The compiler will assume that a reference through `P` can change `A`, but not `B`; this assumption could produce incorrect code.

Examples

Example 1: A simple `POINTER` statement.

```
      POINTER ( P, V )
```

Here, `V` is a pointer-based variable, and `P` is its associated pointer.

Example 2: Using the LOC() function to get an address.

```
      * ptr1.f: Assign an address via LOC()
      POINTER ( P, V )
      CHARACTER A*12, V*12
      DATA A / 'ABCDEFGHIJKL' /
      P = LOC( A )
      PRINT *, V(5:5)
      END
```

In the above example, the CHARACTER statement allocates 12 bytes of storage for A, but *no* storage for V; it merely specifies the type of V because V is a pointer-based variable. Then we assign the address of A to P so now any use of V will refer to A by the pointer P. The program will print an E.

Example 3: Memory allocation for pointers, by MALLOC.

```
      POINTER ( P1, X ), ( P2, Y ), ( P3, Z )
      …
      P1 = MALLOC ( 36 )
      …
      CALL FREE ( P1 )
      …
```

In the above example, you get 36 bytes of memory from MALLOC() and then after some other instructions, probably using that chunk of memory, we tell FREE() to return those same 36 bytes to the memory manager.

Example 4: Get the area of memory and its address.

```
      POINTER ( P, V )
      CHARACTER V*12, Z*1
      P = MALLOC( 12 )
      …
      END
```

In the above example, you obtain 12 bytes of memory from the function MALLOC() and assign the address of that block of memory to the pointer P.

Example 5: Dynamic allocation of arrays. This has the form of a slightly more realistic example. The size might well be some large number, say, 10000. Once that's allocated, the subroutines do their stuff, not knowing (or caring) that the array was dynamically allocated.

```
      PROGRAM UsePointers
      REAL X
      POINTER ( P, X )
      …
      READ ( *,* ) Nsize ! Get the size.
      P = MALLOC( Nsize )! Allocate the memory.
      …
      CALL CALC ( X, Nsize )
      …
      END
      SUBROUTINE CALC ( A, N )
      REAL A(N)
      …           ! Use the array of whatever size.
      RETURN
      END
```

Example 6: One way to use pointers to make a linked list in `f77`.

Linked.f

```
      STRUCTURE /NodeType/
           INTEGER recnum
           CHARACTER*3 label
           INTEGER next
      END STRUCTURE
      RECORD /NodeType/ r, b
      POINTER (pr,r), (pb,b)
      pb = malloc(12)    ! Create the base record, b.
      pr = pb            ! Make pr point to b.
      NodeNum = 1
      DO WHILE (NodeNum .LE. 4)! Initialize/create records
          IF (NodeNum .NE. 1) pr = r.next
          CALL struct_creat(pr,NodeNum)
          NodeNum = NodeNum + 1
      END DO
      r.next = 0
      pr = pb             ! Show all records.
      DO WHILE (pr .NE. 0)
          PRINT *, r.recnum, " ", r.label
          pr = r.next
      END DO
      END
```

Linked.f (continued)

```
      SUBROUTINE struct_creat(pr,Num)
      STRUCTURE /NodeType/
           INTEGER recnum
           CHARACTER*3 label
           INTEGER next
      END STRUCTURE

      RECORD /NodeType/ r
      POINTER (pr,r), (pb,b)
      CHARACTER v*3(4)/'aaa', 'bbb', 'ccc', 'ddd'/

      r.recnum = Num  ! Initialize current record.
      r.label = v(Num)
      pb = malloc(12) ! Create next record.
      r.next = pb
      RETURN
      END
```

```
demo$ f77 -silent Linked.f
"Linked.f", line 6: Warning: local variable "b" never used
"Linked.f", line 31: Warning: local variable "b" never used
demo$ a.out
   1 aaa
   2 bbb
   3 ccc
   4 ddd
demo$
```

Remarks

- Do not optimize programs using pointers like this with `-O3` or `-O4`.
- The warnings can be ignored.
- This is not the normal usage of pointers described at the start of this section.

4.55 PRINT

The PRINT statement writes from a list to stdout.

Syntax

PRINT *f* [, *iolist*]	
PRINT *grname*	
f	Format identifier
iolist	List of variables, substrings, arrays, records, …
grname	Name of the namelist group

Description

Format Identifier

f is a format identifier and can be

- An asterisk (*), indicating list-directed I/O. See Section 5.6 for details.
- The label of a FORMAT statement that appears in the same program unit.
- An integer variable name that has been assigned the label of a FORMAT statement that appears in the same program unit.
- A character expression or integer array specifying the format string. The integer array is nonstandard. ♦

Output List

iolist can be empty or can contain output items and/or implied DO lists. The output items must be one of the following:

- Variables
- Substrings
- Arrays
- Array elements
- Records
- Record fields
- Any other expression

A simple unsubscripted array name specifies all of the elements of the array in memory storage order, with the leftmost subscript increasing more rapidly.

Namelist-directed PRINT

The second form of the `PRINT` statement is used to print the items of the specified namelist group. Here, *grname* is the name of a group previously defined by a `NAMELIST` statement.

Execution proceeds as follows:

1. The format, if specified, is established.
2. If the output list is not empty, data is transferred from the list to standard output. If a format is specified, data is edited accordingly.
3. In the second form of the `PRINT` statement, data is transferred from the items of the specified namelist group to standard output.

Restrictions

- Output from an Exception Handler is Unpredictable

  If you make your own exception handler, do not do any FORTRAN output from it. If you must do some, then call abort right after the output. This reduces the relative risk of a system freeze. FORTRAN I/O from an exception handler amounts to recursive I/O. See next paragraph.

- Recursive I/O Does not Work Reliably

  If you list a function in an I/O list, and if that function does I/O, then during runtime the execution may freeze, or some other unpredictable problem happens. This risk exists independent of using parallelization.

  Example: Recursive I/O fails intermittently.

```
      PRINT *, x, f(x)     ! Not allowed, f() does I/O.
      END
      FUNCTION F(X)
      PRINT *, X
      RETURN
      END
```

Examples

Example 1: Formatted scalars.

```
      CHARACTER TEXT*16
      PRINT 1, NODE, TEXT
 1    FORMAT ( I2, A16 )
```

Example 2: List-directed array.

```
      PRINT *, I, J, ( VECTOR(I), I = 1, 5 )
```

Example 3: Formatted array.

```
      INTEGER VECTOR(10)
      PRINT '( 12 I2 )', I, J, VECTOR
```

Example 4: Namelist.

```
      CHARACTER LABEL*16
      REAL QUANTITY
      INTEGER NODE
      NAMELIST /SUMMARY/ LABEL, QUANTITY, NODE
      PRINT SUMMARY
```

4.56 PROGRAM

The PROGRAM statement identifies the program unit as a main program.

Syntax

PROGRAM *pgm*	
pgm	Symbolic name of the main program

Description

For the loader, the main program is always named MAIN. The PROGRAM statement serves only the person who reads the program.

Restrictions

- The PROGRAM statement can appear only as the first statement of the main program.
- The name of the program *cannot* be:
  - The same as that of an external procedure or common block
  - MAIN (all caps), or you will get a runtime error.
- The name of the program can be the same as a local name in the main program.♦ The FORTRAN Standard doesn't allow this.

Example

Example: PROGRAM statement.

```
      PROGRAM US_ECONOMY
      NVARS = 2
      NEQS = 2
      …
```

4.57 READ

The READ statement reads data from a file or the keyboard to items in the list. If you use this for tapes we recommend the TOPEN( ) routines instead because they are more reliable.

Syntax

```
READ( [ UNIT=] u [, [ FMT=] f ] [, IOSTAT= ios ] [, REC= rn ]
        [, END= s ]  [, ERR= s ] ) iolist
```

```
READ f  [, iolist ]
```

```
READ([UNIT=] u, [NML=] grname [,IOSTAT=ios ] [,END=s ] [,ERR=s ] )
```

```
 READ [ NML= ] grname
```

An alternate to the "UNIT=*u*, REC=*rn*" form is as follows. ♦

```
READ( u ' rn … ) iolist
```

u	Unit identifier of the unit connected to the file
f	Format identifier
ios	I/O status specifier
rn	Record number to be read
s	Statement label for end of file processing
iolist	List of variables
grname	Name of a namelist group

The options can be specified in any order.

Description

Unit Identifier

u is either an external unit identifier or an internal file identifier.

An *external unit identifier* must be one of these:
- Nonnegative integer expression
- Asterisk, identifying `stdin`, normally connected to the keyboard

If the optional characters "`UNIT=`" are omitted from the unit specifier, then *u* must be the first item in the list of specifiers.

Format Identifier

f is a format identifier and can be:

- Asterisk (*), indicating list-directed I/O. See Section 5.6 for details.
- Label of a `FORMAT` statement that appears in the same program unit
- Integer variable name that has been assigned the label of a `FORMAT` statement that appears in the same program unit
- Character expression or integer array specifying the format string. This is called a runtime format or a variable format. The integer array is nonstandard. ♦

If the optional characters "FMT=" are omitted from the format specifier, then *f* must appear as the second argument for a formatted read, otherwise it must not appear at all.

Unformatted data transfer from internal files and terminal files is not allowed, hence, *f* must be present for such files.

List-directed data transfer from direct-access and internal files is allowed, hence, *f* can be an asterisk for such files. ♦

If a file is connected for formatted I/O, unformatted data transfer is not allowed, and vice versa.

I/O Status Specifier

ios must be an integer variable or an integer array element.

Record Number

rn must be a positive integer expression and can be used for direct-access files only. *rn* can be specified for internal files. ♦

End-of-File Specifier

s must be the label of an executable statement in the same program unit in which the READ statement occurs.

The "END=*s*" and "REC=*rn*" specifiers can be present in the same READ statement. ♦

Error Specifier

s must be the label of an executable statement in the same program unit in which the READ statement occurs.

Input List

iolist can be empty or can contain input items and/or implied DO lists. The input items can be any of the following.

- Variables
- Substrings

- Arrays
- Array elements
- Records
- Record fields

A simple unsubscripted array name specifies all of the elements of the array in memory storage order, with the leftmost subscript increasing more rapidly.

Namelist-directed READ

The third and fourth forms of the READ statement are used to read the items of the specified namelist group, and *grname* is the name of the group of variables previously defined in a NAMELIST statement.

Execution Proceeds as Follows:

1. The file associated with the specified unit is determined. The format, if specified, is established. The file is positioned appropriately prior to the data transfer.
2. If the input list is not empty, data is transferred from the file to the corresponding items in the list. The items are processed in order as long as the input list is not exhausted. The next specified item is determined and the value read is transmitted to it. Data editing in formatted READ is done according to the specified format.
3. In the third and fourth forms of namelist-directed READ, the items of the specified namelist group are processed according to the rules of namelist-directed input.
4. The file is repositioned appropriately after data transfer.
5. If *ios* is specified and no error occurred, it is set to zero. It is set to a positive value, if an error or end of file was encountered.
6. If *s* is specified and end of file was encountered, control is transferred to *s*.
7. If *s* is specified and an error occurs, control is transferred to *s*.

Execution for Keyboard Read

There are two forms of READ.

```
READ f [, iolist ]
READ [ NML= ] grname
```

They above two forms operate the same way as the others except that reading from the *keyboard* is implied. Execution has these differences.

1. When the input list is exhausted, the cursor is moved to the start of the line following the input. For an empty input list, the cursor is moved to the start of the line following the input.
2. If an end-of-line, CR, or NL is reached before the input list is satisfied, input continues from the next line.
3. If an end-of-file (Control D) is received before the input list is satisfied, input stops, and unsatisfied items of the input list remain unchanged.

If *u* specifies an external unit that is not connected to a file, an implicit OPEN operation is performed which is equivalent to opening the file with the options in the following example.

```
    OPEN( u, FILE='FORT.u', STATUS='OLD',
&             ACCESS='SEQUENTIAL', FORM=fmt )
```

The value of *fmt* is 'FORMATTED' or 'UNFORMATTED' accordingly, as the read is formatted or unformatted.

An simple unsubscripted array name specifies all of the elements of the array in memory storage order, with the leftmost subscript increasing more rapidly.

An attempt to read the record of a direct-access file that has not been written yet causes all items in the input list to become undefined.

The record number count starts from one.

Namelist-directed input is permitted on sequential access files only.

Examples

Example 1: Formatted read, trap I/O errors, EoF, and I/O status.

```
      READ( 1, 2, ERR=8, END=9, IOSTAT=N ) X, Y
      …
8     WRITE( *, * ) 'I/O error # ', N, ', on 1'
      STOP
9     WRITE( *, * ) 'EoF on 1'
      RETURN
      END
```

Example 2: Direct, unformatted read, trap I/O errors, and I/O status.

```
      …
      READ( 1, REC=3, IOSTAT=N, ERR=8 ) V
      …
4     CONTINUE
      RETURN
8     WRITE( *, * ) 'I/O error # ', N, ', on 1'
      END
```

Example 3: List-directed read from keyboard.

```
      READ( *, * ) A, V
or
      READ *, A, V
```

Example 4: Formatted read from an internal file.

```
      CHARACTER CA*16 / 'abcdefghijklmnop' /, L*8, R*8
      READ( CA, 1 ) L, R
1     FORMAT( 2 A8 )
```

Example 5: Read an entire array.

```
      DIMENSION V(5)
      READ( 3, '(5F4.1)') V
```

Example 6: Namelist-directed read.

```
      CHARACTER SAMPLE*16
      LOGICAL NEW*4
      REAL DELTA*4
      NAMELIST /G/ SAMPLE, NEW, DELTA
      …
      READ( 1, G )
or
      READ( UNIT=1, NML=G )
or
      READ( 1, NML=G )
```

4.58 REAL

The REAL statement specifies the type of a symbolic constant, variable, array, function, or dummy function to be real, and optionally specifies array dimensions and size, and initializes with values.

Syntax

REAL [*len[,]] v[* len [/c/]] [, v [* len [/c/]] …	
v	Name of a variable, symbolic constant, array, array declarator, function, or dummy function
len	Either 4, 8, or 16 (*SPARC only*), the length in bytes of the symbolic constant, variable, array element, or function
c	List of constants for the immediately preceding name

Description

REAL

For a declaration such as REAL W, the variable W is usually a REAL*4 element in memory, interpreted as a real number (more details in ”Default Size,” below.)

REAL*4 ♦

For a declaration such as `REAL*4 W`, the variable `W` is always a `REAL*4` element in memory, interpreted as a single-width real number.

REAL*8 ♦

For a declaration such as `REAL*8 W`, the variable `W` is always a `REAL*8` element in memory, interpreted as a double-width real number.

REAL*16 ♦

(SPARC only) For a declaration such as `REAL*16 W`, the variable `W` is always an element of type `REAL*16` in memory, interpreted as a quadruple-width real.

Default Size

If you specify the size as 4, 8, or 16, you get what you specify; if you *do not* specify the size, you get the default size.

The default size for a declaration such as `REAL X`, depends on the `–r8` option.

- If `–r8` is on the `f77` command line, then for declarations such as `REAL X`, the compiler allocates 8 bytes, and does 8-byte arithmetic. If `–r8` is *not* on the `f77` command line, then the compiler allocates 4 bytes.
- If you put both `–i2` and `–r8` on the `f77` command line, the results are unpredictable.

Examples

Example 1: Simple real scalars. Each of these statements is generally equivalent to the others, but the first is different if you compile with the `-r8` option.

```
      REAL U, V
      REAL*4 U, V
      REAL U*4, V*4
```

Above, don't use all three statements in the same program unit.

Example 2: Initialize scalars *(The* `REAL*16` *is for SPARC only).*

```
      REAL U/ 1.0 /, V/ 4.3 /, D*8/ 1.0 /, Q*16/ 4.5 /
```

Example 3: Specify dimensions for some real arrays.

```
      REAL A(10,100), V(10)
      REAL X*4(10), Y(10)*4
```

Example 4: Initialize some arrays.

```
      REAL A(10,100) / 1000 * 0.0 /, B(2,2) /  1.0, 2.0, 3.0, 4.0 /
```

Example 5: Double and quadruple precision *(The* REAL*16 *is for SPARC only).*

```
      REAL*8  R
      REAL*16 Q
      DOUBLE PRECISION D
```

Above, D and R are both double precision. Q is quadruple precision.

4.59 RECORD

The RECORD ♦ statement defines variables to have a specified structure, or defines arrays to be arrays of variables with such structures.

Syntax

RECORD */struct-name/ record-list* [,*/struct-name/ record-list*]…	
struct-name	Name of a previously declared structure
record-list	List of variables, arrays, or arrays with dimensioning/index ranges

Description

A structure is a template for a record. The name of the structure is included in the STRUCTURE statement, and once a structure is thus defined and named, it can be used in a RECORD statement. The *record* is a generalization of the variable or array: where a variable or array has a *type*, the record has a *structure*. Where all the elements of an array must be of the same type, the fields of a record can be of different types.

The RECORD line is part of an inherently multiline group of statements, and neither the RECORD line nor the END RECORD line has any indication of continuation. Do not put a nonblank in column six, nor an & in column one.

Restrictions

- Each record is allocated separately in memory.
- Initially, records have undefined values.
- Records, record fields, record arrays, and record-array elements are allowed as arguments and dummy arguments. When you pass records as arguments, their fields must match in type, order, and dimension. The record declarations in the calling and called procedures must match.
- Within a union declaration, the order of the map fields is not relevant.
- Records and record fields are allowed in COMMON and DIMENSION.
- Records and record fields are not allowed in DATA, EQUIVALENCE, NAMELIST, PARAMETER, AUTOMATIC, STATIC, or SAVE statements.

Example

Example 1: Declaring some items to be records of a specified structure.

```
      STRUCTURE /PRODUCT/
           INTEGER*4 ID
           CHARACTER*16 NAME
           CHARACTER*8 MODEL
           REAL*4 COST
           REAL*4 PRICE
      END STRUCTURE
      RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
      …
```

Each of the three variables CURRENT, PRIOR, and NEXT is a record which has the PRODUCT structure, and LINE is an array of 10 such records.

Example 2: Define some fields of records, then use them.

```
      STRUCTURE /PRODUCT/
           INTEGER*4    ID
           CHARACTER*16 NAME
           CHARACTER*8  MODEL
           REAL*4       COST
           REAL*4       PRICE
      END STRUCTURE
      RECORD /PRODUCT/  CURRENT, PRIOR, NEXT, LINE(10)
      CURRENT.ID = 82
      PRIOR.NAME = "CacheBoard"
      NEXT.PRICE = 1000.00
      LINE(2).MODEL = "96K"
      PRINT 1, CURRENT.ID, PRIOR.NAME, NEXT.PRICE, LINE(2).MODEL
1     FORMAT(I5/A16/F8.2/A8)
      END
```

The above program has the following output:

```
   82
CacheBoard
 1000.00
96K
```

4.60 RETURN

A RETURN statement returns control to the calling program unit.

Syntax

RETURN [*e*]	
e	Expression of type INTEGER or REAL

Description

Execution of a RETURN statement terminates the reference of a function or subroutine.

Execution of an END statement in a function or a subroutine is equivalent to the execution of a RETURN statement. ♦

The expression *e* is evaluated and converted to integer, if required. *e* defines the ordinal number of the *alternate return* label to be used. Alternate return labels are specified as asterisks (or ampersands ♦) in the SUBROUTINE statement.

If *e* is not specified or the value of *e* is less than one or greater than the number of asterisks or ampersands in the SUBROUTINE statement containing the RETURN statement, control is returned normally to the statement following the CALL statement that invoked the subroutine.

If the value of *e* is between one and the number of asterisks (or ampersands) in the SUBROUTINE statement, control is returned to the statement identified by the e^{th} alternate.A RETURN statement can appear only in a function subprogram or subroutine.

Examples

Example 1: Standard return.

```
      CHARACTER*25 TEXT
      TEXT = "Some kind of minor catastrophe"
      …
      CALL OOPS ( TEXT )
      STOP
      END
      SUBROUTINE OOPS ( S )
      CHARACTER S* 32
      WRITE (*,*) S
      RETURN
      END
```

Example 2: Alternate return.

```
      CALL RANK ( N, *8, *9 )
      WRITE (*,*) 'OK - Normal Return'
      STOP
 8    WRITE (*,*) 'Minor - 1st alternate return'
      STOP
 9    WRITE (*,*) 'Major - 2nd alternate return'
      END
      SUBROUTINE RANK (N, *,*)
      IF ( N .EQ. 0 ) RETURN
      IF ( N .EQ. 1 ) RETURN 1
      RETURN 2
      END
```

4.61 REWIND

REWIND positions the file associated with the specified unit to its initial point.

If you use this for tapes we recommend the TOPEN() routines instead because they are more reliable.

Syntax

REWIND *u*	
REWIND ([UNIT=] *u* [, IOSTAT=*ios*] [, ERR= *s*])	
u	Unit identifier of an external unit connected to the file *u* must be connected for *sequential* access, or *append access.*
ios	I/O specifier, an integer variable or an integer array element
s	Error specifier - *s* must be the label of an executable statement in the same program in which this REWIND statement occurs. The program control is transferred to this label in case of an error during the execution of the REWIND statement.

Description

The options can be specified in any order.
Rewinding a unit not associated with any file has no effect.
REWIND in a terminal file has no effect.

Execution of a REWIND statement on a direct-access file is not defined in the FORTRAN Standard and is unpredictable. We do not recommend using a REWIND statement on a direct-access file.

Examples

Example 1: Simple form of unit specifier.

```
      ENDFILE 3
      REWIND 3
      READ (3,'(I2)') I
      REWIND 3
      READ (3,'(I2)')I
```

Example 2: REWIND with the UNIT=*u* form of unit specifier and error trap.

```
      INTEGER  CODE
      …
      REWIND (UNIT = 3)
      REWIND (UNIT = 3, IOSTAT = CODE, ERR = 100)
      …
100   WRITE (*,*) 'error in rewinding'
      STOP
```

4.62 SAVE

The SAVE statement prevents items in a subprogram from becoming undefined after the RETURN or END statements are executed.

Syntax

SAVE [*v* [, *v*] …]	
v	Name of an array, variable, or common block (enclosed in slashes), occurring in a subprogram

Description

All variables to be saved are placed in an internal static area. All common blocks are saved by allocating a static area. Therefore, common block names specified in SAVE statements are just ignored. A SAVE statement is optional in the main program and has no effect. A SAVE with no list saves everything savable.

SAVE/STATIC

Local variables and arrays are static by default, so in general, this eliminates the need for SAVE. You can still use SAVE to ensure portability. Also, SAVE is safer if you leave a subprogram by some way other than a RETURN.

Restrictions

The following must not appear in a SAVE statement:

- Variables or arrays in a common block
- Dummy argument names
- Record names
- Procedure names
- Automatic variables or arrays

Example:

Example: SAVE.

```
      SUBROUTINE FFT
      DIMENSION A(1000,1000), V(1000)
      SAVE A
      …
      RETURN
      END
```

4.63 Statement Function

A *statement function* statement is a function-like declaration, made in a single statement.

Syntax

fun ([d [, d] …]) = e	
fun	Name of statement function being defined
d	Statement function dummy argument
e	Expression. *e* can be any of the types arithmetic, logical, or character.

Description

If a statement function is referenced, the defined calculations are inserted.

Example: The following statement is a statement function.

```
      ROOT( A, B, C ) = (-B + SQRT(B**2-4.0*A*C))/(2.0*A)
```

The statement function argument list indicates the order, number, and type of arguments for the statement function.

Restrictions

- A statement function must appear only after the specification statements and before the first executable statement of the program unit in which it is referenced.
- A statement function is not executed at the point where it is specified. It is executed, as any other, by the execution of a function reference in an expression.
- The type conformance between *fun* and *e* are the same as those for the assignment statement. Note that the type of *fun* and *e* can be different, in which case *e* is converted to the type of *fun*.
- The actual arguments must agree in order, number, and type with corresponding dummy arguments.
- The same argument cannot be specified more than once in the argument list.
- The statement function must be referenced only in the program unit that contains it.
- The name of a statement function cannot be an actual argument. Nor can it appear in an EXTERNAL statement.
- The type of the argument is determined as if the statement function were a whole program unit in itself.
- Even if the name of a statement function argument is the same as that of another local variable, the reference is considered as a dummy argument of the statement function and not the local variable of the same name.
- The length specification of a character statement function or its dummy argument of type CHARACTER must be an integer constant expression.
- A statement function cannot be invoked recursively.

Referencing a Statement Function

A statement function is referenced by using its name, along with its arguments, as an operand in an expression.

Execution proceeds as follows:

1. If they are expressions, actual arguments are evaluated.
2. Actual arguments are associated with corresponding dummy arguments.

3. The expression *e*, the body of a statement function, is evaluated.
4. If the type of the above result is different from the type of the function name, then the result is converted.
5. Return the value.

The resulting value is thus available to the expression that referenced the function.

Examples

Example 1: Arithmetic statement function.

```
      PARAMETER ( PI=3.14159 )
      REAL RADIUS, VOLUME
      SPHERE ( R ) = 4.0 * PI * (R**3) / 3.0
      READ *, RADIUS
      VOLUME = SPHERE( RADIUS )
      …
```

Example 2: Logical statement function.

```
      LOGICAL OKFILE
      INTEGER STATUS
      OKFILE ( I ) = I .LT. 1
      READ( *, *, IOSTAT=STATUS ) X, Y
      IF ( OK FILE(STATUS) ) CALL CALC ( X, Y, A )
      …
```

Example 3: Character statement function.

```
      CHARACTER FIRST*1, STR*16
      FIRST(S) = S(1:1)
      READ( *, * ) STR
      IF ( FIRST(STR) .LT. " " ) CALL CONTROL ( S, A )
      …
```

4.64 STATIC

The STATIC ♦ statement insures that the specified items are stored in static memory.

Syntax

STATIC *list*	
list	List of variables and arrays

Description

To deal with the problem of local variables becoming undefined between invocations, f77 classifies every variable as either static or automatic, with all local variables being static by default.

For static variables, there is exactly one copy of each datum, and its value is retained between calls. You can also explicitly define variables as static or automatic in a STATIC or AUTOMATIC statement, or in any type statement or IMPLICIT statement.

SAVE/STATIC — Local variables and arrays are static by default, so in general, this eliminates the need for SAVE. You can still use SAVE to insure portability. Also, SAVE is safer if you leave a subprogram by some way other than a RETURN.

Example

```
      STATIC A, B, C
      REAL P, D, Q
      STATIC P, D, Q
      IMPLICIT STATIC (X-Z)
```

Remarks

- Arguments and function values are automatic.
- A STATIC statement and a *type* statement cannot be combined to make a STATIC *type* statement. For example, the statement

```
      STATIC REAL X       !    Not what you might expect
```

  does *not* declare the variable X to be both STATIC and REAL; It declares the variable REALX to be STATIC.

4.65 STOP

The STOP statement terminates execution of the program.

Syntax

STOP [[*str*]	
str	String of no more that 5 digits or a character constant

Description

The argument *str* is displayed when the program stops.

If *str* is not specified, no message is displayed.

Examples

Example 1: Integer.

```
      stop 9
```

The above statement displays:

```
      STOP: 9
```

Example 2: Character.

```
stop 'oyvay'
```

The above statement displays:

```
STOP: oyvay
```

Example 3: Nothing after the stop.

```
stop
```

The above statement displays nothing.

4.66 STRUCTURE

The STRUCTURE ♦ statement organizes data into *structures*.

Syntax

```
STRUCTURE  [/structure-name/]  [ field-list ]
                        field-declaration
                      [ field-declaration]
                      …
                      [ field-declaration]

END STRUCTURE
```

structure-name	Name of the structure
field-list	List of fields of the specified structure
field-declaration	Field of the record

Field declaration

Each field declaration can be one of the following:

- A substructure (either another structure declaration, or a record that has been previously defined)
- A union declaration
- A type declaration (can include initial values)

Description

It defines a *form* for a *record* by specifying the name, type, size, and order of the *fields* that constitute the record. Optionally, it can specify the initial values. A structure is a template for a record. The name of the structure is included in the STRUCTURE statement, and once a structure is thus defined and named, it can be used in a RECORD statement. The *record* is a generalization of the variable or array — where a variable or array has a *type*, the record has a *structure*. Where all the elements of an array must be of the same type, the fields of a record can be of different types.

Restrictions

- The name is enclosed in slashes and is optional in nested structures only.
- If slashes are present, a name must be present.
- You can specify the *field-list* within nested structures only.
- There must be at least one *field-declaration*.
- Each *structure-name* must be unique among structures, although you can use structure names for fields in other structures or as variable names.
- The only statements allowed between the STRUCTURE statement and the END STRUCTURE statement are *field-declaration* statements and PARAMETER statements. A PARAMETER statement inside a structure declaration block is equivalent to one outside.

Restrictions for Fields

Fields that are type declarations use the identical syntax of normal FORTRAN type statements, and all `f77` types are allowed, subject to the following rules and restrictions.

- Any dimensioning needed must be in the type statement. The `DIMENSION` statement has no effect on field names.
- You can specify the pseudonyme `%FILL` for a field name. The `%FILL` is provided for compatibility with other versions of FORTRAN. It is not needed in `f77` because the alignment problems are taken care of for you. It might be considered a useful feature to anyone who wants to make one or more fields not referenceable in some particular subroutine. The only thing that `%FILL` does is provide a field of the specified size and type, and preclude referencing it.
- You must explicitly type all field names. The `IMPLICIT` statement does not apply to statements in a `STRUCTURE` declaration, nor do the implicit `I,J,K,L,M,N` rules apply.
- You can't use arrays with adjustable or assumed size in field declarations, nor can you include passed-length `CHARACTER` declarations.

Field Offsets

- In a structure declaration, the offset of field *n* is the offset of the preceding field, plus the length of the preceding field, possibly corrected for any adjustments made to maintain alignment.
- You can initialize a field that is a variable, array, substring, substructure, or union.

Examples

Example 1: A structure of five fields.

```
      STRUCTURE /PRODUCT/
           INTEGER*4 ID / 99 /
           CHARACTER*16 NAME
           CHARACTER*8 MODEL/ 'Z' /
           REAL*4 COST
           REAL*4 PRICE
      END STRUCTURE
      RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
```

In the above example, a *structure* named PRODUCT is defined to consist of the fields ID, NAME, MODEL, COST, and PRICE. Each of the three variables CURRENT, PRIOR, and NEXT is a record which has the PRODUCT structure, and LINE is an array of 10 such records. Every such record has its ID initially set to 99, and its MODEL initially set to Z.

Example 2: A structure of two fields.

```
      STRUCTURE /VARLENSTR/
           INTEGER*4 NBYTES
           CHARACTER A*25
      END STRUCTURE
      RECORD /VARLENSTR/ VLS
      VLS.NBYTES = 0
```

The above structure matches the one used by the pc Pascal compiler for varying length strings. The 25 is arbitrary.

4.67 SUBROUTINE

The SUBROUTINE statement identifies a named program unit as a subroutine, and specifies arguments for it.

Syntax

SUBROUTINE *sub* [([*fd* [, *fd*] …])]	
sub	Name of subroutine subprogram
d	Variable name, array name, record name, or dummy procedure name, or an asterisk, or an ampersand

Description

A subroutine subprogram must have a SUBROUTINE statement as the first statement. A subroutine can have any other statements except a BLOCK DATA, FUNCTION, PROGRAM, or another SUBROUTINE statement.

sub is the name of a subroutine and is a global name and must not be the same as any other global name such as a common block name or a function name. Nor can it be the same as any local name in the same subroutine.

d is the dummy argument, and multiple dummy arguments are separated by commas. *d* can be one of the following:

- Variable name
- Array name
- Dummy procedure name
- Record name
- Asterisk (*) or an ampersand (&) ♦

The dummy arguments are local to the subroutine and must *not* appear in any of the following statements, except as a common block name:

- EQUIVALENCE
- PARAMETER
- SAVE
- STATIC
- AUTOMATIC
- INTRINSIC
- DATA
- COMMON

The actual arguments in the CALL statement that references a subroutine must agree with the corresponding formal arguments in the SUBROUTINE statement, in order, number, and type. An asterisk (or an ampersand) in the formal argument list denotes an alternate return label. A RETURN statement in this procedure can specify the ordinal number of the alternate return to be taken.

Examples

Example 1: A variable and array as parameters.

```
      SUBROUTINE SHR ( A, B )
      CHARACTER A*8
      REAL B(10,10)
      …
      RETURN
      END
```

In this example, the RETURN 1 statement refers to the first alternate return label (first *) and the RETURN 2 statement refers to the second alternate return label (second *) specified in the SUBROUTINE statement.

Example 2: Standard alternate returns.

```
      PROGRAM TESTALT
      CALL RANK ( N, *8, *9 )
      WRITE (*,*) 'OK - Normal Return [n=0]'
      STOP
 8    WRITE (*,*) 'Minor - 1st alternate return [n=1]'
      STOP
 9    WRITE (*,*) 'Major - 2nd alternate return [n=2]'
      END
      SUBROUTINE RANK ( N, *, * )
      IF ( N .EQ. 0 ) RETURN
      IF ( N .EQ. 1 ) RETURN 1
      RETURN 2
      END
```

Example 3: Nonstandard alternate returns. ♦

```
      CALL SUB(…, &label, …)
```

is treated as

```
      CALL SUB(…, *label, …)
```

4.68 TYPE

The TYPE ♦ statement writes to stdout.

Syntax

```
TYPE  f [, iolist ]
or
TYPE  grname
```

f	Format identifier
iolist	List of output variables
grname	Name of the namelist group

Description

The TYPE statement is equivalent to "PRINT *f* [, *iolist*]" or "PRINT *grname*" or "WRITE(*, *f*) [*iolist*]" or "WRITE(*, *grname*)" and is provided for compatibility with older versions of FORTRAN.

Examples

Example 1: Formatted output.

```
      INTEGER V(5)
      TYPE 1, V
 1    FORMAT( 5 I3 )
```

Example 2: Namelist output.

```
      CHARACTER S*16
      INTEGER N
      NAMELIST /G/ N, S
      …
      TYPE G
```

4.69 The Type Statement

The *type* statement specifies the data type of items in the list, and optionally specifies array dimensions and initializes with values.

Syntax

type v [/ clist /] [,v [/ clist /]…	
v	Variable name, array name, array declarator, symbolic name of a constant, statement function or function subprogram name
clist	List of constants (There are more details about *clist* in the section on the DATA statement.)

type can be preceded by either AUTOMATIC or STATIC.

type can be one of the following type specifiers.

```
BYTE u                          INTEGER
CHARACTER                       INTEGER*2 ◆
CHARACTER*n                     INTEGER*4 ◆
CHARACTER*(*)                   LOGICAL
COMPLEX                         LOGICAL*1 ◆
COMPLEX*8 ◆                     LOGICAL*2 ◆
COMPLEX*16 ◆                    LOGICAL*4 ◆
COMPLEX*32 (SPARC only) ◆       REAL
DOUBLE COMPLEX ◆                REAL*4 ◆
DOUBLE PRECISION                REAL*8 ◆
                                REAL*16 (SPARC only) ◆
```

The *n*, as in CHARACTER*n, must be greater than 0.

Description

Different Usages of type Statement

A *type* statement can be used to conform or to override the type established by default or by the IMPLICIT statement.

A *type* statement can specify dimension information for an array.

A *type* statement can be used to confirm the *type* of an intrinsic function.

A *type* statement can be used to override the length by one of the acceptable lengths for that data type.

Initializing in Type Declarations

A *type* statement can assign initial values to variables, arrays, or record fields by specifying a list of constants (*clist*) as in a DATA statement. ♦

The general form of a *type* statement is:

```
type VariableName / constant / …
```

or

```
type ArrayName / constant, … /
```

or

```
type ArrayName / r*constant /
```

where *r* is a repeat factor.

Example: Various *type* statements.

```
      CHARACTER LABEL*12 / 'Standard' /
      COMPLEX STRESSPT / ( 0.0, 1.0 ) /
      INTEGER COUNT / 99 /, Z / 1 /
      REAL PRICE / 0.0 /, COST / 0.0 /
      REAL LIST(8) / 0.0, 6*1.0, 0.0 /
```

Restrictions for Initializing a Data Type

- For a simple variable, there must be exactly one constant.
- If any element of an array is initialized, all must be.
- You can use an integer as a *repeat factor*, followed by an asterisk (*), followed by a constant. (In the example above, six values of 1.0 are stored into array elements 2, 3, 4, 5, 6, and 7 of `LIST`.)
- If a variable or array is declared `AUTOMATIC`, then it cannot be initialized.
- A pointer-based variable or array cannot be initialized.

  Example:

```
      INTEGER Z / 4 /
      POINTER ( x, Z )
```

  You get a compiler warning message, and `Z` does *not* get initialized.
- If a variable or array is not initialized, its value(s) are undefined.
- If such initialization statements involve variables in `COMMON`, and the `–ansi` compiler flag is set, then a warning is issued.

Restrictions

- A symbolic name can appear only once in *type* statements in a program unit.
- A *type* statement must precede all executable statements.

Example

Example: The *type* statement.

```
      INTEGER*2 I, J/0/
      REAL*4 PI/3.141592654/,ARRAY(10)/5*0.0,5*1.0/
      CHARACTER*10 NAME
      CHARACTER*10 TITLE/'Heading'/
```

In the above example `J` is initialized to `0; PI` is initialized to `3.141592654`; the first five elements of `ARRAY` are initialized to `0.0`; the second five elements of `ARRAY` are initialized to `1.0,` and `TITLE` is initialized to `'Heading'`.

4.70 UNION and MAP

The `UNION` ♦ statement defines groups of fields that share memory at runtime.

Syntax

The syntax of a *union* declaration is as follows.

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

The syntax of a *map* declaration is as follows.

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

Description

A MAP statement defines alternate groups of fields in a union. During execution, one map at a time is associated with a shared storage location. When you reference a field in a map, the fields in any previous map become undefined and are succeeded by the fields in the map of the newly referenced field.

- A UNION declaration can appear only within a STRUCTURE declaration.
- The amount of memory used by a union is that of its biggest map.
- Within a UNION declaration, the order of the MAP statements is not relevant.

The UNION line is part of an inherently multiline group of statements, and neither the UNION line nor the END UNION line has any special indication of continuation. You do not put a nonblank in column six, nor an & in column one.

Fields in a Map

Each *field-declaration* in a *map* declaration can be one of the following.

- S*tructure* declaration
- R*ecord*
- U*nion* declaration
- D*eclaration* of a typed data field

Example

Declare the structure /STUDENT/ to contain either NAME, CLASS, and MAJOR — or NAME, CLASS, CREDITS, and GRAD_DATE.

```
      STRUCTURE /STUDENT/
           CHARACTER*32  NAME
           INTEGER*2  CLASS
           UNION
                MAP
                        CHARACTER*16 MAJOR
                END MAP
                MAP
                        INTEGER*2  CREDITS
                        CHARACTER*8  GRAD_DATE
                END MAP
           END UNION
      END STRUCTURE
      RECORD /STUDENT/ PERSON
```

In the above example, the variable PERSON has the structure /STUDENT/, so:

- PERSON.MAJOR references a field from the first map, and PERSON.CREDITS references a field from the second map.
- If the variables of the second map field are initialized and then the program references the variable PERSON.MAJOR, the first map becomes active and the variables of the second map become undefined.

4.71 VIRTUAL

The VIRTUAL ♦ statement is treated the same as the DIMENSION statement.

Syntax

VIRTUAL *a* (*d*) [, *a* (*d*)] …	
a	Name of an array
a(d)	Specifies the dimension of the array. It is a list of 1 to 7 declarators separated by commas

Description

The VIRTUAL statement has the same form and effect as the DIMENSION statement. It is included for compatibility with older versions of FORTRAN.

Example

```
      VIRTUAL M(4,4), V(1000)
      …
      END
```

4.72 VOLATILE

The VOLATILE ♦ statement prevents optimization on the specified items.

Syntax

VOLATILE *nlist*	
nlist	List of variables, arrays, or common blocks

Description

The VOLATILE statement prevents optimization on the items in the list. Programs relying on it are usually nonportable.

Example

Example: VOLATILE. ♦

```
      PROGRAM FFT
      INTEGER NODE*2, NSTEPS*2
      REAL DELTA, MAT(10,10), V(1000), X, Z
      COMMON /INI/ NODE, DELTA, V
      …
      VOLATILE V, Z, MAT, /INI/
      …
      EQUIVALENCE ( X, V )
      …
```

In the above example, the array V, the variable Z, and the common block /INI/ are explicitly specified as VOLATILE. The variable X is VOLATILE through an equivalence.

4.73 WRITE

The WRITE statement writes data from the list to a file.

Syntax

WRITE([UNIT=] *u* [, [FMT=] *f*] [, IOSTAT=*ios*] [, REC=*rn*] [, ERR=*s*]) *iolist*	
WRITE([UNIT=] *u*, [NML=] *grname* [, IOSTAT= *ios*] [, ERR= *s*])	
u	Unit identifier of the unit connected to the file
f	Format identifier
ios	I/O status specifier
rn	Record number
s	Error specifier (statement label)
iolist	List of variables
grname	Name of the namelist group

The options can be specified in any order.

An alternate for "REC=*rn*" form is allowed, as follows: ♦

```
WRITE( u ' rn … ) iolist ♦
```

Example 3 shows this also.

Description

For *tapes* we recommend the TOPEN() routines because they are more reliable.

Unit Identifier

u is either an external unit identifier or an internal file identifier.

An *external unit identifier* must be either:

- A nonnegative integer expression, or
- An asterisk, identifying stdout, which is normally connected to the console.

If the optional characters "UNIT=" are omitted from the unit specifier, then *u* must be the first item in the list of specifiers.

Format Identifier

f is a format identifier and can be:

- An asterisk (*), indicating list-directed I/O. See Section 5.6 for details.
- The label of a FORMAT statement that appears in the same program unit
- An integer variable name that has been assigned the label of a FORMAT statement that appears in the same program unit
- A character expression or integer array specifying the format string. This is called a runtime format or a variable format. The integer array is nonstandard. ♦

If the optional characters "FMT=" are omitted from the format specifier, then *f* must appear as the second argument for a formatted write, otherwise it must not appear at all.

f must not be an asterisk for direct access.

f can be an asterisk for internal files. ♦

If a file is connected for formatted I/O, unformatted data transfer is prohibited and vice versa.

I/O Status Specifier

ios must be an integer variable, integer array element, or integer record field.

Record Number

rn must be a positive integer expression. This argument can appear only for direct-access files. *rn* can be specified for internal files. ♦

Error Specifier

s must be the label of an executable statement in the same program unit in which this WRITE statement occurs.

Output List

iolist can be empty or can contain output items and/or implied DO lists. The output items must be one of the following.

- Variables
- Substrings
- Arrays
- Array elements
- Records
- Record fields
- Any other expression

A simple unsubscripted array name specifies all of the elements of the array in memory storage order, with the leftmost subscript increasing more rapidly.

If the output item is a character expression employing the concatenation operator, the length specifiers of its operands can be an asterisk (*). This is nonstandard. ◆

If a function appears in the output list, that function must not cause an input/output statement to be executed.

Namelist Write

The second form of `WRITE` is used to output the items of the specified namelist group. Here *grname* is the name of the list previously defined in a `NAMELIST` statement.

Execution proceeds as follows.

1. The file associated with the specified unit is determined. The format, if specified, is established. The file is positioned appropriately prior to data transfer.
2. If the output list is not empty, data is transferred from the list to the file. Data is edited according to the format, if specified.
3. In the second form of namelist-directed `WRITE`, the data is transferred from the items of the specified namelist group according to the rules of namelist-directed output.
4. The file is repositioned appropriately after the data transfer.
5. If *`ios`* is specified, and no error occurs, it is set to zero; otherwise it is set to a positive value.
6. If *`s`* is specified and an error occurs, control is transferred to *s*.

Restrictions

- Output from an Exception Handler is Unpredictable

  If you make your own exception handler, do not do any FORTRAN output from it. If you must do some, then call abort right after the output. This reduces the relative risk of a system freeze. FORTRAN I/O from an exception handler amounts to recursive I/O. See next paragraph.

- Recursive I/O Does not Work Reliably

  If you list a function in an I/O list, and if that function does I/O, then during runtime the execution may freeze, or some other unpredictable problem happens. This risk exists independent of using parallelization.

Example: Recursive I/O fails intermittently.

```
      WRITE(*,*) x, f(x)     ! Not allowed, f() does I/O.
      END
      FUNCTION F(X)
      WRITE(*,*) X
      RETURN
      END
```

Comments

If *u* specifies an external unit that is not connected to a file, an implicit OPEN operation is performed that is equivalent to opening the file with the following options.

```
      OPEN( u, FILE='FORT.u', STATUS='UNKNOWN',
&             ACCESS='SEQUENTIAL', FORM=fmt )
```

The value of *fmt* is 'FORMATTED' if the write is formatted, and 'UNFORMATTED' otherwise.

A simple unsubscripted array name specifies all of the elements of the array in memory storage order, with the leftmost subscript increasing more rapidly.

The record number for direct-access files starts from one onwards.

Namelist-directed output is permitted on sequential access files only.

Examples

Example 1: Formatted write with trap I/O errors and I/O status.

```
      WRITE( 1, 2, ERR=8, IOSTAT=N ) X, Y
      RETURN
      …
8     WRITE( *, * ) 'I/O error # ', N, ', on 1'
      STOP
      END
```

Example 2: Direct, unformatted write, trap I/O errors, and I/O status.

```
      …
      WRITE( 1, REC=3, IOSTAT=N, ERR=8 ) V
      …
4     CONTINUE
      RETURN
8     WRITE( *, * ) 'I/O error # ', N, ', on 1'
      END
```

Example 3: Direct, alternate syntax (equivalent to above example).

```
      …
      WRITE( 1 ' 3, IOSTAT=N, ERR=8 ) V
      …
4     CONTINUE
      RETURN
8     WRITE( *, * ) 'I/O error # ', N, ', on 1'
      END
```

Example 4: List-directed write to screen.

```
      WRITE( *, * ) A, V
or
      PRINT *, A, V
```

Example 5: Formatted write to an internal file.

```
      CHARACTER CA*16, L*8 /'abcdefgh'/, R*8 /'ijklmnop'/
      WRITE( CA, 1 ) L, R
1     FORMAT( 2 A8 )
```

Example 6: Write an entire array.

```
      DIMENSION V(5)
      WRITE( 3, '(5F4.1)') V
```

Example 7: Namelist-directed write.

```
      CHARACTER SAMPLE*16
      LOGICAL NEW*4
      REAL DELTA*4
      NAMELIST /G/ SAMPLE, NEW, DELTA
      …
      WRITE( 1, G )
or
      WRITE( UNIT=1, NML=G )
or
      WRITE( 1, NML=G )
```

4

Input and Output 5

This chapter is organized into the following sections.

5.1 General Concepts of FORTRAN I/O

Any operating system based on the UNIX operating system is not as record-oriented as FORTRAN. This operating system treats files as sequences of characters instead of collections of records. The FORTRAN runtime system keeps track of file formats and access mode during runtimes. It also provides the file facilities, including the FORTRAN libraries and the standard I/O library.

Logical Units

The FORTRAN default value for the maximum number of logical units that a program can have open at one time is 64. For current SunOS releases this limit is 256. A FORTRAN program can increase this limit beyond 64 by calling the `setrlim()` function. See the `man` page `setrlim`(2). If you are running `csh`, you can also do this with the `limit` or `unlimit` command; see `csh`(1).

The standard logical units 0, 5, and 6 are preconnected to the SunOS system as `stderr`, `stdin`, and `stdout`, respectively. These are not actual file names and cannot be used for opening these units. `INQUIRE` does not return these names and indicates that the above units are not named unless they have been opened to real files. However, these units can be redefined with an `OPEN` statement.

The names `stderr`, `stdin`, and `stdout` are meant to make error reporting more meaningful. To preserve error reporting, the system makes it is an error to close logical unit 0, although it can be reopened to another file.

If you want to open a file with the default file name for any preconnected logical unit, remember to close the unit first. Redefining the standard units can impair normal console I/O. An alternative is to use shell redirection to externally redefine the above units.

To redefine default blank control or the format of the standard input or output files, use the `OPEN` statement specifying the unit number and no file name, and use the options for the kind of blank control you want.

I/O Errors

Any error detected during I/O processing will cause the program to abort unless alternative action has been provided specifically in the program. Any I/O statement can include an `ERR=` clause (and `IOSTAT=` clause) to specify an alternative branch to be taken on errors (and return the specific error code). Read statements can include `END=`*n* to branch on end-of-file. File position and the value of I/O list items are undefined following an error. The `END=` will catch both EOF and error conditions; the `ERR=` will catch only error conditions.

If the user's program does not trap I/O errors, then before aborting, an error message is written to `stderr` with an error number in square brackets, [ ], and the logical unit and I/O state. The signal that causes the abort is `IOT`.

Error numbers less than 1000 refer to operating system errors; see `intro`(2). Error numbers greater than or equal to 1000 come from the I/O library.

For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace. For internal I/O, part of the string is printed with a vertical bar (|) at the current position in the string.

General Restriction

Do not reference a function in an I/O list if executing that function will cause an I/O statement to be executed. Example:

```
      WRITE( 1, 10)  Y, A + 2.0 * F(X)   !  Wrong if F() does I/O
```

Kinds of I/O

The four kinds of I/O are *formatted*, *unformatted*, *list-directed*, and `NAMELIST`.

The two modes of access to files are *sequential* and *direct*. When you open a file, the access mode is set to either sequential or direct. If you do not set it explicitly, you get sequential by default.

The two types of files are external and internal. An *external* file resides on a physical peripheral device, such as disk or tape. An *internal* file is a location in main memory, is of character type, and is either a variable, substring, array, array element, or field of a structured record.

Combinations of I/O

I/O combinations on *external* files:

Allowed	Sequential unformatted Sequential formatted Sequential list-directed Sequential `NAMELIST` Direct unformatted Direct formatted
Not allowed	Direct-access, list-directed I/O Direct-access, `NAMELIST` I/O `NAMELIST` I/O on internal files Unformatted, internal I/O

The following table shows combinations of I/O form, access mode, and physical file type.

Table 5-1 Summary of `f77` Input and Output

Kind of I/O		***Access Mode***	
Form	***File Type***	***Sequential***	***Direct***
Formatted	Internal	The file is a character variable, substring, array, or array element. ♦	The file is a character array; each record is one array element.
	External	Only formatted records of same or variable length.	Only formatted records, all the same length.
Unformatted	Internal	(not allowed)	(not allowed)
	External	Contains only unformatted records.	`READ`: Gets one logical record at a time. `WRITE`: Unfilled part of record is undefined.
List-directed	Internal	`READ`: Reads characters until EOF or I/O list is satisfied; WRITE: Writes records until list is satisfied. ♦	(not allowed)
	External	Uses standard formats based on type of variable and size of element. Blanks or commas are separators. Any columns.	(not allowed)
`NAMELIST`	Internal	(not allowed)	(not allowed)
	External	`READ`: Reads records until it finds " $*groupname*" in cols 2-80. Then reads records searching for names in that group, and stores data in those variables. Stops reading on "$" or eof. `WRITE`: Writes records showing groupname and each variable name with value.	(not allowed)

Avoid list-directed internal writes. The number of lines and items per line varies with the values of items.

Print Files

You get a *print* file by using the nonstandard `FORM='PRINT'` in `OPEN`. ♦

```
      OPEN ( ..., FORM='PRINT', ... )
```

This specifier works for sequential access files only.

Definition

A *print* file has the following features:

- With *formatted* output you get *vertical format control* for that logical unit:
  - Column one is not printed
  - If column one is blank, 0, or 1, then vertical spacing is one line, two lines, or top of page (respectively).
- With *list-directed* output you get for that logical unit:
  - Column one is not printed.

In general, if you open a file with `FORM='PRINT'` then for that file list-directed output does *not* provide the FORTRAN Standard blank in column one, otherwise it does provide that blank. The `FORM='PRINT'` is for one file per call.

If you compile with the `-oldldo` option (old list-directed output), then all files written by the program do list-directed output *without* that blank in column one, otherwise they all get that blank. The `-oldldo` option is global.

The `INQUIRE` *Statement*

The `INQUIRE` statement returns '`PRINT'` in the `FORM` variable for logical units opened as 'print' files. It returns -1 for the unit number of an unopened file.

Special Uses of `OPEN`

If a logical unit is already open, an `OPEN` statement using the `BLANK` option does nothing but redefine that option.

As a nonstandard extension, if a logical unit is already open, an `OPEN` statement using the `FORM` option does nothing but redefine that option. ♦

These forms of the OPEN statement need not include the file name, and must not include a file name if UNIT refers to standard input, output, or standard error.

If you connect a unit with OPEN and do not use the file name parameter, then you get the default file name fort.*nn*, where *nn* is the unit number. Therefore, to redefine the standard output as a *print* file, use:

```
OPEN( UNIT=6, FORM='PRINT')
```

Scratch Files

If you create a scratch file, it will normally disappear after execution is completed.

Example: Create a scratch file.

```
OPEN( UNIT=7, STATUS='SCRATCH' )
```

To prevent a temporary file from disappearing after execution is completed, you must execute a CLOSE statement with STATUS='KEEP'. (KEEP is the default status for all other files.)

Example: Close a scratch file that you want to get back to later.

```
CLOSE( UNIT=7, STATUS='KEEP' )
```

Remember to get the real name of the scratch file, using INQUIRE if you want to reopen it later.

Changing I/O Initialization with IOINIT

Traditional FORTRAN environments usually assume carriage control on all logical units. They usually interpret blank spaces on input as zeroes and often provide attachment of global file names to logical units at runtime. The routine IOINIT (3F) can be called to specify these I/O control parameters.

- Recognize carriage control for all formatted files.
- Ignore trailing and embedded blanks in input files.
- Position files at the beginning or end upon opening.
- Preattach file names of a specified pattern with logical units.

Example: IOINIT and logical unit preattachment.

Consider the following call.

```
      CALL IOINIT ( .TRUE., .FALSE., .FALSE., 'FORT', .FALSE.)
```

For the above call, the FORTRAN runtime system looks in the environment for names of the form FORT*nn*, and then it opens the corresponding logical unit for sequential formatted I/O.

With the above example, suppose your program opened unit 7 as follows.

```
      OPEN( UNIT=07, FORM='FORMATTED' )
```

For the above OPEN, the FORTRAN runtime system looks in the environment for the FORT07 file, and connects it to unit 7.

In general, names must be of the form *PREFIXnn*, where the particular *PREFIX* is specified in the call to IOINIT, and *nn* is the logical unit to be opened. Unit numbers less than 10 must include the leading '0'. For details, see IOINIT(3F).

Example: Attach external files ini1.inp and ini1.out to units 1and 2.

sh:

```
demo$ TST01=ini1.inp
demo$ TST02=ini1.out
demo$ export TST01 TST02
```

csh:

```
demo% setenv TST01 ini1.inp
demo% setenv TST02 ini1.out
```

Example: Attach files `ini1.inp` & `ini1.out` to units 1 & 2.

```
demo$ cat ini1.f
      CHARACTER PRFX*8
      LOGICAL CCTL, BZRO, APND, VRBOSE
      DATA CCTL, BZRO, APND, PRFX, VRBOSE
&          /.TRUE., .FALSE., .FALSE., 'TST', .FALSE. /
C
      CALL IOINIT( CCTL, BZRO, APND, PRFX, VRBOSE )
      READ( 1, *) I, B, N
      WRITE( *, *) 'I = ', I, ' B = ', B, ' N = ', N
      WRITE( 2, *) I, B, N
      END
demo$ f77 ini1.f
ini1.f:
 MAIN:
demo$ a.out
  I = 12 B = 3.14159012 N = 6
demo$ ■
```

`IOINIT` should prove adequate for most programs as written. However, it is written in FORTRAN so that it can serve as an example for similar user-supplied routines. A copy can be retrieved as follows.

Solaris 2.x:

```
demo$ cp /opt/SUNWspro/SC3.0.1/src/ioinit.f  .
```

Solaris 1.x:

```
demo% cp /usr/lang/SC3.0.1/src/ioinit.f  .
```

5.2 Direct Access

A direct-access file contains a number of records that are written to or read from by referring to the record number. Direct access is also called random access.

- Records must be all the same length.
- Records are usually all the same type.
- A logical record in a direct access, external file is a string of bytes of a length specified when the file is opened.

- Read and write statements must not specify logical records longer than the original record size definition.
- Shorter logical records are allowed.
  - Unformatted direct writes leave the unfilled part of the record undefined.
  - Formatted direct writes pass the unfilled record with blanks.
- Each `READ` operation acts on exactly *one* record
- In using direct unformatted I/O, you should be careful with the number of values your program expects to read.
- Direct access `READ` and `WRITE` statements have an argument, `REC=`*n*, which gives the record number to be read or written. (An alternate, nonstandard form is `'`*n*.)

Unformatted I/O

Example: Direct-access *unformatted.*

```
      OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=20,
&               FORM='UNFORMATTED', ERR=90 )
      READ( 2, REC=13, ERR=30 ) X, Y
      READ( 2 ' 13, ERR=30 ) X, Y   ! ← Alternate form♦
```

This opens a file for direct-access, unformatted I/O, with a record length of 20 characters, then reads the thirteenth record as is.

Formatted I/O

Example: Direct-access, *formatted.*

```
      OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=20,
&               FORM='FORMATTED', ERR=90 )
      READ( 2, FMT='(I10,F10.3)', REC=13, ERR=30 ) A, B
```

This opens a file for direct-access, formatted I/O, with a record length of 20 characters, then reads the thirteenth record and converts it according to the "`(I10,F10.3)`" format.

5.3 Internal Files

An internal file is a character-string object such as a constant, variable, substring, array, element of an array, or field of a structured record — all of type character. For a variable or substring, there is only a single record in the file but for an array, each array element is a record.

Sequential Formatted I/O

On internal files, the FORTRAN Standard includes only sequential formatted I/O. (I/O is not a precise term to use here, but internal files are dealt with using `READ` and `WRITE` statements.) Internal files are used by giving the name of the character object in place of the unit number. The first read from a sequential-access internal file always starts at the beginning of the internal file; and similarly for a write.

Example: Sequential, formatted reads.

```
      CHARACTER X*80
      READ( 5, '(A)' ) X
      READ( X, '(I3,I4)' ) N1, N2
```

The above reads a print-line image into `X` and then reads two integers from `X`.

Direct Access I/O

`f77` extends direct I/O to internal files.♦

This is like direct I/O on external files, except that the number of records in the file cannot be changed. In this case, a record is a single element of an array of character strings.

Example: Direct-access read of third record of the internal file `LINE`.

```
demo$ cat intern.f
      CHARACTER LINE(3)*14
      DATA LINE(1) / ' 81 81 ' /
      DATA LINE(2) / ' 82 82 ' /
      DATA LINE(3) / ' 83 83 ' /
      READ ( LINE, FMT='(2I4)', REC=3 ) M, N
      PRINT *, M, N
      END
demo$ f77 -silent intern.f
demo$ a.out
  83 83
demo$ ■
```

5.4 Formatted I/O

Description

- The list items are processed in the order they appear in the list.
- Any list item is completely processed before the next item is started.
- Each formatted sequential access reads or writes one or more logical records.

Input

In general, a formatted read statement does the following:

- Reads character data from the external record (or from an internal file).
- Converts the items of the list from character to binary form.
- Conversion is according to the instructions in the associated format.
- Puts converted data into internal storage for each list item of the list.

Example: Formatted read.

```
      READ( 6, 10 ) A, B
10    FORMAT( F8.3, F6.2 )
```

Output

In general, a formatted write statement does the following:

- Gets data from internal storage for each list item specified by the list.
- Converts the items from binary to character form.
- Conversion is according to the instructions in the associated format.
- Transfers the items to the external record (or to an internal file).
- Formatted output records are terminated with newline characters.

Example: Formatted write.

```
      REAL  A / 1.0 /,  B / 9.0 /
      WRITE( 6, 10 ) A, B
10    FORMAT( F8.3, F6.2 )
```

- For formatted write statements, logical record length is determined by the format statement interacting with the list of input or output variables (I/O list) at execution time.
- For formatted write statements, if the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*).
- For formatted read statements, if there are fewer items in the list than there are data fields, the extra fields are ignored.

Format Specifiers

Table 5-2 Format Specifiers

Purpose	***FORTRAN 77***	f77 ***Extensions***
Blank control	BN, BZ	B
Carriage control	/, *space*, 0, 1	$
Character edit	*n*H, A*w*, 'aaa'	"aaa", A
Floating-point edit	D*w.d*E*e*, E*w.d*E*e*, F*w.d*E*e*, G*w.d*E*e*	E*w.d.e*, D*w.d.e*, G*w.d.e*
Hexadecimal edit		Z*w.m*
Integer edit	I*w.m*	
Logical edit	L*w*	
Octal edit		O*w.m*
Position control	*n*X, T*n*, TL*n*, TR*n*	*n*T, T, X
Radix control		*n*R, R
Remaining characters		Q
Scale control	*n*P	P
Sign control	S, SP, SS	SU
Terminate a format	:	
Variable format expression		< *e* >

Specifiers can be uppercase as well as lowercase characters in format statements and in all the alphabetic arguments to the I/O library routines.

The *w*, *m*, *d*, *e* Parameters (as in G*w.d*E*e*)

- *w* specifies that the field occupies *w* positions.
- *m* specifies the insertion of leading zeros to a width of *m*.
- *d* specifies the number of digits to the right of the decimal point.
- *e* specifies the width of the exponent field.

Defaults for *w*, *d*, *e*

You can write field descriptors A, D, E, F, G, I, L, O, or Z without the *w*, *d*, or *e* field indicators. ♦ If these are left unspecified, the appropriate defaults will be used, based on the data type of the I/O list element. See Table 5-3. Typical format field descriptor forms that use *w*, *d*, or *e* include:

```
Aw, Iw, Lw, Ow, Zw, Dw.d, Ew.d, Gw.d, Ew.dEe, Gw.dEe
```

Example: With default *w*=7 for INTEGER*2, and since 161 decimal = A1 hex.

```
      INTEGER*2 M
      M = 161
      WRITE ( *, 8 ) M
8     FORMAT ( Z )
      END
```

The above example displays as shown below.

```
demo$ f77 def1.f
def1.f:
 MAIN:
demo$ a.out
ΔΔΔΔΔa1
demo$ ■
```

↑ column 6

The defaults for *w*, *d*, and *e* are summarized below.

Table 5-3 Default *w*, *d*, *e* Values in Format Field Descriptors

Field Descriptor	*List Element*	*w*	*d*	*e*
I,O,Z	BYTE	7	-	-
I,O,Z	INTEGER*2, LOGICAL*2	7	-	-
I,O,Z	INTEGER*4, LOGICAL*4	12	-	-
O,Z	REAL*4	12	-	-
O,Z	REAL*8	23	-	-
O,Z	REAL*16, COMPLEX*32	44	-	-
L	LOGICAL	2	-	-
F,E,D,G	REAL, COMPLEX*8	15	7	2
F,E,D,G	REAL*8, COMPLEX*16	25	16	2
F,E,D,G	REAL*16, COMPLEX*32	42	33	3
A	LOGICAL*1	1	-	-
A	LOGICAL*2, INTEGER*2	2	-	-
A	LOGICAL*4, INTEGER*4	4	-	-
A	REAL*4, COMPLEX*8	4	-	-
A	REAL*8, COMPLEX*16	8	-	-
A	REAL*16, COMPLEX*32	16	-	-
A	CHARACTER*n	n	-	-

For complex items, the value for *w* is for each real component. Default for the A descriptor with character data is the declared length of the corresponding I/O list element. The REAL*16 and COMPLEX*32 are for *SPARC only.*

Apostrophe Editing ('aaa')

The apostrophe edit specifier is in the form of a character constant. It causes characters to be written from the enclosed characters (including blanks) of the edit specifier itself. An apostrophe edit specifier must not be used on input. The width of the field is the number of characters contained in, but not including, the delimiting apostrophes. Within the field, two consecutive apostrophes with no intervening blanks are counted as a single apostrophe. You can use quotes in a similar way.

Example: apos.f, apostrophe edit (two equivalent ways).

```
      WRITE( *, 1 )
1     FORMAT( 'This is an apostrophe ''.')
      WRITE( *, 2 )
2     FORMAT( "This is an apostrophe '.")
      END
```

The above writes: This is an apostrophe '. twice.

Blank Editing (B, BN, BZ)

The B, BN, and BZ edit specifiers control interpretation of imbedded and trailing blanks for numeric input.

The following blank specifiers are available.

BN

If BN precedes a specification, a nonleading blank in the input data is considered *null*, and is ignored.

BZ

If BZ precedes a specification, a nonleading blank in the input data is considered *zero*.

B

If B precedes a specification, it returns interpretation to the default mode of blank interpretation. This is consistent with S, which returns to default sign control. ♦

Without any specific blank specifiers in the format, nonleading blanks in numeric input fields are normally interpreted as zeros or ignored, depending on the value of the "BLANK=" suboption of OPEN currently in effect for the unit. The default value for that suboption is *ignore*, so if you use defaults for both BN/BZ/B and "BLANK=", you get *ignore*.

Example: Read and print the same data once with BZ and once with BN.

```
demo$ cat bz1.f
*                        12341234
     CHARACTER LINE*18 / ' 82 82 ' /
     READ ( LINE, '( I4, BZ, I4 ) ') M, N
     PRINT *, M, N
     READ ( LINE, '( I4, BN, I4 ) ') M, N
     PRINT *, M, N
     END
demo$ f77 -silent  bz1.f
demo$ a.out
  82 8200
  82 82
demo$ ■
```

Rules and Restrictions for Blank Control

- Blank control specifiers apply to *input* only.
- A blank control specifier remains in effect until another blank control specifier is encountered, or format interpretation is complete.
- The B, BN, and BZ specifiers affect only I, F, E, D, and G editing.

Carriage Control ($, space, 0, 1)

Dollar $

The special edit descriptor $ suppresses the carriage return. ♦
The action does *not* depend on the first character of the format. It is used typically for console prompts. For instance, you can use this to make a typed response follow the output prompt on the same line. This edit descriptor is constrained by the same rules as the colon (:).
Example: The $ carriage control.

```
* dol1.f The $ edit descriptor with space
     WRITE ( *, 2 )
2    FORMAT (' Enter the node number: ', $ )
     READ ( *, * ) NODENUM
     END
```

The above produces a displayed prompt and user input response such as:

```
Enter the node number:  82
```

The first character of the format is printed out, in this case, a blank. For an *input* statement, the $ descriptor is ignored.

space,0,1, +

The following first-character slew controls and actions are provided.

Table 5-4 Carriage Control with Blank, 0, 1, +

Character	*Vertical spacing before printing*
Blank	One line
0	Two lines
1	To first line of next page
+	No advance (`stdout` only, not files)

If the first character of the format is not *space*, `0`, `1`, or +, then it is treated as a space, and it is not printed.

The behavior of the slew control character + is different for standard output and for file output. For output directly to a *file*, the **+** code for *no advance* is *not* implemented, and, like any other character in the first position of a record written to a print file, is dropped.

Standard Output: space, 0, 1, and + work for `stdout` if piped through `asa`.

Example: First-character formatting, standard output piped through `asa`.

```
demo$ cat slew1.f
      WRITE( *, '("abcd")')
      WRITE( *, '(" efg")') ! The blank single spaces
      WRITE( *, '("0hij")') ! The "0" double spaces
      WRITE( *, '("1klm")') ! The "1" starts this on a new page
      WRITE( *, '("+", T5, "nop")') ! The "+" starts this at col 1 of latest line
      END
demo$ f77 -silent slew1.f
demo$ a.out | asa | lpr
demo$
```

The program `slew1.f` produces file `slew1.out`, as printed by `lpr`, below.

printer

```
bcd
efg

hij
```

```
klmnop                ← This starts on a new page.  The "+" of "+nop" is obeyed
```

The results are different on a screen. The tabbing puts in spaces.

screen

```
demo$ cat slew1.out
bcd
efg

hij
```

```
   nop                ← This starts on a new page.  The "+" of "+nop" is obeyed
demo$
```

See `asa` (1).

File Output: The space, 0, and 1 (but not +) work for a *file* opened with:

- Sequential access
- `FORM='PRINT'`

Example: First-character formatting, file output.

```
demo$ cat slew2.f
     OPEN( 1,FILE='slew.out',FORM='PRINT' )
     WRITE( 1, '("abcd")')
     WRITE( 1, '(" efg")')
     WRITE( 1, '("0hij")')
     WRITE( 1, '("1klm")')
     WRITE( 1, '("+", T5, "nop")')
     CLOSE( 1, STATUS='KEEP')
     END
demo$ f77 -silent slew2.f
demo$ a.out
```

The program `slew2.f` produces file `slew2.out`, as printed by `lpr`, below.

```
bcd
efg

hij
```

```
klm                    ← This starts on a new page.  The "+" of "+nop" is ignored
   nop
```

Slew control codes '`0`' and '`1`' in column one are in the output file as '`\n`' and '`\f`', respectively.

*Character Editing (*A*)*

The A specifier is used for character type data items. The general form is

```
A  [ w ]
```

- On input, character data is stored in the corresponding list item.
- On output, the corresponding list item is displayed as character data.
- If *w* is omitted, then:
  - For character data type variables, it assumes the size of the variable.
  - For noncharacter data type variables, it assumes the maximum number of characters that fit in a variable of that data type. This is nonstandard.

Each of the following examples read into a size *n* variable (CHARACTER*_n_), for various values of *n* (for instance, for *n* = 9).

```
      CHARACTER C*9
      READ '( A7 )', C
```

And for the various values of *n*, in `CHARACTER C*`*n*, you have the following.

Size n	***9***	***7***	***4***	***1***
Data	NodeΔId	NodeΔId	NodeΔId	NodeΔId
Format	A7	A7	A7	A7
Memory	NodeΔIdΔΔ	NodeΔId	eΔId	d

The Δ indicates a blank space.

Example: Output strings of 3, 5, and 7 characters, each in a 5 character field.

```
      PRINT 1, 'The', 'whole', 'shebang'
1     FORMAT( A5 / A5 / A5 )
      END
```

The above program displays:

```
ΔΔThe
whole
sheba
```

The maximum characters in noncharacter types are summarized below.

Table 5-5 Maximum Characters in Noncharacter Type Hollerith (nHaaa)

Type of List Item	***Maximum Number of characters***
BYTE	1
LOGICAL*1	1
LOGICAL*2	2
LOGICAL*4	4
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*4	4
REAL*8	8
REAL*16 *(SPARC only)*	16
DOUBLE PRECISION	8
COMPLEX	8
COMPLEX*8	8
COMPLEX*16	16
COMPLEX*32 *(SPARC only)*	32
DOUBLE COMPLEX	16

In f77 you can use Hollerith constants wherever a character constant can be used in FORMAT statements, assignment statements, and DATA statements.♦ These constants are not recommended. FORTRAN 77 does not have these old Hollerith (*n* H) notations, although the FORTRAN Standard recommends implementing the Hollerith feature in order to improve compatibility with old

programs. But such constants cannot be used as input data elements in list-directed or NAMELIST input. For example, the two formats below are equivalent.

```
10   FORMAT( 8H Code = , A6 )
20   FORMAT( ' Code = ', A6 )
```

In f77, commas between edit descriptors are generally optional.

```
10   FORMAT( 5H flex 4Hible )
```

Read into Hollerith Edit Descriptor

For compatibility with older programs, f77 also allows READs into Hollerith edit descriptors. ♦

Example. Read into hollerith edit descriptor. Note that there is no *list* in the READ statement.

```
demo$ cat hol1.f
      WRITE( *, 1 )
1     FORMAT( 6Holder )
      READ( *, 1 )
      WRITE( *, 1 )
      END
demo$ f77 hol1.f
hol1.f:
 MAIN
demo$ a.out
older
newer
newer
demo$ ■
```

In the above, if the format is a runtime format (variable format), then the above reading into the actual format does not work, and the format remains unchanged. That is, the following fails:

```
      CHARACTER F*18 / '(A8)' /
      READ(*,F)          ! <- Does not work.
      …
```

But obviously there are better ways to do it anyway.

Integer Editing (`I`*)*

The `I` specifier is used for decimal integer data items. The general form is

```
I [w [ . m ] ]
```

The `I` *w* and `I` *w.m* edit specifiers indicate that the field to be edited occupies *w* positions. The specified input/output list item must be of type integer. On input, the specified list item will become defined with an integer datum. On output, the specified list item must be defined as an integer datum.

On input, an `I` *w.m* edit specifier is treated identically to an `I` *w* edit specifier.

The output field for the `I` *w* edit specifier consists of

- Zero or more leading blanks followed by
- Either a minus if the value is negative, or an optional plus, followed by
- The magnitude of the value in the form on an unsigned integer constant without leading zeros.

An integer constant always has at least one digit.

The output field for the `I` *w.m* edit specifier is the same as for the `I` *w* edit specifier, except that the unsigned integer constant consists of at least *m* digits, and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the item is zero, the output field consists of only blank characters, regardless of the sign control in effect.

Example: `int1.f`, Integer input.

```
      CHARACTER LINE*8 / '12345678' /
      READ( LINE, '(I2, I3, I2 )') I, J, K
      PRINT *, I, J, K
      END
```

The program above displays:

```
      12 345 67
```

Example: `int2.f`, integer output.

```
      N = 1234
      PRINT 1, N, N, N, N
1     FORMAT( I6 / I4 / I2 / I6.5 )
      END
```

The program above displays:

```
  1234
1234
**
01234
```

*Logical Editing (*L*)*

The L specifier is used for logical data items. The general form is:

```
 L  w
```

The L *w* edit specifier indicates that the field occupies *w* positions. The specified input-output list item must be of type LOGICAL. On input, the list item will become defined with a logical datum. On output, the specified list item must be defined as a logical datum.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F can be followed by additional characters in the field. The logical constants .TRUE. and .FALSE. are acceptable as input. The output field consists of *w*-1 blanks followed by a T for true or F for false.

Example: `log1.f`, logical output.

```
      LOGICAL A*1 /.TRUE./, B*2 /.TRUE./, C*4 /.FALSE./
      PRINT '( L1 / L2 / L4 )', A, B, C
      END
```

The program above displays:

```
T
ΔT
ΔΔΔF
```

Example: `log2.f`, logical input.

```
      LOGICAL*4 A
1     READ '(L8)', A
      PRINT *, A
      GO TO 1
      END
```

The program above accepts any of these as valid input data.

```
t true T TRUE .t .t. .T .T. .TRUE. TooTrue
f false F FALSE .f .F .F. .FALSE. Flakey
```

Octal and Hexadecimal Editing (O,Z)

The O and Z field descriptors for a FORMAT statement are for octal and hexadecimal integers, respectively, but they can be used with any data type.♦

The general form is

O*w*[.*m*]
Z*w*[.*m*]

where *w* is the number of characters in the external field, and for output, *m*, if specified, determines the total number of digits in the external field (that is, if there are fewer than *m* nonzero digits, the field is zero-filled on the left to a total of *m* digits). The *m* has no effect on input.

Octal and Hex Input

A READ, with the O or Z field descriptors in the FORMAT, reads in *w* characters as octal or hexadecimal, respectively, and assigns the value to the corresponding member of the I/O list.

Example: Octal input, the external data field is as follows.

```
654321
```

↑ column 1

The program that does the input is the following.

```
      READ ( *, 2 ) M
2     FORMAT ( O6 )
```

The above data and program result in the octal value 654321 being loaded into the variable M Further examples are included in the table below.

Table 5-6 Sample Octal/Hex Input Values

Format	*External Field*	*Internal (Octal or Hex) Value*
O4	1234Δ	1234
O4	16234	1623
O3	97ΔΔΔ	Error: “9” not allowed
Z5	A23DEΔ	A23DE
Z5	A23DEF	A23DE
Z4	95.AF2	Error: “.” not allowed

General Rules for Octal and Hex Input

- For octal values, the external field can contain only numerals 0 through 7.
- For hexadecimal values, the external field can contain only numerals 0 through 9 and the letters A through F or a through f.
- Signs, decimal points, and exponent fields are not allowed.
- All-blank fields are treated as having a value of zero.
- If a data item is too big for the corresponding variable, an error message is displayed.

Octal and Hex Output

A WRITE, with the O or Z field descriptors in the FORMAT, writes out values as octal or hexadecimal integers, respectively. It writes to a field that is *w* characters wide, right-justified.

Example: Hex Output.

```
      M = 161
      WRITE ( *, 8 ) M
8     FORMAT ( Z3 )
      END
```

The program above displays A1 (161 decimal = A1 hex).

```
ΔA1
```

↑ column 2

Further examples are included in the table below.

Table 5-7 Sample Octal/Hex Output Value

Format	***Internal (Decimal) Value***	***External (Octal/Hex) Representation***
O6	32767	Δ77777
O2	14251	**
O4.3	27	Δ033
O4.4	27	0033
O6	-32767	100001
Z4	32767	7FFF
Z3.3	2708	A94
Z6.4	2708	ΔΔ0A94
Z5	-32767	Δ8001

General Rules for Octal and Hex Output

- Negative values are written as if unsigned; no negative sign is printed.
- The external field is filled with leading spaces, as needed, up to the width *w*.
- If the field is too narrow, it is filled with asterisks.
- If *m* is specified, the field is left-filled with leading zeros, to a width of *m*.
- In general, do not use these descriptors for printing character strings, but restrict usage to 4 or 8 byte numeric data only.

*Positional Editing (*T,*n*T,TR*n*,TL*n*,*n*X*)*

For horizontal positioning along the print line, `f77` supports the forms

TR*n*, TL*n*, T*n*, *n*T, T

where *n* is a strictly positive integer.The format specifier T can appear by itself, or be preceded or followed by a positive nonzero number.

T*n* — *Absolute Columns*

This reads from the *nth* column or writes to the *nth* column.

TL*n* — *Relative Columns*

This reads from the *nth* column to the *left* or writes to the *nth* column to the *left*.

TR*n* — *Relative Columns*

This reads from the *nt*h column to the *right* or writes to the *nth c*olumn to the *right*.

*n*TL — *Relative Tab Stop*

This tabs to the *nth* tab stop for both read and write. If *n* is omitted, this uses *n* = 1 and tabs to the *next* tab stop.

TL — *Relative Tab Stop*

This tabs to the *next* tab stop for both read and write. This is the same as the *n*TL with *n* omitted; it tabs to the *next* tab stop.

Rules and Restrictions for Tabbing

- Tabbing right beyond the end of an input logical record is an error.
- Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record.
- Nondestructive tabbing is implemented for both internal and external formatted I/O. Nondestructive tabbing means that tabbing left or right on output does not destroy previously written portions of a record.

- Tabbing right on output causes unwritten portions of a record to be filled with blanks.
- Tabbing left requires that the logical unit allows a `seek`. Therefore, it is not allowed in I/O to or from a terminal or pipe.
- Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with the X edit specifier writes blanks on the output.
- Tab stops are hard-coded every eight columns.

*n*X — *Positions*

The *n*X edit specifier indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position.

On input, the *n*X edit specifier advances the record pointer by *n* positions, skipping *n* characters.

A position beyond the last character of the record can be specified if no characters are transmitted from such positions.

On output, the *n*X specifier writes *n* blanks.

The *n* defaults to 1.

Example: Input, T*n* (absolute tabs).

```
demo$ cat rtab.f
     CHARACTER C*2, S*2
     OPEN( 1, FILE='mytab.data')
     DO I = 1, 2
          READ( 1, 2 ) C, S
2         FORMAT( T5, A2, T1, A2 )
          PRINT *, C, S
     END DO
     END
demo$ ■
```

The 2-line data file is as follows.

```
demo$ cat mytab.data
defguvwx
12345678
demo$ ■
```

The run and the output are as follows.

```
demo$ a.out
uvde
5612
demo$ ■
```

The above example first reads columns 5 and 6, then columns 1 and 2.

Example: Output T*n* (absolute tabs), this program writes an output file.

```
demo$ cat otab.f
      CHARACTER C*20 / "12345678901234567890" /
      OPEN( 1, FILE='mytab.rep')
      WRITE( 1, 2 ) C, ":", ":"
2     FORMAT( A20, T10, A1, T20, A1 )
      END
demo$ ■
```

The output file is as follows.

```
demo$ cat mytab.rep
123456789:123456789:
demo$ ■
```

The above example wrote 20 characters, then changed columns 10 and 20.

Example: Input, TR*n* and TL *n* (relative tabs), the program is as follows.

```
demo$ cat rtabi.f
      CHARACTER C, S, T
      OPEN( 1, FILE='mytab.data')
      DO I = 1, 2
           READ( 1, 2 ) C, S, T
2          FORMAT( A1, TR5, A1, TL4, A1 )
           PRINT *, C, S, T
      END DO
      END
demo$ ■
```

The 2-line data file:

```
demo$ cat mytab.data
defguvwx
12345678
demo$ ■
```

The run and the output:

```
demo$ a.out
dwg
174
demo$ ■
```

The above example reads column 1, then tabs right 5 to column 7, then tabs left 4 to column 4.

Example: Output TR *n* and TL *n* (relative tabs), this program writes an output file.

```
demo$ cat rtabo.f
      CHARACTER C*20 / "12345678901234567890" /
      OPEN( 1, FILE='rtabo.rep')
      WRITE( 1, 2 ) C, ":", ":"
2     FORMAT( A20, TL11, A1, TR9, A1 )
      END
demo$  ■
```

The run shows nothing, but you can list the mytab.rep output file.

```
demo$ cat rtabo.rep
123456789:123456789:
demo$ ■
```

The above program wrote 20 characters, then tabbed left 11 to column 10, then tabbed right 9 to column 20.

Quotes Editing ("aaa"*)*

The quotes edit specifier is in the form of a character constant.♦
It causes characters to be written from the enclosed characters (including blanks) of the edit specifier itself. A quotes edit specifier must not be used on input.

The width of the field is the number of characters contained in, but not including, the delimiting quotes. Within the field, two consecutive quotes with no intervening blanks are counted as a single quote. You can use apostrophes in a similar way.
Example: `quote.f`, (two equivalent ways).

```
      WRITE( *, 1 )
1     FORMAT( 'This is a quote ".' )
      WRITE( *, 2 )
2     FORMAT( "This is a quote "." )
      END
```

This writes the message "`This is a quote ".`" twice.

*Radix Control (*R*)*

The format specifier is R or *n*R, where $2 \leq n \leq 36$. ♦
If *n* is omitted, the default decimal radix is restored.

This lets you specify radixes other than 10 for formatted integer I/O conversion. The specifier is patterned after P, the scale factor for floating-point conversion. It remains in effect until another radix is specified or format interpretation is complete. The I/O item is treated as a 32-bit integer.

Example: Radix 16, the format for an *unsigned*, *hex*, integer, 10 places wide, zero-filled to 8 digits, is `(su, 16r, I10.8)`, as in the following:

The "SU" is described under *Sign Editing*, below.

```
demo$ cat radix.f
      integer i / 110 /
      write( *, 1 ) i
1     format( su, 16r, I10.8 )
      end
demo$ f77 -silent radix.f
demo$ a.out
 ΔΔ0000006e
demo$ ■
```

*Real Editing (*D,E,F,G*)*

The D, E, F, and G specifiers are for decimal real data items.

D Editing

The D specifier is for the exponential form of decimal double-precision items. The general form is

```
D [ w [ .d ] ]
```

The D *w* and D *w.d* edit specifiers indicate that the field to be edited occupies *w* positions. The *d* indicates that the fractional part of the number (the part to the right of the decimal point) has *d* digits. However, if the input datum contains a decimal point, that decimal point overrides the *d* value.

On input, the specified list item will become defined with a real datum. On output, the specified list item must be defined as a real datum.

In an output statement, the D edit descriptor does the same thing as the E edit descriptor, except that a D is used in place of an E. The output field for the D *w.d* edit specifier has width *w*. The value is right-justified in that field. The field consists of zero or more leading blanks followed by either a minus if the value is negative, or an optional plus, followed by the magnitude of the value of the list item rounded to *d* decimal digits.

The *w* must allow for a minus sign, at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal point. Therefore it must be the case that w ≥ d+3.

Example: Real input with D editing. Program Dinp.f.

```
      CHARACTER LINE*24 / '12345678 23.5678 .345678' /
      READ( LINE, '( D8.3, D8.3, D8.3 )') R, S, T
      PRINT '( D10.3, D11.4, D13.6 )', R, S, T
      END
```

The above displays:

```
0.123D+05 0.2357D+02 0.345678D+00
```

In the above example, the first input data item has no decimal point, so the `D8.3` determines the decimal point. The other input data items have decimal points, so those decimal points override the `D` edit descriptor as far as decimal points are concerned.

Example: Real Output with `D` editing. The program `Dout.f`.

```
      R = 1234.678
      PRINT 1, R, R, R
1     FORMAT( D9.3 / D8.4 / D13.4 )
      END
```

The above displays:

```
0.123D+04
********
ΔΔΔ0.1235D+04
```

In the above example, the second printed line is asterisks because the `D8.4` does not allow for the sign; in the third printed line the `D13.4` results in three leading blanks.

E Editing

The `E` specifier is for the exponential form of decimal real data items. The general form is

```
   E [ w [ .d ] [ Ee ] ]
```

- The *w* indicates that the field to be edited occupies *w* positions.
- The *d* indicates that the fractional part of the number (the part to the right of the decimal point) has *d* digits. However, if the input datum contains a decimal point, that decimal point overrides the *d* value.
- The *e* indicates the number of digits in the exponent field. Default: 2.

The specified input/output list item must be of type real. On input, the specified list item will become defined with a real datum. On output, the specified list item must be defined as a real datum.

The output field for the E *w.d* edit specifier has width *w*. The value is right-justified in that field. The field consists of zero or more leading blanks followed by either a minus if the value is negative, or an optional plus, followed by a zero, followed by a decimal point, followed by the magnitude of the value of the list item rounded to *d* decimal digits, followed by an exponent.

For the form E*w.d*

If | exponent | ≤ 99, it has the form E±*nn* or 0±*nn*.

If 99 ≤ | exponent | ≤ 999, it has the form ±*nnn*.

For the form E*w.d*E*e*

If | exponent | ≤ (10^e) - 1, then the exponent has the form ±*nnn*.

For the form D*w.d*

If | exponent | ≤ 99, it has the form D±*nn* or E±*nn* or 0±*nn*.

If 99 ≤ | exponent | ≤ 999, it has the form ±*nnn*.

where *n* is any digit.

The sign in the exponent is required.

The *w* need not allow for a minus sign, but must allow for a zero, the decimal point, and *d* digits to the right of the decimal point, and an exponent. Therefore for nonnegative numbers, *w* ≥ *d*+6, or if *e* is present then *w* ≥ *d*+*e*+4. For negative numbers, *w* ≥ *d*+7, or if *e* is present then *w* ≥ *d*+*e*+5.

Example: Real input with E editing. Program Einp.f.

```
* 123456789 23456789012 23456789012
      CHARACTER L*40/'1234567E2 1234.67E-3 12.4567 '/
      READ( L, '( E9.3, E12.3, E12.6 )') R, S, T
      PRINT '( E15.6, E15.6, E15.7 )', R, S, T
      END
```

The above displays:

```
ΔΔΔ0.123457E+06ΔΔΔ0.123467E+01ΔΔ0.1245670E+02
```

In the above example, the first input data item has no decimal point, so the E9.3 determines the decimal point. The other input data items have decimal points, so those decimal points override the D edit descriptor as far as decimal points are concerned.

Example: Real Output with E editing. The program Eout.f.

```
      R = 1234.678
      PRINT 1, R, R, R
1     FORMAT( E9.3 / E8.4 / E13.4 )
      END
```

The above displays:

```
0.123E+04
********
ΔΔΔ0.1235E+04
```

In the above example, the E8.4 does not allow for the sign, so we get asterisks. Also the extra wide field of the E13.4 results in three leading blanks.

Example: Real Output with E*w.d*E*e* editing. The program EwdEe.f.

```
      REAL  X / 0.000789 /
      WRITE(*,'( E13.3)') X
      WRITE(*,'( E13.3E4)') X
      WRITE(*,'( E13.3E5)') X
      END
```

The above displays:

```
ΔΔΔΔ0.789E-03
ΔΔ0.789E-0003
Δ0.789E-00003
```

F Editing

The F specifier is for decimal real data items. The general form is

```
F [ w [ .d ] ]
```

The F*w* and F*w.d* edit specifiers indicate that the field to be edited occupies *w* positions.

The *d* indicates that the fractional part of the number (the part to the right of the decimal point) has *d* digits. However, if the input datum contains a decimal point, that decimal point overrides the *d* value.

The specified input/output list item must be of type real. On input, the specified list item will become defined with a real datum. On output, the specified list item must be defined as a real datum.

The output field for the F *w.d* edit specifier has width *w*. The value is right-justified in that field. The field consists of zero or more leading blanks followed by either a minus if the value is negative, or an optional plus, followed by the magnitude of the value of the list item rounded to *d* decimal digits.

The *w* must allow for a minus sign, at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal point. Therefore it must be the case that $w \geq d+3$.

Example: Real input with F editing. The program `Finp.f`.

```
      CHARACTER LINE*24 / '12345678 23.5678 .345678' /
      READ( LINE, '( F8.3, F8.3, F8.3 )') R, S, T
      PRINT '( F9.3, F9.4, F9.6 )', R, S, T
      END
```

The above displays:

```
12345.678DD23.5678D0.345678
```

In the above example, the first input data item has no decimal point, so the `F8.3` determines the decimal point. The other input data items have decimal points, so those decimal points override the F edit descriptor as far as decimal points are concerned.

Example: Real Output with F editing. The program `Fout.f`.

```
      R = 1234.678
      PRINT 1, R, R, R
1     FORMAT( F9.3 / F8.4 / F13.4 )
      END
```

The above displays:

```
Δ1234.678
********
ΔΔΔΔ1234.6780
```

In the above example, the `F8.4` does not allow for the sign; the `F13.4` results in four leading blanks and one trailing zero.

G Editing

The G specifier is for decimal real data items. The general form is

```
G [ w [ .d ] ]
or
G w.d E e
```

The D, E, F, and G edit specifiers interpret data in the same way.

The representation for output by the G edit descriptor depends on the magnitude of the internal datum. In the following table, *N* is the magnitude of the internal datum.

Range	*Form*
0.1 ≤ *N* < 1.0	F(*w*-4).*d*, *n*(Δ)
1.0 ≤ *N* < 10.0	F(*w*-4).(*d*-1), *n*(Δ)
...	...
$10^{(d-2)} \le N \le 10^{(d-1)}$	F(*w*-4).1, *n*(Δ)
$10^{(d-1)} \le N < 10^{d}$	F(*w*-4).0, *n*(Δ)

Commas in Formatted Input

If you are entering numeric data that is controlled by a fixed-column format, then you can use commas to override any exacting column restrictions.

Example: Format.

```
      (I10, F20.10, I4)
```

Using the above format reads the record below correctly.

```
-345,.05e-3,12
```

The I/O system is just being more lenient than described in the FORTRAN Standard. In general, when doing a *formatted* read of *noncharacter* variables, commas override field lengths. More precisely, for I*w*, F*w.d*, E*w.d*[E*e*], and G*w.d* input fields, the field ends when *w* characters have been scanned or a comma has been scanned, whichever occurs first. If it's a comma, the field consists of the characters up to but not including the comma; the next field begins with the character following the comma.

*Remaining Characters (*Q*)*

The Q edit descriptor gets the *length* of an input record, or of the remaining portion of it that is unread. ♦

It gets the number of characters remaining to be read from the current record.

Example: From a real and a string, get: real, string length, and string.

```
demo$ cat qed1.f
* qed1.f Q edit descriptor (real & string)
      CHARACTER CVECT(80)*1
      OPEN ( UNIT=4, FILE='qed1.data' )
      READ ( 4, 1 ) R, L, ( CVECT(I), I=1,L )
1     FORMAT ( F4.2, Q, 80 A1 )
      WRITE ( *, 2 ) R, L, '"', (CVECT(I),I=1,L), '"'
2     FORMAT ( 1X, F7.2, 1X, I2, 1X, 80A1 )
      END
demo$ cat qed1.data
8.10qwerty
demo$ f77 qed1.f -o qed1
qed1.f:
 MAIN:
demo$ qed1
   8.10 6 "qwerty"
demo$ ■
```

The above example reads a field into the variable R, then reads the number of characters remaining after that field into L, then reads L characters into CVECT. Q as the *nth* edit descriptor matches with L as the *nth* element in the READ list.

Example: Get length of input record; put the Q descriptor first.

```
demo$ cat qed2.f
      CHARACTER CVECT(80)*1
      OPEN ( UNIT=4, FILE='qed2.data' )
      READ ( 4, 1 ) L, ( CVECT(I), I=1,L )
1     FORMAT ( Q, 80A1 )
      WRITE ( *, 2 ) L, '"', (CVECT(I),I=1,L), '"'
2     FORMAT ( 1X, I2, 1X, 80A1 )
      END
demo$ cat qed2.data
qwerty
demo$ f77 qed2.f -o qed2
qed2.f:
 MAIN:
demo$ qed2
  6 "qwerty"
demo$ ■
```

The above example gets the length of the input record. With the whole input string and its length, you can then parse it yourself.

Restrictions on the Q Edit Descriptor

- The list element it corresponds to must be of `INTEGER` or `LOGICAL` data type.
- `Q` does strictly a *character* count. It gets the number of *characters* remaining in the input record. It does not get the number of integers or reals or anything else.
- This operates on files and `stdin` (terminal) input.
- This is ignored for output.

*Scale Factor (*P*)*

The P edit descriptor lets you scale real *input* values by a power of 10. It also gives you more control over the significant digit displayed for *output* values.

The general form is:

[*k*] P	
k	Integer constant, with an optional sign

k is called the *scale factor*, and the default value is zero.

Examples: I/O statements with scale factors

```
READ ( 1, '( 3P E8.2 )' ) X
WRITE ( 1, '( 1P E8.2 )' ) X
```

P by itself is equivalent to 0P. It resets the scale factor to the default value 0P. This P by itself is nonstandard.

Scope

The scale factor is reset to zero at the start of execution of each I/O statement. The scale factor can have an effect on D, E, F, and G edit descriptors.

Input

On input, any external datum that does not have an exponent field will be divided by 10*k* before it is stored internally.

Input Examples: Showing data, scale factors, and resulting value stored

Data	18.63	18.63	18.63E2	18.63
Format	E8.2	3P E8.2	3P E8.2	-3P E8.2
Memory	18.63	.01863	18.63E2	18630.

Output

On output, with D, and E descriptors, and with G descriptors if the E editing is required, the internal item will get its basic real constant part multiplied by 10^k and the exponent will be reduced by *k* before it is written out.

On output with the F descriptor, and with G descriptors if the F editing is sufficient, the internal item will get its basic real constant part multiplied by 10^k before it is written out.

Output Examples: Showing value stored, scale factors, and resulting output

Memory	290.0	290.0	290.0	290.0
Format	2P E9.3	1P E9.3	-1P E9.3	F9.3
Display	29.00E+01	2.900E+02	0.029E+04	0.290E+03

*Sign Editing (*SU,SP,SS,S*)*

The SU, SP, and S edit descriptors control leading signs for output. For normal output, without any specific sign specifiers, if a value is negative, a minus sign is printed in the first position to the left of the leftmost digit; and if the value is positive, printing a plus sign depends on the implementation, but f77 omits the plus sign.

The following sign specifiers are available:

SP

If SP precedes a specification, a sign is printed.

SS

If SS precedes a specification, plus-sign printing is suppressed.

S

If S precedes a specification, system default is restored. The default is SS.

SU

If SU precedes a specification, integer values are interpreted as *unsigned.* This is nonstandard.

For example, the *unsigned* specifier can be used with the radix specifier to format a hexadecimal dump, as follows.

```
2000 FORMAT( SU, 16R, 8I10.8 )
```

Rules and Restrictions for Sign Control

- Sign-control specifiers apply to *output* only.
- A sign-control specifier remains in effect until another sign-control specifier is encountered, or format interpretation is complete.
- The S, SP, and SS specifiers affect only I, F, E, D, and G editing.
- The SU specifier affects only I editing.

Slash Editing (/)

The slash (/) edit specifier indicates the end of data transfer on the current record.

Sequential Access

Input— On input, any remaining portion of the current record is skipped, and the file is positioned at the beginning of the next record. Two successive slashes (//) skip a whole record.

Output— On output, an end-of-record is written, and a new record is started. Two successive slashes (//) produce a record of no characters. If the file is an *internal* file, that record is filled with blanks.

Direct Access

Input and Output — Each slash increases the record number by one, and the file is positioned at the start of the record with that record number.

Output — On output, two successive slashes (//) produce a record of no characters, and that record is filled with blanks.

Termination Control (:)

The colon (:) edit descriptor allows for conditional termination of the format. If the I/O list is exhausted before the format, then the format terminates at the colon.

Example: Termination control

```
* col1.f The colon (:) edit descriptor
      DATA INIT / 3 /, LAST / 8 /
      WRITE ( *, 2 ) INIT
      WRITE ( *, 2 ) INIT, LAST
2     FORMAT ( 1X 'INIT = ', I2, :, 3X, 'LAST = ', I2 )
      END
```

The above program produces output such as the following

```
      INIT = 3
      INIT = 3 LAST = 8
```

Without the colon, the output is more like this.

```
      INIT = 3 LAST =
      INIT = 3 LAST = 8
```

Runtime Formats

You can put the format specifier into an object that you can change during execution. This improves flexibility. There is some increase in execution time because this kind of format specifier gets parsed every time the I/O statement is executed. These are also called variable formats.

The object must be one of the following kinds:

- Character expression

  The character expression can be a scalar, an array, an element of an array, a substring, a field of a structured record ♦, the concatenation of any of the above, and so forth.

- Integer array ♦

  The integer array can get its character values by a DATA statement, an assignment statement, a READ statement, and so forth.

You must provide the delimiting left and right parentheses, but not the word FORMAT and not a statement number.

You must declare the object so that it is big enough to hold the entire format. For instance, '(8X,12I)' does not fit in an INTEGER*4 or a CHARACTER*4 object.

Examples. Runtime formats in character expressions and integer arrays.

```
demo$ cat runtim.f
      CHARACTER CS*8
      CHARACTER CA(1:7)*1 /'(','1','X',',','I','2',')'/
      CHARACTER S(1:7)*6
      INTEGER*4 IA(2)
      STRUCTURE / STR /
           CHARACTER*4 A
           INTEGER*4 K
      END STRUCTURE
      CHARACTER*8 LEFT, RIGHT
      RECORD /STR/ R
      N = 9
      CS = '(I8)'
      WRITE( *, CS ) N ! Character Scalar
      CA(2) = '6'
      WRITE( *, CA ) N ! Character Array
      S(2) = '(I8)'
      WRITE( *, S(2) ) N ! Element Of Character Array
      IA(1) = '(I8)'
      WRITE(*, IA ) N ! Integer Array
      R.A = '(I8)'
      WRITE( *, R.A ) N ! Field Of Record
      LEFT = '(I'
      RIGHT = '8)'
      WRITE(*, LEFT // RIGHT ) N ! Concatenate
      END
demo$ f77 -silent runtim.f
demo$ a.out
       9
       9
       9
       9
       9
       9
demo$ ■
```

Variable Format Expressions (<e>)

In general, inside a FORMAT statement, any integer constant can be replaced by an arbitrary expression.♦

The expression itself must be enclosed in angle brackets.

For example, the "6" in

```
1     FORMAT( 3F6.1 )
```

can be replaced by the variable "N", as in

```
1     FORMAT( 3F<N>.1 )
```

or by the slightly more complicated expression "2*N+M", as in

```
1     FORMAT( 3F<2*N+M>.1 )
```

Similarly, the "3" or "1" can be replaced by any expression.

The single exception is the "*n*" in an "*n*H…" edit descriptor.

Rules and Restrictions for Variable Format Expressions

- The expression is reevaluated each time it is encountered in a format scan.
- If necessary, the expression is converted to integer type.
- Any valid FORTRAN expression is allowed, including function calls.
- Variable expressions are not allowed in formats generated at runtime.
- The "*n*" in an "*n*H…" edit descriptor cannot be a variable expression.

5.5 Unformatted I/O

Unformatted I/O is used to transfer binary information to or from memory locations without changing its internal representation. Each execution of an *unformatted* I/O statement causes a single logical record to be read or written. Since internal representation varies with different architectures, unformatted I/O is limited in its portability.

You can use unformatted I/O to write data out temporarily, or to write data out quickly for subsequent input to another FORTRAN program running on a machine with the same architecture.

Sequential Access I/O

Logical record length for unformatted, sequential files is determined by the number of bytes required by the items in the I/O list. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size.

Example:

```
      WRITE( 8 ) A, B
```

The FORTRAN runtime system embeds the record boundaries in the data by inserting an `INTEGER*4` byte count at the beginning and end of each unformatted sequential record during an unformatted sequential `WRITE`. The trailing byte count enables `BACKSPACE` to operate on records. The result is that FORTRAN programs can use an unformatted sequential `READ` only on data that was written by an unformatted sequential `WRITE` operation. Any attempt to read such a record as formatted would have unpredictable results.

Guidelines:

- Avoid using the unformatted sequential `READ` unless your file was written that way.
- Because of the extra data at the beginning and end of each unformatted *sequential* record, you might want to try using the unformatted *direct* `I/O` whenever that extra data is significant. It is more significant with short records than with very long ones.

Direct Access I/O

If your I/O lists are different lengths, you can `OPEN` the file with the `"RECL=1"` option. This signals FORTRAN to use the I/O list to determine how many items to read or write.

For each read, you still must tell it the initial record to start at, in this case which byte, so you must know the size of each item. ♦

A simple example follows.

Example: Direct access, write 3 records, 2 integers each.

```
demo$ cat Direct1.f
      integer u/4/, v /5/, w /6/, x /7/, y /8/, z /9/
      open( 1, access='DIRECT', recl=8 )
      write( 1, rec=1 ) u, v
      write( 1, rec=2 ) w, x
      write( 1, rec=3 ) y, z
      end
demo$ f77 -silent Direct1.f
demo$ a.out
demo$ ■
```

Example: Direct access, read 3 records, 2 integers each.

If you know record length is *n*, then you can use the "`recl=`*n*" option.

Here you read it as it was written.

This is simpler, easier, better.

```
demo$ cat Direct2.f
      integer u, v, w, x, y, z
      open( 1, access='DIRECT', recl=8 )
      read( 1, rec=1 ) u, v
      read( 1, rec=2 ) w, x
      read( 1, rec=3 ) y, z
      write(*,*) u, v, w, x, y, z
      end
demo$ f77 -silent Direct2.f
demo$ a.out
  4  5  6  7  8  9
demo$ ■
```

Example: Direct-access read, variable-length records, `recl=1`.

If you know the size of each item, but not the record length, then you can use the "`recl=1`" option.

Here you can read it using different record lengths than it was written with.

This is trickier.

```
demo$ cat Direct3.f
      integer u, v, w, x, y, z
      open( 1, access='DIRECT', recl=1 )
      read( 1, rec=1 ) u, v, w
      read( 1, rec=13 ) x, y, z
      write(*,*) u, v, w, x, y, z
      end
demo$ f77 -silent Direct3.f
demo$ a.out
  4  5  6  7  8  9
demo$ ■
```

Above, after reading 3 integers (12 bytes) you start the next read at record 13.

5.6 List-Directed I/O

List-directed I/O is a *free-form* I/O for sequential access devices. To get it, use an asterisk as the format identifier, as in:

```
READ( 6, * ) A, B, C
```

Rules for List-Directed Input

- On input, values are separated by strings of blanks and (possibly) a comma.
- Values, except for character strings, cannot contain blanks.
- Character strings can be quoted strings, using pairs of quotes ("), or pairs of apostrophes (’), or unquoted strings (see “Unquoted Strings” on page 294), but *not* hollerith (*n*Hxyz) strings.
- End of record counts as a blank, except in character strings, where it is ignored.
- Complex constants are given as two real constants separated by a comma and enclosed in parentheses.
- A null input field, such as between two consecutive commas, means that the corresponding variable in the I/O list is not changed.
- Input data items can be preceded by repetition counts, as in the following.

```
4*(3.,2.) 2*, 4*’hello’
```

  The above input stands for 4 complex constants, 2 null input fields, and 4 string constants.

- A slash (/) in the input list terminates assignment of values to the input list during *list-directed* input, and the remainder of the current input line is skipped. Text following the slash is ignored and can be used to comment the data line.

Output Format

List-directed output provides a quick and easy way to print output without fussing with format details. If you need exact formats, use formatted I/O. A suitable format is chosen for each item, and where a conflict exists between complete accuracy and simple output form, the simple form is chosen.

Rules for List-Directed Output

- In general, each record starts with a blank space.
  For a *print* file, that blank is not printed. See "Print Files," for details.♦
- Character strings are printed as is. They are not enclosed in quotes, so only certain forms of strings can be read back using list-directed input. These forms are described in the next section.
- A number with no exact binary representation is rounded off.

  Example: No exact binary representation.

```
demo$ cat lis5.f
      READ ( 5, * ) X
      WRITE( 6, * ) X, '    beauty'
      WRITE( 6, 1 ) X
1     FORMAT( 1X, F13.8, ' truth' )
      END
demo$ f77 lis5.f
lis5.f:
 MAIN:
demo$ a.out
1.4
    1.40000000 beauty
    1.39999998 truth
demo$ ■
```

  Above, if you need accuracy, specify the format.

- Output lines longer than 80 characters are avoided where possible.
- Complex and double complex values include an appropriate comma.
- Real, double and quadruple precision values are formatted differently.
- A backslash-n (\ n) in a character string is output as a carriage return, unless the `–xl` option is on, and then it is output as a backslash-n.

Example: List-directed I/O and backslash-n, with and without `-xl`.

```
demo$ cat f77 bslash.f
      CHARACTER S*8 / '12\n3' /
      PRINT *, S
      END
demo$ ■
```

Without `-xl`, `\n` prints as a carriage return.

```
demo$ f77 -silent bslash.f
demo$ a.out
12
3
demo$ ■
```

With `-xl`, `\n` prints as a character string.

```
demo$ f77 -xl -silent bslash.f
demo$ a.out
12\n3
demo$ ■
```

Table 5-8 Default Formats for List-Directed Output

Type	***Format***
`BYTE`	Two blanks followed by the number
`CHARACTER`	`A(`*n*`+2)` {*n* = length of character expression}
`COMPLEX`	`'ΔΔ(', 1PE14.5E2, ',', 1PE14.5E2, ')'`
`COMPLEX*16`	`'ΔΔ(', 1PE22.13.E2, ',', 1PE22.13.E2, ')'`
`COMPLEX*32` (*SPARC only*)	`'ΔΔ(', 1PE44.34E3, ',', 1PE44.34E3, ')'`
`INTEGER*2`	Two blanks followed by the number
`INTEGER*4`	Two blanks followed by the number
`LOGICAL*1`	Two blanks followed by the number
`LOGICAL*2`	`L3`
`LOGICAL*4`	`L3`
`REAL`	`1PE14.5E2`
`REAL*8`	`1PE22.13.E2`
`REAL*16` (*SPARC only*)	`1PE44.34E4`

Unquoted Strings

f77 list-directed I/O allows reading of a string not enclosed in quotes.♦

The string must not start with a digit, and cannot contain separators (commas or slashes (/)) or whitespace (spaces or tabs). A newline terminates the string unless escaped with a backslash (\ ). Any string not meeting the above restrictions must be enclosed in single or double quotes.

Example: List-directed input of unquoted strings.

```
      CHARACTER C*6, S*8
      READ *, I, C, N, S
      PRINT *, I, C, N, S
      END
```

The above program, unquoted.f, reads and displays as follows.

```
demo$ a.out
23 label 82 locked
  23label 82locked
demo$ ■
```

Internal I/O

f77 extends list-directed I/O to allow internal I/O.♦

During internal, list-directed reads, characters are consumed until the input list is satisfied or the end-of-file is reached. During internal, list-directed writes, records are filled until the output list is satisfied. The length of an internal array element should be at least 20 characters to avoid logical record overflow when writing double-precision values. Internal, list-directed read was implemented to make command line decoding easier. Internal, list-directed output should be avoided.

5.7 NAMELIST I/O

NAMELIST I/O lets you do format-free input or output of whole groups of variables, or input of selected items in a group of variables. ♦

The NAMELIST statement defines a group of variables or arrays. It specifies a *group-name*, and it lists the variables and arrays of that group.

The syntax of the NAMELIST statement is

NAMELIST */group-name/namelist*[[,]*/group-name/namelist*]…	
group-name	Identifier
namelist	List of variables or arrays, separated by commas

Example: NAMELIST statement.

```
      CHARACTER*18 SAMPLE
      LOGICAL*4 NEW
      REAL*4 DELTA
      NAMELIST /CASE/ SAMPLE, NEW, DELTA
```

Restrictions

- The group name can appear in only the NAMELIST, READ, or WRITE statements, and must be unique for the program.
- The list cannot include any constants, dummy arguments, array elements, structures, substrings, records, record fields, pointers, or pointer-based variables.
- The *input data* can include array elements, strings, and substrings in the sense that the input constant data string can be shorter than the declared size of the variable.
- A variable or array can be listed in more than one NAMELIST group.

Example: A variable in two NAMELIST groups.

```
      REAL ARRAY(4,4)
      CHARACTER*18 SAMPLE
      LOGICAL*4 NEW
      REAL*4 DELTA
      NAMELIST /CASE/ SAMPLE, NEW, DELTA
      NAMELIST /GRID/ ARRAY, DELTA
```

In the above example, DELTA is in the group CASE, and in the group GRID.

NAMELIST *Output*

NAMELIST output uses a special form of WRITE statement. This makes a report showing the group name, and for each variable of the group, it shows the name and current value in memory. It formats each value according to the type of each variable, and it writes the report so that NAMELIST input can read it.

The syntax of NAMELIST WRITE is:

```
WRITE ( extu,  namelist-specifier [, iostat] [, err])
```

where *namelist-specifier* has the form

```
[NML=]group-name
```

and *group-name* has been previously defined in a NAMELIST statement.

The NAMELIST WRITE statement writes values of all variables in the group, in the same order as in the NAMELIST statement.

Example: NAMELIST output.

```
demo$ cat nam1.f
* nam1.f Namelist output
      CHARACTER*8 SAMPLE
      LOGICAL*4 NEW
      REAL*4 DELTA
      NAMELIST /CASE/ SAMPLE, NEW, DELTA
      DATA SAMPLE /'Demo'/, NEW /.TRUE./, DELTA /0.1/
      WRITE ( *, CASE )
      END
demo$ f77 nam1.f
f77 nam1.f
nam1.f:
 MAIN:
demo$ a.out
Δ&case sample= Demo , new= T, delta= 0.100000
Δ&end
demo$ ■
```

↑ column 2

Note that if you do omit the keyword NML then the unit parameter must be first, *namelist-specifier* must be second, and there must *not* be a format specifier.

Alternate — The WRITE can have the form of the following example.

```
      WRITE ( UNIT=6, NML=CASE )
```

NAMELIST *Input*

The NAMELIST input statement reads the next external record, skipping over column one, and looking for the symbol "$" in column two or beyond, followed by the group name specified in the READ statement. The records are input and values assigned by matching names in the data with names in the group, using the data types of the variables in the group. Variables in the group that are not found in the input data are unaltered.

The syntax of NAMELIST READ is:

```
READ ( extu, namelist-specifier [, iostat] [, err] [, end])
```

where *namelist-specifier* has the form

```
[NML=]group-name
```

and *group-name* has been previously defined in a NAMELIST statement.

Example: NAMELIST input.

```
      CHARACTER*14 SAMPLE
      LOGICAL*4 NEW
      REAL*4 DELTA, MAT(2,2)
      NAMELIST /CASE/ SAMPLE, NEW, DELTA, MAT
      READ ( 1, CASE )
```

In this example, the group CASE consists of the three variables SAMPLE, NEW, DELTA, and MAT. If you do omit the keyword NML, then you must also omit the keyword UNIT; and the unit parameter must be first, *namelist-specifier* must be second, and there must *not* be a format specifier.

Alternate — The READ can have the form of the following example.

```
      READ ( UNIT=1, NML=CASE )
```

NAMELIST *Data*

The first record of NAMELIST input data has the special symbol "$" (dollar sign) in column two or beyond, followed by the NAMELIST group name. This is followed by a series of assignment statements, starting in or after column two, on the same or subsequent records, each assigning a value to a variable (or one or more values to array elements) of the specified group. The input data is terminated with another "$" in or after column two, as in the pattern

```
Δ$group-name variable=value [,variable=value,…] $[END]
```

You can alternatively use an ampersand (&) in place of each dollar sign, but the beginning and ending delimiters must match. The END is an optional part of the last delimiter.

The input data assignment statements must be in one of the following forms:

```
variable=value

array=value1[, value2,]…

array(subscript)=value1[, value2,]…

array(subscript,subscript)=value1[, value2,]…

variable=character constant

variable(index:index)=character constant
```

If an array is subscripted, it must be subscripted with the appropriate number of subscripts: 1, 2, 3,…

Use quotes (either " or ') to delimit character constants. For more on character constants, see "Syntax Rules for NAMELIST Data" below.

The following is sample data to be read by the program segment above.

```
Δ$case delta=0.05, mat( 2, 2 ) = 2.2, sample='Demo' $
```

↑ column 2

The data could be on several records.

Here NEW was not input, and the order is not the same as in the example NAMELIST statement.

```
Δ$case
Δdelta=0.05
Δmat( 2, 2 ) = 2.2
Δsample='Demo'
Δ$
```

↑ column 2

Syntax Rules for NAMELIST *Data*

The following syntax rules apply for input data to be read by NAMELIST.

- The variables of the named group can be in any order, and any can be omitted.
- The data must start in or after column two. Column one is totally ignored.
- There must be at least one comma, space, or tab between variables, and one or more spaces or tabs are the same as a single space. Consecutive commas are not permitted before a variable name. Spaces before or after a comma have no effect.
- No spaces or tabs are allowed inside a *group* name or a *variable* name, except around the commas of a subscript, around the colon of a substring, and after the "("and before the ")" marks. No name can be split over two records.
- The end of a record acts like a space character.

  Exception: In a character constant, it is ignored, and the character constant is continued with the next record. The last character of the current record is immediately followed by the second character of the next record. The first character of each record is ignored.
- The equal sign of the assignment statement can have zero or more blanks or tabs on each side of it.
- Only *constant* values can be used for subscripts, range indicators of substrings, and the values assigned to variables or arrays. You cannot use a symbolic constant (parameter) in the actual input data.

  Hollerith, octal, and hexadecimal constants are not permitted.

  Each constant assigned has the same form as the corresponding FORTRAN constant.

There must be at least one comma, space, or tab between constants, and zero or more spaces or tabs are the same as a single space. You can enter `1,2,3` or `1 2 3` or `1, 2, 3` and so forth.

Inside a character constant, consecutive spaces or tabs are preserved, not compressed.

A character constant is delimited by apostrophes (') or quotes ("), but if you start with one of those, you must finish that character constant with the same one. If you use the apostrophe as the delimiter, then to get an apostrophe in a string, use two consecutive apostrophes.

Example: Character constants.

```
Δsample='use "$" in 2' {goes in as: use "$" in 2 }
Δsample='don''t' {goes in as: don't }
Δsample="don''t" {goes in as: don''t }
Δsample="don't" {goes in as: don't }
```

A complex constant is a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur only around the punctuation.

A logical constant is any form of true or false value, such as `.TRUE.` or `.FALSE.`, or any value beginning with `.T, .F`, etc.

A null data item is denoted by two consecutive commas, and it means the corresponding array element or complex variable value is not to be changed. Null data item can be used with array elements or complex variables only. One null data item represents an entire complex constant; you cannot use it for either part of a complex constant.

Example: NAMELIST input with some null data, the program is as follows.

```
* nam2.f Namelist input with consecutive commas
      REAL ARRAY(4,4)
      NAMELIST /GRID/ ARRAY
      WRITE ( *, * ) 'Input?'
      READ ( *, GRID )
      WRITE ( *, GRID )
      END
```

The data for nam2.f is as follows.

```
Δ$GRID ARRAY = 9,9,9,9,,,,,8,8,8,8 $
```

↑ column 2 ↑ 5 consecutive commas

This loads 9s into row 1, skips 4 elements, and loads 8s into row 3 of ARRAY.

Arrays Only

The forms "*r***c*" and "*r**" can be used only with an array.

- The form" *r***c*" stores *r* copies of the constant *c* into an array, where *r* is a nonzero, unsigned integer constant, and *c* is any constant.

  Example: NAMELIST with repeat-factor in data, the program is as follows.

```
* nam3.f Namelist "r*c" and "r* "
      REAL PSI(10)
      NAMELIST /GRID/ PSI
      WRITE ( *, * ) 'Input?'
      READ ( *, GRID )
      WRITE ( *, GRID )
      END
```

The input for nam3.f is as follows.

```
Δ$GRID PSI = 5*980 $
```

↑ column 2

The above program, nam3.f, reads the above input and loads 980.0 into the first 5 elements of the array PSI.

- The form *r** skips *r* elements of an array (that is, does *not* change them) where *r* is an unsigned integer constant.

Example: NAMELIST input with some skipped data.

Other input.

```
Δ$GRID PSI = 3* 5*980 $
```

↑ column 2

The program, nam3.f, with the above input, skips the first 3 elements and loads 980.0 into elements 4,5,6,7,8 of PSI.

Requesting Names

If your program is doing NAMELIST input from the terminal, you can request the group name and NAMELIST names that it will accept. To do this, enter a question mark (?) in column *two*, and press RETURN. The group name and variable names for that group will be displayed, and then it will wait again for input.

Example: Requesting names.

```
demo$ cat nam4.f
* nam4.f Namelist: requesting names
      CHARACTER*14 SAMPLE
      LOGICAL*4 NEW
      REAL*4 DELTA
      NAMELIST /CASE/ SAMPLE, NEW, DELTA
      WRITE ( *, * ) 'Input?'
      READ ( *, CASE )
      END
demo$ f77 -silent nam4.f
demo$ a.out
  Input?
Δ?
Δ$case
Δsample
Δnew
Δdelta
D

Δ$case sample="Test 2", delta=0.03 $
demo$ ■
```

User input 1 →

User input 2 →

↑ column 2

5

Intrinsic Functions 6

This chapter is organized into the following sections.

6.1 Arithmetic and Mathematical Functions

Arithmetic

Table 6-1 Arithmetic Functions

Intrinsic Function	***Definition***	***No. of Args***	***Generic Name***	***Specific Name***	***Type of Argument***	***Type of Function***
Truncation	int(a) Read Note 1	1	AINT	AINT DINT QINT ♦	Real Double Real*16	Real Double Real*16
Nearest Whole Number	int(a+.5) if a ≥ 0 int(a-.5) if a < 0	1	ANINT	ANINT DNINT QNINT ♦	Real Double Real*16	Real Double Real*16
Nearest Integer	int(a+.5) if a ≥ 0 int(a-.5) if a < 0	1	NINT	NINT IDNINT IQNINT ♦	Real Double Real*16	Integer Integer Integer

The REAL*16 and COMPLEX*32 are ***SPARC only***

Table 6-2 More Arithmetic Functions

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of Argument*	*Type of Function*
Absolute Value	|a|	1	ABS	IABS	Integer	Integer
				ABS	Real	Real
	Read Note 6.			DABS	Double	Double
				CABS	Complex	Real
	$(ar^2 + ai^2)^{**}(1/2)$			CQABS ♦	Complex*32	Real*16
				QABS ♦	Real*16	Real*16
				ZABS ♦	Complex*16	Double
				CDABS ♦	Complex*16	Double
Remainder	a1-int(a1/a2)*a2	2	MOD	MOD	Integer	Integer
	Read Note 1			AMOD	Real	Real
				DMOD	Double	Double
				QMOD ♦	Real*16	Real*16
Transfer of Sign	|a1| if a2 ≥ 0	2	SIGN	ISIGN	Integer	Integer
	-|a1| if a2 < 0			SIGN	Real	Real
				DSIGN	Double	Double
				QSIGN ♦	Real*16	Real*16
Positive Difference	a1-a2 if a1 > a2	2	DIM	IDIM	Integer	Integer
	0 if a1 ≤ a2			DIM	Real	Real
				DDIM	Double	Double
				QDIM ♦	Real*16	Real*16
Double & Quad Products	a1 * a2	2		DPROD	Real	Double
				QPROD ♦	Double	Real*16
Choosing Largest Value	max(a1, a2, ...)	2	MAX	MAX0	Integer	Integer
				AMAX1	Real	Real
				DMAX1	Double	Double
				QMAX1 ♦	Real*16	Real*16
				AMAX0	Integer	Real
				MAX1	Real	Integer
Choosing Smallest Value	min(a1, a2, ...)	2	MIN	MIN0	Integer	Integer
				AMIN1	Real	Real
				DMIN1	Double	Double
				QMIN1 ♦	Real*16	Real*16
				AMIN0	Integer	Real
				MIN1	Real	Integer

Type Conversion

Table 6-3 Type Conversion Functions

Conversion to	***No. of Args***	***Generic Name***	***Specific Name***	***Type of Argument***	***Type of Function***
Integer Read Note 1.	1	INT	-	Integer	Integer
			INT	Real	Integer
			IFIX	Real	Integer
			IDINT	Double	Integer
			-	Real*16	Integer
			-	Complex	Integer
			-	Complex*16	Integer
			-	Complex*32	Integer
			IQINT ♦	Real*16	Integer
Real Read Note 2.	1	REAL	REAL	Integer	Real
			FLOAT	Integer	Real
			-	Real	Real
			SNGL	Double	Real
			-	Real*16	Real
			-	Complex	Real
			-	Complex*16	Real
			-	Complex*32	Real
			SNGLQ ♦	Real*16	Real
				Double	Real
				Complex	Real
				Complex*16	Real
				Complex*32	Real
Double Read Note 3.	1	DBLE	DBLE	Integer	Double Precision
			DFLOAT	Integer	Double Precision
			DREAL ♦	Real	Double Precision
			DBLEQ	Real*16	Double Precision
				Double	Double Precision
				Complex	Double Precision
				Complex*16	Double Precision
				Complex*32	Double Precision
Real*16	1	QREAL	QREAL ♦	Integer	Real*16
		QEXT	QFLOAT ♦	Integer	Real*16
			QEXT ♦	Integer	Real*16
			QEXTD ♦	Double	Real*16

Table 6-3 Type Conversion Functions *(Continued)*

Conversion to	***No. of Args***	***Generic Name***	***Specific Name***	***Type of Argument***	***Type of Function***
Complex	1 or 2	CMPLX	-	Integer	Complex
			-	Real	Complex
			-	Double	Complex
			-	Real*16	Complex
			-	Complex	Complex
			-	Complex*16	Complex
			-	Complex*32	Complex
Complex*16	1 or 2	DCMPLX	-	Integer	Double Complex
			-	Real	Double Comlex
			-	Double	Double Complex
			-	Real*16	Double Complex
			-	Complex	Double Complex
			-	Complex*16	Double Complex
			-	Complex*32	Double Complex
			-	Real*16	Double Complex
Complex*32	1 or 2	QCMPLX	-	Integer	Complex*32
			-	Real	Complex*32
			-	Double	Complex*32
			-	Real*16	Complex*32
			-	Complex	Complex*32
			-	Complex*16	Complex*32
			-	Complex*32	Complex*32
Integer Read Note 5	1		ICHAR IACHAR ♦	Character	Integer
Character Read Note 5	1		CHAR ACHAR ♦	Integer	Character

On an ASCII machine (including Sun systems):

- ACHAR is a nonstandard synonym for CHAR
- IACHAR is a nonstandard synonym for ICHAR

On a non-ASCII machine, ACHAR and IACHAR were intended to provide a way to deal directly with ASCII.

Trigonometric

Table 6-4 Trigonometric Functions

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of Argument*	*Type of Function*
Sine	sin(a)	1	SIN	SIN	Real	Real
				DSIN	Double	Double
				QSIN	Real*16	Real*16
				CSIN	Complex	Complex
				ZSIN ♦	Complex*16	Complex*16
				CDSIN ♦	Complex*16	Complex*16
				CQSIN ♦	Complex*32	Complex*32
Sine (degrees)	sin(a)	1	SIND ♦	SIND ♦	Real	Real
				DSIND ♦	Double	Double
				QSIND ♦	Real*16	Real*16
Cosine	cos(a)	1	COS	COS	Real	Real
				DCOS	Double	Double
				QCOS	Real*16	Real*16
				CCOS	Complex	Complex
				ZCOS ♦	Complex*16	Complex*16
				CDCOS ♦	Complex*16	Complex*16
				CQCOS ♦	Complex*32	Complex*32
Cosine (degrees)	cos(a)	1	COSD ♦	COSD ♦	Real	Real
				DCOSD ♦	Double	Double
				QCOSD ♦	Real*16	Real*16
Tangent	tan(a)	1	TAN	TAN	Real	Real
				DTAN	Double	Double
				QTAN ♦	Real*16	Real*16
Tangent (degrees)	tan(a)	1	TAND ♦	TAND ♦	Real	Real
				DTAND ♦	Double	Double
				QTAND ♦	Real*16	Real*16
Arcsine	arcsin(a)	1	ASIN	ASIN	Real	Real
				DASIN	Double	Double
				QASIN ♦	Real*16	Real*16
Arcsine (degrees)	arcsin(a)	1	ASIND ♦	ASIND ♦	Real	Real
				DASIND ♦	Double	Double
				QASIND ♦	Real*16	Real*16

Table 6-4 Trigonometric Functions *(Continued)*

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Arccosine	arccos(a)	1	ACOS	ACOS	Real	Real
				DACOS	Double	Double
				QACOS ♦	Real*16	Real*16
Arccosine (degrees)	arccos(a)	1	ACOSD ♦	ACOSD ♦	Real	Real
				DACOSD ♦	Double	Double
				QACOSD ♦	Real*16	Real*16
Arctangent	arctan(a)	1	ATAN	ATAN	Real	Real
				DATAN	Double	Double
				QATAN ♦	Real*16	Real*16
	arctan(a1/a2)	2	ATAN2	ATAN2	Real	Real
				DATAN2	Double	Double
				QATAN2 ♦	Real*16	Real*16
Arctangent (degrees)	arctan(a)	1	ATAND ♦	ATAND ♦	Real	Real
				DATAND ♦	Double	Double
				QATAND ♦	Real*16	Real*16
	arctan(a1/a2)	2	ATAN2D ♦	ATAN2D ♦	Real	Real
				DATAN2D ♦	Double	Double
				QATAN2D ♦	Real*16	Real*16
Hyperbolic Sine	sinh(a)	1	SINH	SINH	Real	Real
				DSINH	Double	Double
				QSINH ♦	Real*16	Real*16
Hyperbolic Cosine	cosh(a)	1	COSH	COSH	Real	Real
				DCOSH	Double	Double
				QCOSH ♦	Real*16	Real*16
Hyperbolic Tangent	tanh(a)	1	TANH	TANH	Real	Real
				DTANH	Double	Double
				QTANH ♦	Real*16	Real*16

The REAL*16 and COMPLEX*32 are *SPARC only.*

Other Mathematical Functions

Table 6-5 Other Mathematical Functions

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of Argument*	*Function*
Imaginary Part of a Complex	ai Read Note 6.	1	IMAG	AIMAG DIMAG ♦ QIMAG ♦	Complex Complex*16 Complex*32	Real Double Complex*32
Conjugate of a Complex	(ar, -ai) Read Note 6.	1	CONJ	CONJG DCONJG ♦ QCONJG ♦	Complex Complex*16 Complex*32	Complex Complex*16 Complex*32
Square Root	a**(1/2)	1	SQRT	SQRT DSQRT QSQRT CSQRT ZSQRT ♦ CDSQRT ♦ CQSQRT ♦	Real Double Real*16 Complex Complex*16 Complex*16 Complex*32	Real Double Real*16 Complex Complex*16 Complex*16 Complex*32
Exponential	e**a	1	EXP	EXP DEXP QEXP ♦ CEXP ZEXP ♦ CDEXP ♦ CQEXP ♦	Real Double Real*16 Complex Complex*16 Complex*16 Complex*32	Real Double Real*16 Complex Complex*16 Complex*16 Complex*32
Natural Logarithm	log(a)	1	LOG	ALOG DLOG QLOG ♦ CLOG ZLOG ♦ CDLOG ♦ CQLOG ♦	Real Double Real*16 Complex Complex*16 Complex*16 Complex*32	Real Double Real*16 Complex Complex*16 Complex*16 Complex*32
Common Logarithm	log10(a)	1	LOG10	ALOG10 DLOG10 QLOG10 ♦	Real Double Real*16	Real Double Real*16

The REAL*16 and COMPLEX*32 are *SPARC only.*

Table 6-6 Functions for Returning IEEE Values

IEEE Value	*Single Precision*	*Double Precision*	*Quadruple Precision*
infinity	`r_infinity()` ♦	`d_infinity()` ♦	`q_infinity()` ♦
quiet NaN	`r_quiet_nan()` ♦	`d_quiet_nan()` ♦	`q_quiet_nan()` ♦
signaling NaN	`r_signaling_nan()` ♦	`d_signaling_nan()` ♦	`q_signaling_nan()` ♦
min_normal	`r_min_normal()` ♦	`d_min_normal()` ♦	`q_min_normal()` ♦
min_subnormal	`r_min_subnormal()` ♦	`d_min_subnormal()` ♦	`q_min_subnormal()` ♦
max_subnormal	`r_max_subnormal()` ♦	`d_max_subnormal()` ♦	`qmax_subnormal()` ♦
max_normal	`r_max_normal()` ♦	`d_max_normal()` ♦	`q_max_normal()` ♦

Table 6-7 Other IEEE-Related Functions

Name	*Purpose*	*Call*	*Return Type*
ieee_flags ♦	Rounding and exception status	`i=ieee_flags(act,mode,in,out)`	Integer
ieee_handler ♦	Exception handler	`i=ieee_handler(act,except,hand)`	Integer
ieee_values ♦	Infinity, NaN, etc.	`x=r_infinity(), etc.`	Varies
ieee_retrospective ♦	Report exceptions	Called automatically before exit	None
standard_arithmetic ♦	Enable standard mode	`call standard_arithmetic()`	None
nonstandard_arithmetic ♦	Enable nonstandard mode	`call nonstandard_arithmetic()`	None

For more on standard or nonstandard mode, see `standard_arithmetic`(3M) or `nonstandard_arithmetic`(3M).

For more math functions, see also `libm_double`(3f), `libm_single`(3f), and `ieee_values`(3m).

For more on IEEE or floating-point usage, refer to the *FORTRAN User's Guide* or the *Numerical Computation Guide*.

6.2 Character Functions

Table 6-8 Character Functions

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of Argument*	*Type of Function*
Conversion *Read Note 5.*	Conversion to Character	1		CHAR ACHAR ♦	Integer	Character
	Conversion to Integer	1		ICHAR IACHAR ♦	Character	Integer
Index of a Substring	Location of Substring a2 in String a1 Read Note 10.	2		INDEX	Character	Integer
Length	Length of Character Entity Read Note 11.	1		LEN	Character	Integer
Lexically Greater Than or Equal	a1 ≥ a2 Read Note 12.	2		LGE	Character	Logical
Lexically Greater Than	a1 > a2 Read Note 12.	2		LGT	Character	Logical
Lexically Less Than or Equal	a1 ≤ a2 Read Note 12.	2		LLE	Character	Logical
Lexically Less Than	a1 < a2 Read Note 12.	2		LLT	Character	Logical

On an ASCII machine (including Sun systems):

- ACHAR is a nonstandard synonym for CHAR
- IACHAR is a nonstandard synonym for ICHAR

On a non-ASCII machine, ACHAR and IACHAR were intended to provide a way to deal directly with ASCII.

6.3 Miscellaneous Functions

Bit Manipulation

Table 6-9 Bitwise Functions

Bitwise Operations *Read Note 13.*	***No. of Args***	***Specific Name***	***Type of Argument***	***Type of Function***
Complement	1	NOT ♦	Integer	Integer
And	2	AND ♦	Integer	Integer
Inclusive Or	2	OR ♦	Integer	Integer
Exclusive Or	2	XOR ♦	Integer	Integer
Shift *Read Note 14.*	2	ISHFT ♦	Integer	Integer
Left Shift *Read Note 14.*	2	LSHIFT ♦	Integer	Integer
Right Shift *Read Note 14.*	2	RSHIFT ♦	Integer	Integer
Logical Right Shift *Read Note 14.*	2	LRSHFT ♦	Integer	Integer
Bit Extraction	3	IBITS ♦	Integer	Integer
Bit Set	2	IBSET ♦	Integer	Integer
Bit Test	2	BTEST ♦	Integer	Logical
Bit Clear	2	IBCLR ♦	Integer	Integer
Circular Shift	3	ISHFTC ♦	Integer	Integer

Environment

Table 6-10 Environmental Inquiry Functions

Definition	***No. of Args***	***Generic Name***	***Specific Name***	***Type of Argument***	***Function***
Base of Number System	1	EPBASE ♦	-	Integer	Integer
				Real	Integer
				Double	Integer
				Real*16	Integer
Number of Significant Bits	1	EPPREC ♦	-	Integer	Integer
				Real	Integer
				Double	Integer
				Real*16	Integer
Minimum Exponent	1	EPEMIN ♦	-	Real	Integer
				Double	Integer
				Real*16	Integer
Maximum Exponent	1	EPEMAX ♦	-	Real	Integer
				Double	Integer
				Real*16	Integer
Least Nonzero Number	1	EPTINY ♦	-	Real	Real
				Double	Double
				Real*16	Real*16
Largest Number Representable	1	EPHUGE ♦	-	Integer	Integer
				Real	Real
				Double	Double
				Real*16	Real*16
Epsilon *Read Note 16.*	1	EPMRSP ♦	-	Real	Real
				Double	Double
				Real*16	Real*16

Memory

Table 6-11 Memory Allocation and Deallocation Functions

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of Argument*	*Function*
Location	Address-of *(Read Note 17.)*	1		LOC ♦	Any	Integer
Allocate	Allocate memory and return the address *(Read Note 17.)*	1		MALLOC ♦	Integer	Integer
Deallocate	Deallocate memory allocated by MALLOC	1		FREE ♦	Any	None. This is a subroutine.

Remarks for Intrinsic Function Tables

The following remarks apply to all of the intrinsic function tables.

- The abbreviation “Double” stands for *Double Precision*.
- The abbreviation “DComplex” stands for *Double Complex*.
- An intrinsic that takes an INTEGER argument accepts either INTEGER*2 or INTEGER*4.
- An intrinsic that returns an INTEGER value returns the prevailing INTEGER type: if no -i2, then INTEGER*4; if -i2, then INTEGER*2.

  The exceptions are LOC and MALLOC, which always return an INTEGER*4.
- *(SPARC only)* An intrinsic that returns a REAL value returns the prevailing REAL type:
  if no -r8, then REAL*4; if -r8, then REAL*8.
- *(SPARC only)* An intrinsic that returns a DOUBLE PRECISION value returns the prevailing DOUBLE PRECISION type: if no -r8 then REAL*8; if -r8 then REAL*16.
- *(SPARC only)* An intrinsic that returns a COMPLEX value returns the prevailing COMPLEX type: if no -r8, then COMPLEX*8; if -r8, then COMPLEX*16.
- *(SPARC only)* An intrinsic that returns a DOUBLE COMPLEX value returns the prevailing DOUBLE COMPLEX type: if no -r8, then COMPLEX*16; if -r8, then COMPLEX*32.

- A function with a *generic* name returns a value with the same type as the argument — except for type conversion functions, the nearest integer function, and absolute value of a complex argument. If there is more than one argument, they must all be of the same type.
- If a function name is used as an *actual* argument, then it must be a *specific* name.
- If a function name is used as a *dummy* argument, then it does not identify an intrinsic function in the subprogram, and it has a data type according to the same rules as for variables and arrays.

Notes on Functions

Tables and notes 1 through 12 are based on the "Table of Intrinsic Functions," from *ANSI X3.9-1978 Programming Language FORTRAN*, with the SPARCompiler FORTRAN extensions added.

(1) `INT`

If `A` is type integer, then `INT(A)` is `A`.
If `A` is type real or double precision then:

if `|A| < 1`, then `INT(A)` is 0
if `|A| ≥ 1`, then

`INT(A)` is the greatest integer that does not exceed the magnitude of `A`, and whose sign is the same as the sign of `A`. (Such a mathematical integer value may be too large to fit in the computer integer type.)

If `A` is type complex or double complex then

apply the above rule to the real part of `A`.

If `A` is type real, then `IFIX(A)` is the same as `INT(A)`.

(2) REAL

If A is type real, then REAL(A) is A.

If A is type integer or double precision, then

REAL(A) is as much precision of the significant part of A as a real datum can contain.

If A is type complex, then REAL(A) is the real part of A.

If A is type double complex, then

REAL(A) is as much precision of the significant part of the real part of A as a real datum can contain.

(3) DBLE

If A is type double precision, then DBLE(A) is A.

If A is type integer or real, then DBLE(A) is

as much precision of the significant part of A as a double precision datum can contain.

If A is type complex, then DBLE(A) is

as much precision of the significant part of the real part of A as a double precision datum can contain.

If A is type Complex*16, then DBLE(A) is the real part of A.

(3′) QREAL

If A is type REAL*16, then QREAL(A) is A.

If A is type integer, real, or double precision then QREAL(A) is

as much precision of the significant part of A as a REAL*16 datum can contain.

If A is type complex or double complex, then QREAL(A) is

as much precision of the significant part of the real part of A as a REAL*16 datum can contain.

If A is type Complex*16, then QREAL(A) is the real part of A.

(4) CMPLX

If A is type complex, then CMPLX(A) is A.

If A is type integer, real, or double precision, then

CMPLX(A) is REAL(A) + 0i.

If A1 and A2 are type integer, real, or double precision, then

CMPLX(A1,A2) is REAL(A1) + REAL(A2)*i

If A is type double complex, then

CMPLX(A) is REAL(DBLE(A)) + i*REAL(DIMAG(A)).

If CMPLX has two arguments, then

they must be of the same type, and
they may be one of integer, real, or double precision.

If CMPLX has one argument, then

it may be one of integer, real, double precision, complex, or Complex*16.

(4') DCMPLX

If A is type Complex*16, then DCMPLX(A) is A.

If A is type integer, real, or double precision, then

DCMPLX(A) is DBLE(A) + 0i.

If A1 and A2 are type integer, real, or double precision, then

DCMPLX(A1,A2) is DBLE(A1) + DBLE(A2)*i.

If DCMPLX has two arguments, then

they must be of the same type, and

they may be one of integer, real, or double precision.

If DCMPLX has one argument, then

it may be one of integer, real, double precision, complex, or Complex*16.

(5) ICHAR

ICHAR(A) is the position of A in the collating sequence.

The first position is 0, the last is N-1, 0≤ICHAR(A)≤N-1, where N is the number of characters in the collating sequence, and A is of type character of length one.

CHAR and ICHAR are inverses in the following sense:

ICHAR(CHAR(I)) = I, for 0≤I≤N-1

CHAR(ICHAR(C)) = C, for any character C capable of representation in the processor.

(6) Complex

A Complex value is expressed as an ordered pair of reals, (ar, ai), where ar is the real part and ai is the imaginary part.

(7) Radians

All angles are expressed in radians, unless the "*Intrinsic Function*" column includes the "*(degrees)*" remark.

(8) Complex Function

The result of a function of type complex is the principal value.

(9) Argument types

All arguments in an intrinsic function reference must be of the same type.

(10) INDEX

INDEX(X,Y) is the place in X where Y starts. That is, it is the starting position within character string X of the first occurrence of character string Y

If Y does not occur in X, then INDEX(X,Y) is 0.

If LEN(X) < LEN(Y), then INDEX(X,Y) is 0.

(11) Argument to LEN

The value of the argument of the LEN function need not be defined at the time the function reference is executed.

(12) Lexical Compare

LGE(X, Y) is true if X=Y or if X follows Y in the collating sequence; otherwise it is false.

LGT(X, Y) is true if X follows Y in the collating sequence; otherwise it is false.

LLE(X, Y) is true if X=Y or if X precedes Y in the collating sequence; otherwise it is false.

LLT(X, Y) is true if X precedes Y in the collating sequence; otherwise it is false.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks.

(13) Bit Functions

See Chapter 9, "VMS Language Extensions," for details on other bitwise operations. ♦

(14) Shift

LSHIFT shifts *a1* logically *left* by *a2* bits (inline code).
LRSHFT shifts *a1* logically *right* by *a2* bits (inline code).
RSHIFT shifts *a1* arithmetically *right* by *a2* bits.
ISHFT shifts *a1* logically *left* if *a2* > 0 and *right* if *a2* < 0.

The LSHIFT and RSHIFT functions are the FORTRAN analogs of C's "<<" and ">>" operators. As in C, the semantics depend on the hardware.

(15) Environmental inquiries

Only the type of the argument is significant.

(16) Epsilon

Epsilon is the least e such that 1.0 + e ≠ 1.0.

(17) LOC and MALLOC

The LOC function returns the 32-bit address of a variable or of an external procedure. The function call MALLOC(*n*) allocates a block of at least *n* bytes, and returns the 32-bit address of that block.

6.4 VMS Intrinsic Functions

This section lists VMS FORTRAN intrinsic routines recognized by `f77`. They are, of course, nonstandard. ◆

Double-Precision Complex

Table 6-12 Double-Precision Complex Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
CDABS	Specific	Absolute value	`COMPLEX*16`	`REAL*8`
CDEXP	Specific	Exponential, ea	`COMPLEX*16`	`COMPLEX*16`
CDLOG	Specific	Natural log	`COMPLEX*16`	`COMPLEX*16`
CDSQRT	Specific	Square root	`COMPLEX*16`	`COMPLEX*16`
CDSIN	Specific	Sine	`COMPLEX*16`	`COMPLEX*16`
CDCOS	Specific	Cosine	`COMPLEX*16`	`COMPLEX*16`
DCMPLX	Specific	Convert to Dcomplex	Any numeric	`COMPLEX*16`
DCONJG	Specific	Complex conjugate	`COMPLEX*16`	`COMPLEX*16`
DIMAG	Specific	Imaginary part of complex	`COMPLEX*16`	`REAL*8`
DREAL	Specific	Real part of complex	`COMPLEX*16`	`REAL*8`

Degree-Based Trigonometric

Table 6-13 Degree-Based Trigonometric Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
SIND	Generic	Sine	-	-
SIND	Specific	Sine	REAL*4	REAL*4
DSIND	Specific	Sine	REAL*8	REAL*8
QSIND	Specific	Sine	REAL*8	REAL*16
COSD	Generic	Cosine	-	-
COSD	Specific	Cosine	REAL*4	REAL*4
DCOSD	Specific	Cosine	REAL*8	REAL*8
QCOSD	Specific	Cosine	REAL*16	REAL*16
TAND	Generic	Tangent	-	-
TAND	Specific	Tangent	REAL*4	REAL*4
DTAND	Specific	Tangent	REAL*8	REAL*8
QTAND	Specific	Tangent	REAL*16	REAL*16
ASIND	Generic	Arc sine	-	-
ASIND	Specific	Arc sine	REAL*4	REAL*4
DASIND	Specific	Arc sine	REAL*8	REAL*8
QASIND	Specific	Arc sine	REAL*16	REAL*16
ACOSD	Generic	Arc cosine	-	-
ACOSD	Specific	Arc cosine	REAL*4	REAL*4
DACOSD	Specific	Arc cosine	REAL*8	REAL*8
QACOSD	Specific	Arc cosine	REAL*16	REAL*16
ATAND	Generic	Arc tangent	-	-
ATAND	Specific	Arc tangent	REAL*4	REAL*4
DATAND	Specific	Arc tangent	REAL*8	REAL*8
QATAND	Specific	Arc tangent	REAL*16	REAL*16
ATAN2D	Generic	Arc tangent of a1/a2	-	-
ATAN2D	Specific	Arc tangent of a1/a2	REAL*4	REAL*4
DATAN2D	Specific	Arc tangent of a1/a2	REAL*8	REAL*8
QATAN2D	Specific	Arc tangent of a1/a2	REAL*16	REAL*16

Bit-Manipulation

Table 6-14 Bit-Manipulation Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IBITS	Generic	From a1, initial bit a2, extract a3 bits	-	-
IIBITS	Specific	From a1, initial bit a2, extract a3 bits	INTEGER*2	INTEGER*2
JIBITS	Specific	From a1, initial bit a2, extract a3 bits	INTEGER*4	INTEGER*4
ISHFT	Generic	Shift a1 logically by a2 bits *	-	-
ISHFTC	Generic	In a1, circular shift by a2 places, of right a3 bits	-	-
IISHFTC	Specific	In a1, circular shift by a2 places, of right a3 bits	INTEGER*2	INTEGER*2
JISHFTC	Specific	In a1, circular shift by a2 places, of right a3 bits	INTEGER*4	INTEGER*4
IISHFT	Specific	Shift a1 logically left by a2 bits	INTEGER*2	INTEGER*2
JISHFT	Specific	Shift a1 logically left by a2 bits	INTEGER*4	INTEGER*4
IAND	Generic	Bitwise AND of a1, a2	-	-
IIAND	Specific	Bitwise AND of a1, a2	INTEGER*2	INTEGER*2
JIAND	Specific	Bitwise AND of a1, a2	INTEGER*4	INTEGER*4
IOR	Generic	Bitwise OR of a1, a2	-	-
IIOR	Specific	Bitwise OR of a1, a2	INTEGER*2	INTEGER*2
JIOR	Specific	Bitwise OR of a1, a2	INTEGER*4	INTEGER*4
IEOR	Generic	Bitwise exclusive OR of a1, a2	-	-
IIEOR	Specific	Bitwise exclusive OR of a1, a2	INTEGER*2	INTEGER*2
JIEOR	Specific	Bitwise exclusive OR of a1, a2	INTEGER*4	INTEGER*4
NOT	Generic	Bitwise complement	-	-
INOT	Specific	Bitwise complement	INTEGER*2	INTEGER*2
JNOT	Specific	Bitwise complement	INTEGER*4	INTEGER*4
IBSET	Generic	In a1, set bit a2 to 1	-	-
IIBSET	Specific	In a1, set bit a2 to 1; return new a1	INTEGER*2	INTEGER*2
JIBSET	Specific	In a1, set bit a2 to 1; return new a1	INTEGER*4	INTEGER*4
BTEST	Generic	If bit a2 of a1 is 1, return .TRUE.	-	-
BITEST	Specific	If bit a2 of a1 is 1, return .TRUE.	INTEGER*2	LOGICAL*2
BJTEST	Specific	If bit a2 of a1 is 1, return .TRUE.	INTEGER*4	INTEGER*4
IBCLR	Generic	In a1, set bit a2 to 0; return new a1	-	-
IIBCLR	Specific	In a1, set bit a2 to 0; return new a1	INTEGER*2	INTEGER*2
JIBCLR	Specific	In a1, set bit a2 to 0; return new a1	INTEGER*4	INTEGER*4

* ISHFT — If *a2* is positive, then shift left; if negative, then shift right.

Multiple Integer Types

The possibility of multiple integer types is not addressed by the FORTRAN Standard. f77 copes with their existence by treating a specific INTEGER → INTEGER function name (IABS, and so forth) as a special sort of generic. The argument type is used to select the appropriate runtime routine name, which is not accessible to the programmer. VMS FORTRAN takes a similar approach but makes the specific names available.

Table 6-15 Integer Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IIABS	Specific	Absolute value	INTEGER*2	INTEGER*2
JIABS	Specific	Absolute value	INTEGER*4	INTEGER*4
IMAX0	Specific	Maximum [1]	INTEGER*2	INTEGER*2
JMAX0	Specific	Maximum [1]	INTEGER*4	INTEGER*4
IMIN0	Specific	Minimum [1]	INTEGER*2	INTEGER*2
JMIN0	Specific	Minimum [1]	INTEGER*4	INTEGER*4
IIDIM	Specific	Positive difference [2]	INTEGER*2	INTEGER*2
JIDIM	Specific	Positive difference [2]	INTEGER*4	INTEGER*4
IMOD	Specific	Remainder of a1/a2	INTEGER*2	INTEGER*2
JMOD	Specific	Remainder of a1/a2	INTEGER*4	INTEGER*4
IISIGN	Specific	Transfer sign, |a1|* sign(a2)	INTEGER*2	INTEGER*2
JISIGN	Specific	Transfer sign, |a1|* sign(a2)	INTEGER*4	INTEGER*4

1. At least two arguments

2. Positive difference: a1-min(a1,a2))

Functions Coerced to a Particular Type

Some VMS FORTRAN functions coerce to a particular `INTEGER` type.

Table 6-16 Translated Functions that VMS Coerces to a Particular Type

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IINT	Specific	Truncation toward zero	`REAL*4`	`INTEGER*2`
JINT	Specific	Truncation toward zero	`REAL*4`	`INTEGER*4`
IIDINT	Specific	Truncation toward zero	`REAL*8`	`INTEGER*2`
JIDINT	Specific	Truncation toward zero	`REAL*8`	`INTEGER*4`
IQINT	Generic	Truncation toward zero	`REAL*16`	`INTEGER`
IIQINT	Specific	Truncation toward zero	`REAL*16`	`INTEGER*2`
JIQINT	Specific	Truncation toward zero	`REAL*16`	`INTEGER*4`
ININT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*4`	`INTEGER*2`
JNINT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*4`	`INTEGER*4`
IIDNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*8`	`INTEGER*2`
JIDNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*8`	`INTEGER*4`
IQNINT	Generic	Nearest integer, `INT(a+.5*sign(a))`	`REAL*16`	`INTEGER`
IIQNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*16`	`INTEGER*2`
JIQNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*16`	`INTEGER*4`
IIFIX	Specific	Fix	`REAL*4`	`INTEGER*2`
JIFIX	Specific	Fix	`REAL*4`	`INTEGER*4`
IMAX1	Specific	Maximum	`REAL*4`	`INTEGER*2`
JMAX1	Specific	Maximum	`REAL*4`	`INTEGER*4`
IMIN1	Specific	Minimum	`READ*4`	`INTEGER*2`
JMIN1	Specific	Minimum	`READ*4`	`INTEGER*4`

The `REAL*16` is *SPARC only.*

Functions Translated to a Generic Name

In some cases, each VMS specific name is translated into an `f77` generic name.

Table 6-17 Other Conversions by `f77`

Name	Gen/Spec	Function	Arg Type	Result Type
FLOATI	Specific	Convert to REAL*4	INTEGER*2	REAL*4
FLOATJ	Specific	Convert to REAL*4	INTEGER*4	REAL*4
DFLOAT	Generic	Convert to REAL*8	INTEGER	REAL*8
DFLOTI	Specific	Convert to REAL*8	INTEGER*2	REAL*8
DFLOTJ	Specific	Convert to REAL*8	INTEGER*4	REAL*8
AIMAX0	Specific	Maximum	INTEGER*2	REAL*4
AJMAX0	Specific	Maximum	INTEGER*4	REAL*4
AIMIN0	Specific	Minimum	INTEGER*2	REAL*4
AJMIN0	Specific	Minimum	INTEGER*4	REAL*4

Zero Extend

The following zero-extend functions are recognized by `f77`. The first unused high-order bit is set to zero and extended toward the higher-order end to the width indicated in the table

Table 6-18 Zero-Extend Functions

Name	Gen/Spec	Function	Arg Type	Result Type
ZEXT	Generic	Zero-extend	-	-
IZEXT	Specific	Zero-extend	BYTE LOGICAL*1 LOGICAL*2 INTEGER*2	INTEGER*2
JZEXT	Specific	Zero-extend	BYTE LOGICAL*1 LOGICAL*2 LOGICAL*4 INTEGER INTEGER*2 INTEGER*4	INTEGER*4

6

FORTRAN Library Routines 7

See also Chapter 8, “VMS Routines,” for more routines.

7.1 `abort`: *Terminate and Write Memory to Core File*

Subroutine

```
call abort
```

`abort` cleans up the I/O buffers and then aborts producing a `core` file in the current directory. See also `abort`(3).

7.2 `access`: *Check File for Permissions or Existence*

Function

`status = access ( name, mode )`			
`name`	`character`	Input	File name
`mode`	`character`	Input	Permissions
Return value	`integer`	Output	*`status`*=0: OK *`status`*>0: Error code

`access` tells you if you can access the file *`name`* with the permissions *`mode`*.

You can set *mode* to one or more of r, w, or x, in any order, and in any combination, where r, w, x have the following meanings:

r	read
w	write
x	execute
blank	existence

Example 1: Write, and arguments are literals.

```
      integer  access, status
      status = access ( 'taccess.data', 'w' )
      if ( status .eq. 0 ) write(*,*) "ok"
      if ( status .ne. 0 ) write(*,*) 'cannot write', status
      end
```

Example 2: Test for existence.

```
      integer  access, status
      status = access ( 'taccess.data', ' ' )! blank mode
      if ( status .eq. 0 ) write(*,*) "ok"
      if ( status .ne. 0 ) write(*,*) 'no such file', status
      end
```

See also access(2), perror(3F).

7.3 alarm: *Execute a Subroutine after a Specified Time*

Function

n = alarm (*time*, *sbrtn*)			
time	integer	Input	No. of seconds to wait (0=do not call)
sbrtn	Routine name	Input	Subprogram to execute must be listed in an external statement.
Return value	integer	Output	Time remaining on the last alarm

Example: `alarm` – wait 9 seconds then call `sbrtn`.

```
      integer alarm, time / 1 /
      common / alarmcom / i
      external sbrtn
      i = 9
      write(*,*) i
      nseconds =  alarm ( time, sbrtn )
      do n = 1,100000     ! Wait until alarm activates sbrtn.
           r = n      ! (any calculations that take enough time)
           x=sqrt(r)
      end do
      write(*,*) i
      end

      subroutine sbrtn
      common / alarmcom / i
      i = 3              ! Do no I/O in this routine.
      return
      end
```

See also: `alarm`(3C), `sleep`(3F), `signal`(3F)

Restrictions:

- A subroutine cannot pass its own name to `alarm` because of restrictions in the FORTRAN Standard.
- Your subroutine must not do any I/O because the `alarm` routine generates signals, and signals interfere with any I/O. I/O is interrupt-driven.
- Do not call `alarm()` from a FORTRAN MP program—it has unpredictable behavior in MP mode.

7.4 `bit`: *Bit Functions:* `and`, `or`, ..., `bit`, `setbit`, ...

Definitions

`and`	Computes the bitwise *and* of its arguments
`or`	Computes the bitwise *or* of its arguments
`xor`	Computes the bitwise *exclusive or* of its arguments
`not`	Returns the bitwise *complement* of its argument
`lshift`	Is a logical left shift with no end around carry
`rshift`	Is an arithmetic right shift with sign extension
`bis`	Sets bit *`bitnum`* in *`word`*
`bic`	Clears bit *`bitnum`* in *`word`*
`bit`	Tests bit *`bitnum`* in *`word`* and returns .true. if the bit is a 1 (one), and returns .false. if the bit is a 0 (zero)
`setbit`	Sets bit *`bitnum`* in *`word`* to 1 if state is nonzero and clears it otherwise

`and`, `or`, `xor`, `not`, `rshift`, `lshift`

`x` = `and(` *`word1, word2`* `)`		
`x` = `or(` *`word1, word2`* `)`		
`x` = `xor(` *`word1, word2`* `)`		
`x` = `not(` *`word`* `)`		
`x` = `rshift(` *`word, nbits`* `)`		
`x` = `lshift(` *`word, nbits`* `)`		
`word1, word2, word, nbits`	`integer` or `logical` (short or long)	Input

- These are generic functions expanded inline by the compiler.
- No test is made for a reasonable value of *`nbits`*.

Example: `and, or, xor, not`.

```
      print 1, and(7,4), or(7,4), xor(7,4), not(4)
 1    format(4x 'and(7,4)', 5x 'or(7,4)', 4x 'xor(7,4)',
&           6x 'not(4)'/4o12.11)
      end
demo$ f77 -silent tandornot.f
demo$ a.out
    and(7,4)     or(7,4)    xor(7,4)      not(4)
 00000000004 00000000007 00000000003 37777777773
demo$
```

Example: `lshift, rshift`.

```
      integer lshift, rshift
      print 1, lshift(7,1), rshift(4,1)
 1    format(1x 'lshift(7,1)', 1x 'rshift(4,1)'/2o12.11)
      end
demo$ f77 -silent tlrshift.f
demo$ a.out
 lshift(7,1) rshift(4,1)
 00000000016 00000000002
demo$
```

bic, bis, bit, setbit

`call bic(` *bitnum*`,` *word* `)`		
`call bis(` *bitnum*`,` *word* `)`		
`call setbit(` *bitnum*`,` *word*`,` *state* `)`		
x `= bit(` *bitnum*`,` *word* `)`		
Return value	`logical`	Logical value.
bitnum	`integer*4`	Input
state	`integer*4`	Input
word	`integer*4`	Input and output (an input that gets changed)

Bits are numbered such that bit 0 is the *least* significant bit, and bit 31 is the *most* significant.

`bic`, `bis`, and `setbit` are external subroutines, and `bit` is an external function.

Example 3: `bic`, `bis`, `setbit`, `bit`.

```
      integer bitnum/2/, state/0/, word/7/
      logical bit

      print 1, word
 1    format(13x 'word', o12.11)

      call bic( bitnum, word )
      print 2, word
 2    format('after bic(2,word)', o12.11)

      call bis( bitnum, word )
      print 3, word
 3    format('after bis(2,word)', o12.11)

      call setbit( bitnum, word, state )
      print 4, word
 4    format('after setbit(2,word,0)', o12.11)

      print 5, bit(bitnum, word)
 5    format('bit(2,word)', L )
      end
demo$ f77 -silent tbit.f
demo$ a.out
            word 00000000007
after bic(2,word) 00000000003
after bis(2,word) 00000000007
after setbit(2,word,0) 00000000003
bit(2,word) F
demo$
```

7.5 `chdir`*: Change Default Directory*

Function

n = chdir(*dirname*)			
dirname	character	Input	Directory name
Return value	integer	Output	*n*=0: OK, *n*>0: Error code

Example: chdir, change cwd to MyDir.

```
      integer  chdir, n
      n =  chdir ( 'MyDir' )
      if ( n .ne. 0 ) stop 'chdir: error'
      end
```

See also: chdir(2), cd(1), perror(3F)

Path names can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Use of this function can cause inquire by unit to fail.

Certain FORTRAN file operations reopen files by name. Using chdir while doing I/O can cause the runtime system to lose track of files created with relative path names (including files created by open statements without file names).

7.6 chmod: *Change the Mode of a File*

Function

n = chmod(*name*, *mode*)			
name	character	Input	Single path name
mode	character	Input	Anything recognized by chmod(1), such as o-w, 444, etc.
Return value	integer	Output	*n*= 0: OK, *n*>0: System error number

Example: chmod – add write permissions to MyFile.

```
      character*18 name, mode
      integer chmod, n
      name = 'MyFile'
      mode = '+w'
      n =  chmod( name, mode )
      if ( n .ne. 0 ) stop 'chmod: error'
      end
```

See also: chmod(1). Bugs: Path names can be no longer than MAXPATHLEN as defined in <sys/param.h>.

7.7 `date`: *Get Current System Date as a Character String*

`call date( c )`			
c	`CHARACTER*9`	Output	Variable, array, array element, or character substring

The form of the returned string ***C*** is as follows.

dd-mmm-yy	
dd	Day of the month, as a 2-digit integer
mmm	Month name as a 3-letter abbreviation
yy	Year, as a 2-digit integer

Example: `date`.

```
demo$ cat dat1.f
* dat1.f -- Get the date as a character string.
      character c*9
      call date ( c )
      write(*,"(' The date today is: ', A9 )" ) c
      end
demo$ f77 -silent dat1.f -lV77
demo$ a.out
 The date today is: 23-Sep-88
demo$
```

To use this VMS routine you need `-lV77`. If you use `-lV77` and invoke `idate()` or `time()`, then you get the VMS versions of both.

See also Section 7.27, “idate: Return Current System Date.”

7.8 dtime, etime: *Elapsed Execution Time*

Both functions have return values of elapsed time (or -1.0 as error indicator). The time is in seconds. The resolution is to a nanosecond under Solaris 2.x and is determined by the system clock frequency under SunOS 4.x..

dtime: *Elapsed Time Since the Last* dtime *Call*

For dtime, elapsed time is:

- First call: elapsed time since start of execution
- Subsequent calls: elapsed time since the last call to dtime
- Single Processor: Time used by the CPU
- Multiple Processor: Sum of times for all the CPUs (not useful—use etime)

Note – Do not call dtime from within a parallelized loop.

Function

e = dtime(*tarray*)				
tarray	real(2)	Output	*e* = -1.0:	Error: *tarray* values are undefined
			e ≠ -1.0:	User time in *tarray(1)* (if no error) System time in *tarray(2)* (if no error)
Return value	real	Output	*e* = -1.0:	Error
			e ≠ -1.0:	The sum of *tarray(1)* and *tarray(2)*

Example: `dtime()`, *single* processor.

```
      real e, dtime, t(2)
      print *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
      do i = 1, 10000
           k=k+1
      end do
      e = dtime( t )
      print *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
      end
demo$ f77 -silent tdtime.f
demo$ a.out
elapsed:  0., user:  0., sys:  0.
elapsed:   0.180000, user:    6.00000E-02, sys:   0.120000
demo$
```

`etime`: *Elapsed Time Since Start of Execution*

For `etime`, elapsed time is:

- *Single Processor*—CPU time for the calling process
- *Multiple Processor*—Wallclock time while processing your program

How FORTRAN Decides Single Processor or Multiple Processor

For a FORTRAN MP program (uses an MP option—ultimately, linked with `libF77_mt`), if the environment variable `PARALLEL` is:

- *Undefined*, the current run is *single* processor.
- *Defined* and in the range 1, 2, 3, …, the current run is *multiple processor.*
- *Defined*, but some value other than 1, 2, 3, …, the results are unpredictable.

Function

`e = etime( tarray )`					
tarray	`real(2)`	Output	*e* = -1.0*:*	Error: *tarray* values are undefined	
			e ≠ -1.0:	Single Processor: User time in	*tarray(1)*
				System time in	*tarray(2)*
				Multiple Processor: Wall clock time in	*tarray(1)*
				0.0 in	*tarray(2)*

e = etime(*tarray*)			
Return value	real	Output	*e* = -1.0: Error *e* ≠ -1.0: The sum of *tarray(1)* and *tarray(2)*

Example: etime(), *single* processor.

```
      real e, etime, t(2)
      do i = 1, 10000
           k=k+1
      end do
      e = etime( t )
      print *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
      end
demo$ f77 -silent tetime.f
demo$ a.out
elapsed:   0.190000, user:    6.00000E-02, sys:   0.130000
demo$
```

See also times(2), f77(1), and the *FORTRAN User's Guide*.

7.9 exit: *Terminate a Process and Set the Status*

Subroutine

call exit(*status*)		
status	integer	Input

Example: exit().

```
      integer status
      status = 7
      call exit( status )
      end
```

exit flushes and closes all the process's files, and notifies the parent process if it is executing a wait.

The low-order 8 bits of *status* are available to the parent process. These 8 bits are shifted left 8 bits, and all other bits are zero. (Therefore *status* should be in the range 256 - 65280). This call will never return.

The C function `exit` can cause cleanup actions before the final '`sys exit`'.

If you call `exit` without an argument, you will get a warning message, and a zero will be automatically provided as an argument. See also: `exit`(2), `fork`(2), `fork`(3f), `wait`(2), `wait`(3f).

7.10 `f77_floatingpoint`: *FORTRAN IEEE Definitions*

The file `f77_floatingpoint.h` defines constants and types used to implement standard floating-point according to ANSI/IEEE Std 754-1985.

Include the file in a source program as follows:

```
#include <f77/f77_floatingpoint.h>
```

The file `f77_floatingpoint.h` defines constants and types used to implement standard floating-point according to ANSI/IEEE Std 754-1985. Use these constants and types to write more easily understood `.F` source files that will undergo automatic preprocessing prior to FORTRAN compilation.

IEEE Rounding Mode

`fp_direction_type`	The type of the IEEE rounding direction mode. Note that the order of enumeration varies according to hardware.

SIGFPE Handling

`sigfpe_code_type`	The type of a `SIGFPE` code.
`sigfpe_handler_type`	The type of a user-definable `SIGFPE` exception handler called to handle a particular `SIGFPE` code.
`SIGFPE_DEFAULT`	A macro indicating default `SIGFPE` exception handling: IEEE exceptions to continue with a default result and to abort for other `SIGFPE` codes.

`sigfpe_code_type`	The type of a `SIGFPE` code.
`SIGFPE_IGNORE`	A macro indicating an alternate `SIGFPE` exception handling, namely to ignore and continue execution.
`SIGFPE_ABORT`	A macro indicating an alternate `SIGFPE` exception handling, namely to abort with a core dump.

IEEE Exception Handling

`N_IEEE_EXCEPTION`	The number of distinct IEEE floating-point exceptions.
`fp_exception_type`	The type of the `N_IEEE_EXCEPTION` exceptions. Each exception is given a bit number.
`fp_exception_field_type`	The type intended to hold at least `N_IEEE_EXCEPTION` bits corresponding to the IEEE exceptions numbered by `fp_exception_type`. Thus `fp_inexact` corresponds to the least significant bit and `fp_invalid` to the fifth least significant bit. Some operations can set more than one exception.

IEEE Classification

`fp_class_type`	A list of the classes of IEEE floating-point values and symbols.

Refer to the *Numerical Computation Guide*. See also `ieee_environment`(3M), and `f77_ieee_environment`(3F).

7.11 f77_ieee_environment: *IEEE Arithmetic*

Summary

ieee_flags	*ieeer* = ieee_flags(*action*,*mode*,*in*,*out*)	
ieee_handler	*ieeer* = ieee_handler(*action*,*exception*,*hdl*)	
sigfpe	*ieeer* = sigfpe(*code*, *hdl*)	
action	character	Input
code	sigfpe_code_type	Input
mode	character	Input
in	character	Input
exception	character	Input
hdl	sigfpe_handler_type	Input
out	character	Output
Return value	integer	Output

These subprograms provide modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a FORTRAN program. They correspond closely to the functions `ieee_flags(3M)`, `ieee_handler(3M)`, and `sigfpe(3)`.

If you use `sigfpe`, you must do your own setting of the corresponding trap-enable-mask bits in the floating-point status register. The details are in the SPARC architecture manual. The `libm` function `ieee_handler` sets these trap-enable-mask bits for you.

Example 1: Set rounding direction to round toward zero, unless the hardware does not support directed rounding modes.

```
      integer ieeer
      character*1 mode, out, in
      ieeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

Example 2: Clear rounding direction to default (round toward nearest).

```
      character*1 out, in
      ieeer = ieee_flags('clear','direction', in, out )
```

Example 3: Clear all accrued exception-occurred bits.

```
      character*18 out
      ieeer = ieee_flags( 'clear', 'exception', 'all', out )
```

Example 4: If Example 3 generates the overflow exception, detect it as follows.

```
      character*18 out
      ieeer = ieee_flags( 'get', 'exception', 'overflow', out )
```

The above sets *out* to "overflow" and *ieeer* to 25. Similar coding detects exceptions such as invalid or inexact.

Example 5: hand1.f, write and use a signal handler (*Solaris 2.x*).

```
      external hand
      real r / 14.2 /,  s / 0.0 /
      i = ieee_handler( 'set', 'division', hand )
      t = r/s
      end

      integer function hand ( sig, sip, uap )
      integer sig, address
      structure /fault/
           integer address
      end structure
      structure /siginfo/
           integer si_signo
           integer si_code
           integer si_errno
           record /fault/ fault
      end structure
      record /siginfo/ sip
      address = sip.fault.address
      write (*,10) address
 10   format('Exception at hex address ', z8 )
      end
```

Read the *Numerical Computation Guid*e. See also: floatingpoint(3), signal(3), sigfpe(3), f77_floatingpoint(3F), ieee_flags(3M), ieee_handler(3M).

7.12 fdate: *Return Date and Time in an ASCII String*

Subroutine or function

call fdate(*string*)		
string	character*30	Output

or

string = fdate()			If you use it as a function, the calling routine must define the type and length of fdate.
Return value	character*30	Output	

Example 1: fdate as a subroutine.

```
      character*30 string
      call fdate( string )
      write(*,*) string
      end
```

Output:

```
 Mon Aug 1 09:24:21 PST 1993
```

Example 2: fdate as a function, same output.

```
      character*30 fdate
      write(*,*)  fdate()
      end
```

See also: ctime(3), time(3F), idate(3F)

7.13 `flush`: *Flush Output to a Logical Unit*

Subroutine

`call flush(` *`lunit`* `)`			
`lunit`	`integer`	Input	Logical unit

The `flush` subroutine flushes the contents of the buffer for logical unit `lunit` to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

See also `fclose(3S)`.

7.14 `fork`: *Create a Copy of the Current Process*

Function

`n` `= fork()`			
Return value	`integer`	Output	*n*>0: *n*=Process ID of copy *n*<0, *n*=-(system error code)

The `fork` function creates a copy of the calling process. The only distinction between the 2 processes is that the value returned to one of them (referred to as the *parent* process) will be the *process ID* of the copy. The copy is usually referred to as the `child` process. The value returned to the child process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

Example: `fork()`.

```
      integer fork, pid
      pid = fork()
      end
```

A corresponding `exec` routine has not been provided because there is no satisfactory way to retain open logical units across the `exec` routine. However, the usual function of `fork/exec` can be performed using `system`(3F). See also: `fork`(2), `wait`(3F), `kill`(3F), `system`(3F), `perror`(3F).

7.15 `free`: *Deallocate Memory Allocated by Malloc*

Subroutine

`call free ( ptr )`		
ptr	pointer	Input

`free` deallocates a region of memory previously allocated by `malloc`. The region of memory is returned to the memory manager; it is not explicitly available to the user's program.

Example: `free()`.

```
      real x
      pointer ( ptr, x )
      ptr = malloc ( 10000 )
      call free ( ptr )
      end
```

See Section 7.40, "malloc: Allocate Memory and Get Address," for details.

7.16 `fseek`, `ftell`: *Reposition a File*

`fseek`: *Reposition a File on a Logical Unit*

Function

n = fseek(*lunit*, *offset*, *from*)			
lunit	integer	Input	Open logical unit
offset	integer	Input	Offset in bytes relative to position specified by *from*
from	integer	Input	0=Beginning of file 1=Current position 2=End of file
Return value	integer	Output	*n*=0: OK. *n*>0: System error code.

Example: `fseek()` – Reposition `MyFile` to 2 bytes from beginning

```
      integer fseek, lunit/1/, offset/2/, from/0/, n
      open( UNIT=lunit, FILE='MyFile' )
      n = fseek( lunit, offset, from )
      if ( n .gt. 0 ) stop 'fseek error'
      end
```

`ftell`: *Return Current Position of File*

Function

n = ftell(*lunit*)			
lunit	integer	Input	Open logical unit
Return value	integer	Input	*n*>=0: *n*=offset in bytes from start of file. *n*<0: *n*=-(system error code).

Example: `ftell()`.

```
      integer ftell, lunit/1/, n
      open( UNIT=lunit, FILE='MyFile' )
*     …
      n = ftell( lunit )
      if ( n .lt. 0 ) stop 'ftell error'
      end
```

See also `fseek`(3S), `perror`(3F)

7.17 getarg, iargc: *Get Command-line Arguments*

getarg: *Get the k*th *Command Line Argument*

Subroutine

call getarg(*k*, *arg*)			
k	integer	Input	Index of argument (0=first=command name)
arg	character**n*	Output	*k*th argument
n	integer	Size of *arg*	Large enough to hold longest argument

iargc: *Get the Count of Command-line Arguments*

Function

m = iargc()			
Return value	integer	Output	Number of arguments on command line

Example: iargc and getarg, get argument count and each argument.

```
      character argv*10
      integer i, iargc, n
      n = iargc()
      do i = 1, n
           call getarg( i, argv )
           write( *, '( i2, 1x, a )' ) i, argv
      end do
      end
```

Sample run of above source (after compiling):

```
demo$ a.out first second last
1 first
2 second
3 last
demo$
```

See also execve(2), getenv(3F).

7.18 `getc`, `fgetc`: *Get Next Character*

`getc`: *Get Next Character from stdin*

Function

status = getc(*char*)			
char	character	Output	Next character
Return value	integer	Output	*status*=0: OK *status*=-1: End of File *status*>0: System error code or f77 I/O error code

Example: `getc` gets each character from keyboard. Note the Control-`D` (EOF).

```
      character char
      integer getc, status
      status = 0
      do while ( status .eq. 0 )
           status = getc( char )
           write(*, '(i3, o4.3)') status, char
      end do
      end
```

Sample run of above source (after compiling):

```
demo$ a.out
ab
^D
0 141
0 142
0 012
-1 012
demo$
```

For any logical unit, do not mix normal FORTRAN input with `getc`().

fgetc: *Get Next Character from Specified Logical Unit*

Function

status = fgetc(*lunit*, *char*)			
lunit	integer	Input	Logical unit
char	character	Output	Next character
Return value	integer	Output	*status*=-1: End of File *status*>0: System error code or f77 I/O error code

Example: `fgetc` gets each character from `tfgetc.data`. Note linefeeds (Octal 012).

```
      character char
      integer fgetc, status
      open( unit=1, file='tfgetc.data' )
      status = 0
      do while ( status .eq. 0 )
           status = fgetc( 1, char )
           write(*, '(i3, o4.3)') status, char
      end do
      end
```

Sample run of above source (after compiling)

```
demo$ cat tfgetc.data
ab
yz
demo$ a.out
0 141
0 142
0 012
0 171
0 172
0 012
-1 012
demo$
```

For any logical unit, do not mix normal FORTRAN input with `fgetc()`.

See also: `getc`(3S), `intro`(2), `perror`(3F).

7.19 getcwd: *Get Path of Current Working Directory*

Function

status = getcwd(*dirname*)			
dirname	character*n	Output	Path name of the current working directory
Return value	integer	Output	*status*=0: OK *status*>0: Error code
n	integer	Size of *dirname*, in bytes	Must be big enough for longest path name

Example: getcwd.

```
      integer getcwd, status
      character*64 dirname
      status = getcwd( dirname )
      if ( status .ne. 0 ) stop 'getcwd: error'
      write(*,*) dirname
      end
```

See also: chdir(3F), perror(3F), getwd(3).

Bug: Path names can be no longer than MAXPATHLEN as defined in <sys/param.h>.

7.20 getenv: *Get Value of Environment Variables*

Subroutine

call getenv(*ename*, *evalue*)			
ename	character*n	Input	Name of the environment variable sought
evalue	character*n	Output	Value of the environment variable found, blanks if not successful
n	integer	Size of *evalue*	*n* must be large enough for the value.

The getenv subroutine searches the environment list for a string of the form *ename=evalue* and returns the value in *evalue* if such a string is present; otherwise it fills *evalue* with blanks.

Example: getenv()

```
      character*18  evalue
      call getenv( 'SHELL', evalue )
      write(*,*) "'", evalue, "'"
      end
```

See also: execve(2), environ(5).

7.21 getfd: *Get File Descriptor for External Unit Number*

Function

fildes = getfd(*unitn*)			
unitn	integer	Input	External unit number
Return value	integer	Output	File descriptor if file is connected -1 if file is not connected

Example: getfd().

```
      integer fildes, getfd, unitn/1/
      open( unitn, file='tgetfd.data' )
      fildes = getfd( unitn )
      if ( fildes .eq. -1 ) stop 'getfd: file not connected'
      write(*,*) 'file descriptor = ', fildes
      end
```

See also open(2).

7.22 `getfilep`: *Get File Pointer for External Unit Number*

Function

irtn = *c_read*(getfilep(*unitn*), *inbyte*, 1)			
c_read	C function	Input	You write this C function. Sample below.
unitn	integer	Input	External unit number
getfilep	integer	Return value	File pointer if file is connected -1 if file is not connected

This function is used for mixing standard FORTRAN I/O with C I/O. Such mixing is nonportable, and is not guaranteed for subsequent releases of the operating system or FORTRAN. Use of this function is not recommended, and no direct interface is provided. You must enter your own C routine to use the value returned by `getfilep`. A sample C routine is shown below.

Example: FORTRAN uses `getfilep` by passing it to a C function.

tgetfilepF.f

```
      character*1  inbyte
      integer*4    c_read,  getfilep, unitn / 5 /
      external     getfilep
      write(*,'(a,$)') 'What is the digit? '

      irtn = c_read( getfilep( unitn ), inbyte, 1 )

      write(*,9)  inbyte
    9 format('The digit read by C is ', a )
      end
```

Sample C function actually using `getfilep`.

tgetfilepC.c

```
#include <stdio.h>
int c_read_ ( fd, buf, nbytes, buf_len )
FILE **fd ;
char *buf ;
int *nbytes, buf_len ;
{
   return fread( buf, 1, *nbytes, *fd ) ;
}
```

Sample compile/build/run.

```
demo 11% cc -c tgetfilepC.c
demo 12% f77 tgetfilepC.o tgetfilepF.f
tgetfileF.f:
MAIN:
demo 13% a.out
What is the digit? 3
The digit read by C is 3
demo 14%
```

Read the chapter on the C-FORTRAN interface in the *FORTRAN User's Guide*. See also open(2).

7.23 getlog*: Get User's Login Name*

Subroutine

call getlog(*name*)			
name	character**n*	Output	User's login name, or all blanks if the process is running detached from a terminal.
n	integer	Size of *name*	Large enough to hold longest name

Example: getlog.

```
      character*18 name
      call getlog( name )
      write(*,*) "'", name, "'"
      end
```

See also getlogin(3).

7.24 `getpid`: *Get Process ID*

Function

pid = getpid()			
Return value	integer	Output	Process ID of the current process

Example: `getpid`.

```
      integer getpid, pid
      pid = getpid()
      write(*,*) 'process id = ', pid
      end
```

See also `getpid`(2).

7.25 `getuid`, `getgid`: *Get User or Group ID of Process*

`getuid`: *Get User ID of the Process*

Function

uid = getuid()			
Return value	integer	Output	User ID of the process

`getgid`: *Get Group ID of the Process*

Function

gid = getgid()			
Return value	integer	Output	Group ID of the process

Example: `getuid()` and `getpid()`.

```
      integer getuid, getgid, gid, uid
      uid = getuid()
      gid = getgid()
      write(*,*) uid, gid
      end
```

See also: `getuid`(2).

7.26 `hostnm`*: Get Name of Current Host*

Function

status = hostnm(*name*)			
name	character*n	Output	Name of current host
Return value	integer	Output	*status*=0: OK, *status*>0: Error
n	integer	Size of *name*	Big enough to hold host name, or memory gets clobbered.

Example: `hostnm()`.

```
      integer hostnm, status
      character*8 name
      status = hostnm( name )
      write(*,*) 'host name = "', name, '"'
      end
```

See also `gethostname`(2).

7.27 `idate`: *Return Current System Date*

`idate`	Put current system date into an integer array: day, month, and year (standard version).
`idate`	Put current system date into three integer variables: month, day, and year (VMS version).

If you use the `-lV77` compiler option to request the VMS library, then you get the VMS versions of both `time()` and of `idate()`; otherwise, you get the standard versions.

Standard Version

Subroutine

Put the current system date into one integer array: day, month, and year.

`call idate( iarray )`			
`iarray`	`integer`	Output	`array(3)`, Note the order: day, month, year

Example: `idate` (standard version).

```
      integer iarray(3)
      call idate( iarray )
      write(*, "(' The date is: ',3i5)" )  iarray
      end
```

Compile and run the above source:

```
demo$ f77 -silent tidate.f
demo$ a.out
 The date is: 10 8 1991
demo$
```

VMS Version

Subroutine

Put the current system date into three integer variables: month, day, and year.

`call idate( m, d, y )`			
m	`integer`	Output	Month (1 - 12)
d	`integer`	Output	Day (1 - 7)
y	`integer`	Output	Year (1 - 99)

Example: `idate` (VMS version).

```
      integer m, d, y
      call idate ( m, d, y )
      write (*, "(' The date is: ',3i5)" ) m, d, y
      end
```

Compile and run the above source (note the `-lV77`):

```
demo$ f77 -silent tidateV.f -lV77
demo$ a.out
 The date is: 8 10 91
demo$
```

7.28 `itime:` *Current System Time*

`itime`	Put current system time into an integer array: Hour, minute, second.

Subroutine

`call itime( iarray )`			
iarray	`integer`	Output	`array(3)`, Note the order: hour, minute, second

Example: `itime`.

```
      integer iarray(3)
      call itime( iarray )
      write (*, "(' The time is: ',3i5)" )  iarray
      end
```

Compile and run the above source:

```
demo$ f77 -silent titime.f
demo$ a.out
 The time is: 15 42 35
demo$
```

See also `time`(3f), `ctime`(3F), `fdate`(3F).

7.29 `index`*: Index or Length of Substring*

`index(`*`a1`*`,`*`a2`*`)`	Index of first occurrence of string *a2* in string *a1*
`rindex(`*`a1`*`,`*`a2`*`)`	Index of last occurrence of string *a2* in string *a1*
`lnblnk(`*`a1`*`)`	Index of last nonblank in string *a1*
`len(`*`a1`*`)`	Declared length of string *a1*

`index`*: First Occurrence of String A2 in String A1*

Function (intrinsic)

n = index(*a1*, *a2*)			
a1	character	Input	Main string
a2	character	Input	Substring
Return value	integer	Output	*n*>0: Index of first occurrence of *a2* in *a1* *n*=0: *a2* does not occur in *a1*.

rindex: *Last Occurrence of String A2 in String A1*

Function

n = rindex(*a1*, *a2*)			
a1	character	Input	Main string
a2	character	Input	Substring
Return value	integer	Output	*n*>0: Index of last occurrence of *a2* in *a1* *n*=0: *a2* does not occur in *a1*.

lnblnk: *Last Nonblank in String A1*

Function

n = lnblnk(*a1*)			
a1	character	Input	String
Return value	integer	Output	*n*>0: Index of last nonblank in *a1* *n*=0: *a1* is all nonblank

len: *Declared Length of String A1*

Function (intrinsic)

declen = len(*a1*)			
a1	character	Input	String
Return value	integer	Output	Declared length of *a1*

This is useful since all f77 character objects are fixed length, blank padded.

Example: `len()`, `index()`, `rindex()`, `lnblnk()`.

```
*                      123456789 123456789 1234
      character s*24 / 'abcPDQxyz...abcPDQxyz   ' /
      integer declen, index, first, last, len, lnblnk, rindex
      declen = len( s )
      first = index( s, 'abc' )
      last = rindex( s, 'abc' )
      lastnb = lnblnk( s )
      write(*,*) declen, lastnb
      write(*,*) first, last
      end
demo$ f77 -silent tindex.f
demo$ a.out
32 21
1 13
demo$
```

In the above example, `declen` is 32, not 21.

7.30 `inmax`*: Return Maximum Positive Integer*

Function

m = inmax()			
Return value	integer	Output	The maximum positive integer

Example: `inmax`.

```
      integer inmax, m
      m = inmax()
      write(*,*) m
      end
demo$ f77 -silent tinmax.f
demo$ a.out
   2147483647
demo$
```

See also `libm_single`(3f), `libm_double`(3f).

7

7.31 `ioinit`: *Initialize I/O: Carriage Control, File Names, …*

Purpose

The `IOINIT` routine establishes properties of file I/O for files opened after the call to `IOINIT`. The file I/O properties that `IOINIT` controls are as follows:

Carriage control	Recognize carriage control on any logical unit.
Blanks/zeroes	Treat blanks in input data fields as blanks or zeroes.
File position	Open files at beginning or at EoF.
Prefix	Find and open files named *prefixNN*, $0 \le NN \le 19$.

Implementation

`IOINIT` does the following:

- Initializes global parameters specifying f77 file I/O properties
- Opens logical units 0 through 19 with the specified file I/O properties (attaches externally defined files to logical units at runtime)

Duration of File I/O Properties

The file I/O properties apply as long as the connection exists. If you close the unit, the properties no longer apply. The exception is the preassigned units 5 and 6, to which *Carriage Control* and *Blanks/Zeroes* apply at any time.

Internal Flags

`IOINIT` uses labeled common to communicate with the runtime I/O system. It stores internal flags in the equivalent of the following labeled common block.

```
      INTEGER*2 IEOF, ICTL, IBZR
      COMMON /_ _IOIFLG/ IEOF, ICTL, IBZR ! Not in user name space.
```

In releases prior to SC3.0.1, the labeled common block was named `IOIFLG`. We changed this to `_ _IOIFLG` so that a user common block named `IOIFLG` does not cause a disaster. This is safer because `_ _IOIFLG` is not part of the user name space

Source Code

Some user's needs are not satisfied with a generic version of `IOINIT`, so we provide the source code. It is written in FORTRAN 77 and is located as follows:

- For a standard installation, it is in `/opt/SUNWspro/SC3.0.1/src/ioinit.f`
- If you installed in /mydir, it is in */mydir*`/SC3.0.1/src/ioinit.f`

Usage

`call ioinit (` *`cctl`*`,` *`bzro`*`,` *`apnd`*`,` *`prefix`*`,` *`vrbose`* `)`			
`cctl`	`logical`	Input	True: Recognize carriage control, all formatted output (except unit 0)
`bzro`	`logical`	Input	True: Treat trailing and imbedded blanks as zeroes.
`apnd`	`logical`	Input	True: Open files at EoF. (Append)
`prefix`	`character*`*`n`*	Input	Nonblank: For unit *`NN`*, seek and open file *`prefixNN`*
`vrbose`	`logical`	Input	True: Report `ioinit` activity as it happens

See also `getarg`(3F), `getenv`(3F).

Restrictions

- *`prefix`* can be no longer than 30 characters.
- A path name associated with an environment name can be no longer than 255 characters.
- The "+" carriage control does not work.

Details of arguments

cctl

Carriage Control: By default, carriage control is not recognized on any logical unit. If *`cctl`* is `.true.`, then carriage control will be recognized on formatted output to all logical units except unit 0, the diagnostic channel. Otherwise the default will be restored.

bzro

Blanks: By default, trailing and embedded blanks in input data fields are ignored. If *bzro* is `.true.` then such blanks will be treated as zeros. Otherwise the default will be restored.

apnd

Append: By default, all files opened for sequential access are positioned at their beginning. It is sometimes necessary or convenient to open at the end-of-file so that a write will append to the existing data. If *apnd* is `.true.` then files opened subsequently on any logical unit will be positioned at their end upon opening. A value of `.false.` restores the default behavior.

prefix

Automatic file connection: If the argument *prefix* is a nonblank string, then names of the form *prefixNN* will be sought in the program environment. The value associated with each such name found will be used to open logical unit *NN* for formatted sequential access.

This search and connection is provided only for *NN* between 0 and19, inclusive. For *NN* > 19, nothing is done, but see “Source Code” on page 363.

vrbose

`IOINIT` *activity*: If the argument *vrbose* is `.true.`, then `ioinit` will report on its own activity.

Example: The program `myprogram` has the following `ioinit` call.

```
      call ioinit( .true., .false., .false., 'FORT', .false.)
```

You can assign file name in at least two ways.

`sh`:

```
demo$ FORT01=mydata
demo$ FORT12=myresults
demo$ export FORT02 FORT12
demo$ myprogram
```

csh:

```
demo% setenv FORT01 mydata
demo% setenv FORT12 myresults
demo% myprogram
```

With either shell, the ioinit call in the above example gives these results:

- Open logical unit 1 to file mydata
- Open logical unit 12 to file myresults.
- Both files are positioned at their beginning.
- Any formatted output has column 1 removed and interpreted as carriage control.
- Embedded and trailing blanks are be ignored on input.

Example: ioinit() – List and compile.

```
demo$ cat tioinit.f
     character*3  s
     call ioinit( .true., .false., .false., 'FORT', .false.)
     do i = 1, 2
          read( 1, '(a3,i4)')  s, n
          write( 12, 10 ) s, n
     end do
10   format(a3,i4)
     end
demo$ cat tioinit.data
abc 123
PDQ 789
demo$ f77 -silent tioinit.f
demo$
```

Set environment variables. Use either sh or csh.

ioinit() – sh:

```
demo$ FORT01=tioinit.data
demo$ FORT12=tioinit.au
demo$ export tioinit.data tioinit.au
demo$
```

`ioinit()` – `csh`:

```
demo$ a.out
demo$ cat tioinit.au
abc 123
PDQ 789
demo$
```

`ioinit()` – Run and test:

```
demo$ a.out
demo$ cat tioinit.au
abc 123
PDQ 789
demo$
```

7.32 `kill`: *Send a Signal to a Process*

Function

status = kill(*pid*, *signum*)			
pid	integer	Input	Process ID of one of the user's processes
signum	integer	Input	Valid signal number. See signal(3).
Return value	integer	Output	*status*=0: OK *status*>0: Error code

Example (fragment): Send a message using `kill()`.

```
      integer kill, pid, signum
*     …
      status = kill( pid, signum )
      if ( status .ne. 0 ) stop 'kill: error'
      write(*,*) 'Sent signal ', signum, ' to process ', pid
      end
```

Note that this function just sends a message. It does not necessarily kill the process. Some users have been known to consider this a UNIX misnomer. If you really mean to kill a process, use the following example.

Example (fragment): Kill a process using `kill()`.

```
      status = kill( pid, SIGKILL )
```

See also: `kill`(2), `signal`(3), `signal`(3F), `fork`(3F), `perror`(3F)

7.33 `libm_double`*:* `libm` *Double-Precision Functions*

These subprograms provide access to double-precision `libm` functions and subroutines.

Intrinsic Functions

The following FORTRAN intrinsic functions return double-precision values if they have double-precision arguments. You need not put them in a type statement. If the function needed is available as an *intrinsic* function, it is simpler to use an intrinsic than a non-intrinsic function.

The ♦ indicates it is nonstandard that this is an intrinsic function.

`sqrt(x)`	`asin(x)`	`cosd(x)` ♦
`log(x)`	`acos(x)`	`asind(x)` ♦
`log10(x)`	`atan(x)`	`acosd(x)` ♦
`exp(x)`	`atan2(x,y)`	`atand(x)` ♦
`x**y`	`sinh(x)`	`atan2d(x,y)` ♦
`sin(x)`	`cosh(x)`	`aint(x)`
`cos(x)`	`tanh(x)`	`anint(x)`
`tan(x)`	`sind(x)` ♦	`nint(x)`

Non-Intrinsic Functions

In general, these functions do *not* correspond to standard FORTRAN generic intrinsic functions—data types are determined by the usual data typing rules.

Samples: Subroutine and non-Intrinsic double-precision functions.

Note that the `DOUBLE PRECISION` functions used are in a `DOUBLE PRECISION` statement.

```
      DOUBLE PRECISION c, d_acosh, d_hypot, d_infinity, s, x, y, z
      ...
      z = d_acosh( x )
      i = id_finite( x )
      z = d_hypot( x, y )
      z = d_infinity()
      CALL d_sincos( x, s, c )
```

7

For meanings of routines and arguments, do a `man` on the routine name without the "`d_`"; it is a C man page, but the meanings are the same.

Table 7-1 Double-Precision `libm` Functions

`d_acos( x )`	`double precision`	Function	arc cosine
`d_acosd( x )`	`double precision`	Function	
`d_acosh( x )`	`double precision`	Function	arc cosh
`d_acosp( x )`	`double precision`	Function	
`d_acospi( x )`	`double precision`	Function	
`d_atan( x )`	`double precision`	Function	arc tangent
`d_atand( x )`	`double precision`	Function	
`d_atanh( x )`	`double precision`	Function	arc tanh
`d_atanp( x )`	`double precision`	Function	
`d_atanpi( x )`	`double precision`	Function	
`d_asin( x )`	`double precision`	Function	arc sine
`d_asind( x )`	`double precision`	Function	
`d_asinh( x )`	`double precision`	Function	arc sinh
`d_asinp( x )`	`double precision`	Function	
`d_asinpi( x )`	`double precision`	Function	
`d_atan2(( y, x )`	`double precision`	Function	arc tangent
`d_atan2d( y, x )`	`double precision`	Function	
`d_atan2pi( y, x )`	`double precision`	Function	
`d_cbrt( x )`	`double precision`	Function	cube root
`d_ceil( x )`	`double precision`	Function	ceiling
`d_copysign( x, y )`	`double precision`	Function	
`d_cos( x )`	`double precision`	Function	cosine
`d_cosd( x )`	`double precision`	Function	
`d_cosh( x )`	`double precision`	Function	hyperbolic cos
`d_cosp( x )`	`double precision`	Function	
`d_cospi( x )`	`double precision`	Function	
`d_erf( x )`	`double precision`	Function	error function
`d_erfc( x )`	`double precision`	Function	
`d_expm1( x )`	`double precision`	Function	(e**x)-1
`d_floor( x )`	`double precision`	Function	floor
`d_hypot( x, y )`	`double precision`	Function	hypotenuse
`d_infinity( )`	`double precision`	Function	

Variables *c*, *l*, *p*, *s*, *u*, *x*, and *y* are of type `DOUBLE PRECISION`.

If you use one of these `DOUBLE PRECISION` functions, put it into a `DOUBLE PRECISION` statement (or type it by some `IMPLICIT` statement).

`sin`**`d`**`(x)`, `asin`**`d`**`(x)`, ... involve ***degrees*** rather than ***radians***

Table 7-1 Double-Precision `libm` Functions *(Continued)*

`d_j0( x )`	`double precision`	Function	bessel
`d_j1( x )`	`double precision`	Function	
`d_jn( x )`	`double precision`	Function	
`id_finite( x )`	`integer`	Function	
`id_fp_class( x )`	`integer`	Function	
`id_ilogb( x )`	`integer`	Function	
`id_irint( x )`	`integer`	Function	
`id_isinf( x )`	`integer`	Function	
`id_isnan( x )`	`integer`	Function	
`id_isnormal( x )`	`integer`	Function	
`id_issubnormal( x )`	`integer`	Function	
`id_iszero( x )`	`integer`	Function	
`id_signbit( x )`	`integer`	Function	
`d_addran()`	`double precision`	Function	random
`d_addrans(x, p, l, u)`	`n/a`	Function	number
`d_lcran()`	`double precision`	Subroutine	generators
`d_lcrans(x, p, l, u )`	`n/a`	Subroutine	
`d_shufrans(x, p, l,u)`	`n/a`	Subroutine	
`d_lgamma( x )`	`double precision`	Function	log gamma
`d_logb( x )`	`double precision`	Function	
`d_log1p( x )`	`double precision`	Function	
`d_log2( x )`	`double precision`	Function	
`d_max_normal()`	`double precision`	Function	
`d_max_subnormal()`	`double precision`	Function	
`d_min_normal()`	`double precision`	Function	
`d_min_subnormal()`	`double precision`	Function	
`d_nextafter( x, y )`	`double precision`	Function	
`d_quiet_nan( n )`	`double precision`	Function	
`d_remainder( x, y )`	`double precision`	Function	
`d_rint( x )`	`double precision`	Function	
`d_scalb( x, y )`	`double precision`	Function	
`d_scalbn( x, n )`	`double precision`	Function	
`d_signaling_nan( n )`	`double precision`	Function	
`d_significand( x )`	`double precision`	Function	
`d_sin( x )`	`double precision`	Function	sine
`d_sind( x )`	`double precision`	Function	
`d_sinh( x )`	`double precision`	Function	hyperbolic sin
`d_sinp( x )`	`double precision`	Function	
`d_sinpi( x )`	`double precision`	Function	

Table 7-1 Double-Precision `libm` Functions *(Continued)*

`d_sincos( x, s, c )`	`n/a`	Subroutine	sine & cosine
`d_sincosd( x, s, c )`	`n/a`	Subroutine	
`d_sincosp( x, s, c )`	`n/a`	Subroutine	
`d_sincospi( x, s, c )`	`n/a`	Subroutine	
`d_tan( x )`	`double precision`	Function	tangent
`d_tand( x )`	`double precision`	Function	
`d_tanh( x )`	`double precision`	Function	hyperbolic tan
`d_tanp( x )`	`double precision`	Function	
`d_tanpi( x )`	`double precision`	Function	
`d_y0( x )`	`double precision`	Function	bessel
`d_y1( x )`	`double precision`	Function	
`d_yn( n,x )`	`double precision`	Function	

See also: `intro`(3M) and the *Numerical Computation Guide.*

7.34 `libm_quadruple`*:* `libm` *Quad-Precision Functions*

These subprograms provide access to quadruple-precision (`REAL*16`) `libm` functions and subroutines *(SPARC only).*

Intrinsic Functions

The following FORTRAN intrinsic functions return quadruple-precision values if they have quadruple-precision arguments. You need not put them in a type statement. If the function needed is available as an *intrinsic* function, it is simpler to use an intrinsic than a non-intrinsic function.

The ♦ indicates it is nonstandard that this is an intrinsic function.

`sqrt(x)`	`asin(x)`	`cosd(x)` ♦
`log(x)`	`acos(x)`	`asind(x)` ♦
`log10(x)`	`atan(x)`	`acosd(x)` ♦
`exp(x)`	`atan2(x,y)`	`atand(x)` ♦
`x**y`	`sinh(x)`	`atan2d(x,y)` ♦
`sin(x)`	`cosh(x)`	`aint(x)`
`cos(x)`	`tanh(x)`	`anint(x)`
`tan(x)`	`sind(x)` ♦	`nint(x)`

Non-Intrinsic Functions

In general, these do *not* correspond to standard generic *intrinsic* functions; data types are determined by the usual data typing rules.

Samples: Quadruple precision functions.

Note that the quadruple precision functions used are in a `REAL*16` statement.

```
      REAL*16 c, q_acosh, q_hypot, q_infinity, s, x, y, z
      ...
      z = q_acosh( x )
      i = iq_finite( x )
      z = q_hypot( x, y )
      z = q_infinity()
      CALL q_sincos( x, s, c )
```

Table 7-2 Quadruple-Precision `libm` Functions

The variables `c`, `l`, `p`, `s`, `u`, `x`, and `y` are of type quadruple precision.

If you use one of these quadruple precision functions, put it into a `REAL*16` statement (or type it by some `IMPLICIT` statement).

`sin`**`d`**`(x)`, `asin`**`d`**`(x)`, ... involve ***degrees*** rather than ***radians***

For meanings of routines and arguments, do a man on the routine name without the "q_"; it is a C man page for the double precision function, but the meanings are the same.

`q_copysign( x, y )`	`real*16`	Function
`q_fabs( x )`	`real*16`	Function
`q_fmod( x )`	`real*16`	Function
`q_infinity( )`	`real*16`	Function
`iq_finite( x )`	`integer`	Function
`iq_fp_class( x )`	`integer`	Function
`iq_ilogb( x )`	`integer`	Function
`iq_isinf( x )`	`integer`	Function
`iq_isnan( x )`	`integer`	Function
`iq_isnormal( x )`	`integer`	Function
`iq_issubnormal( x )`	`integer`	Function
`iq_iszero( x )`	`integer`	Function
`iq_signbit( x )`	`integer`	Function
`q_max_normal()`	`real*16`	Function
`q_max_subnormal()`	`real*16`	Function
`q_min_normal()`	`real*16`	Function
`q_min_subnormal()`	`real*16`	Function
`q_nextafter( x, y )`	`real*16`	Function
`q_quiet_nan( n )`	`real*16`	Function
`q_remainder( x, y )`	`real*16`	Function
`q_scalbn( x, n )`	`real*16`	Function
`q_signaling_nan( n )`	`real*16`	Function

If you need to use any other quadruple-precision `libm` function, you can call it using a "`$PRAGMA C(`*fcn*`)`" before the call. For details, read "The C-FORTRAN Interface" in the *FORTRAN User's Guide.*

7.35 libm_single: libm *Single-Precision Functions*

These subprograms provide access to single-precision libm functions and subroutines.

Intrinsic Functions

The following FORTRAN intrinsic functions return single-precision values if they have single-precision arguments. If the function needed is available as an *intrinsic* function, it may be simpler to use it than a *non-intrinsic* function.

The ♦ indicates it is nonstandard that this is an intrinsic function.

sqrt(x)	asin(x)	cosd(x) ♦
log(x)	acos(x)	asind(x) ♦
log10(x)	atan(x)	acosd(x) ♦
exp(x)	atan2(x,y)	atand(x) ♦
x**y	sinh(x)	atan2d(x,y) ♦
sin(x)	cosh(x)	aint(x)
cos(x)	tanh(x)	anint(x)
tan(x)	sind(x) ♦	nint(x)

Non-Intrinsic Functions

In general, the functions below provide access to single-precision libm functions that do *not* correspond to standard FORTRAN generic intrinsic functions—data types are determined by the usual data typing rules.

Samples: Single-precision libm functions.

Note that the REAL functions used are not in a REAL statement. Type is determined by the default typing rules for the letter "r".

```
      REAL c, s, x, y, z
      ..
      z = r_acosh( x )
      i = ir_finite( x )
      z = r_hypot( x, y )
      z = r_infinity()
      CALL r_sincos( x, s, c )
```

For meanings of routines and arguments, do a `man` on the routine name without the "`r_`"; it is a C man page, but the meanings are the same.

Variables `c`, `l`, `p`, `s`, `u`, `x`, and `y` are of type `REAL`.

If you use one of these `REAL` functions, it will get the default type of `REAL`, unless you have some `IMPLICIT` statement for variables starting with "`r`".

`sin`**`d`**`(x)`, `asin`**`d`**`(x)`, ... involve ***degrees*** rather than ***radians***

Table 7-3 Single-Precision `libm` Functions

`r_acos( x )`	`real`	Function	arc cosine
`r_acosd( x )`	`real`	Function	
`r_acosh( x )`	`real`	Function	arc cosh
`r_acosp( x )`	`real`	Function	
`r_acospi( x )`	`real`	Function	
`r_atan( x )`	`real`	Function	arc tangent
`r_atand( x )`	`real`	Function	
`r_atanh( x )`	`real`	Function	arc tanh
`r_atanp( x )`	`real`	Function	
`r_atanpi( x )`	`real`	Function	
`r_asin( x )`	`real`	Function	arc sine
`r_asind( x )`	`real`	Function	
`r_asinh( x )`	`real`	Function	arc sinh
`r_asinp( x )`	`real`	Function	
`r_asinpi( x )`	`real`	Function	
`r_atan2(( y, x )`	`real`	Function	arc tangent
`r_atan2d( y, x )`	`real`	Function	
`r_atan2pi( y, x )`	`real`	Function	
`r_cbrt( x )`	`real`	Function	cube root
`r_ceil( x )`	`real`	Function	ceiling
`r_copysign( x, y )`	`real`	Function	
`r_cos( x )`	`real`	Function	cosine
`r_cosd( x )`	`real`	Function	
`r_cosh( x )`	`real`	Function	hyperbolic cos
`r_cosp( x )`	`real`	Function	
`r_cospi( x )`	`real`	Function	
`r_erf( x )`	`real`	Function	error function
`r_erfc( x )`	`real`	Function	
`r_expm1( x )`	`real`	Function	(e**x)-1
`r_floor( x )`	`real`	Function	floor
`r_hypot( x, y )`	`real`	Function	hypotenuse
`r_infinity( )`	`real`	Function	bessel
`r_j0( x )`	`real`	Function	
`r_j1( x )`	`real`	Function	
`r_jn( x )`	`real`	Function	

Table 7-3 Single-Precision `libm` Functions *(Continued)*

`ir_finite( x )`	`integer`	Function	
`ir_fp_class( x )`	`integer`	Function	
`ir_ilogb( x )`	`integer`	Function	
`ir_irint( x )`	`integer`	Function	
`ir_isinf( x )`	`integer`	Function	
`ir_isnan( x )`	`integer`	Function	
`ir_isnormal( x )`	`integer`	Function	
`ir_issubnormal( x )`	`integer`	Function	
`ir_iszero( x )`	`integer`	Function	
`ir_signbit( x )`	`integer`	Function	
`r_addran()`	`real`	Function	random number
`r_addrans( x, p, l, u )`	`n/a`	Function	
`r_lcran()`	`real`	Subroutine	
`r_lcrans( x, p, l, u )`	`n/a`	Subroutine	
`r_shufrans(x, p, l, u)`	`n/a`	Subroutine	
`r_lgamma( x )`	`real`	Function	log gamma
`r_logb( x )`	`real`	Function	
`r_log1p( x )`	`real`	Function	
`r_log2( x )`	`real`	Function	
`r_max_normal()`	`real`	Function	
`r_max_subnormal()`	`real`	Function	
`r_min_normal()`	`real`	Function	
`r_min_subnormal()`	`real`	Function	
`r_nextafter( x, y )`	`real`	Function	
`r_quiet_nan( n )`	`real`	Function	
`r_remainder( x, y )`	`real`	Function	
`r_rint( x )`	`real`	Function	
`r_scalb( x, y )`	`real`	Function	
`r_scalbn( x, n )`	`real`	Function	
`r_signaling_nan( n )`	`real`	Function	
`r_significand( x )`	`real`	Function	
`r_sin( x )`	`real`	Function	sine
`r_sind( x )`	`real`	Function	
`r_sinh( x )`	`real`	Function	hyperbolic sin
`r_sinp( x )`	`real`	Function	
`r_sinpi( x )`	`real`	Function	
`r_sincos( x, s, c )`	`n/a`	Subroutine	sine & cosine
`r_sincosd( x, s, c )`	`n/a`	Subroutine	
`r_sincosp( x, s, c )`	`n/a`	Subroutine	
`r_sincospi( x, s, c )`	`n/a`	Subroutine	

Table 7-3 Single-Precision `libm` Functions *(Continued)*

`r_tan( x )`	`real`	Function	tangent
`r_tand( x )`	`real`	Function	
`r_tanh( x )`	`real`	Function	hyperbolic tan
`r_tanp( x )`	`real`	Function	
`r_tanpi( x )`	`real`	Function	
`r_y0( x )`	`real`	Function	bessel
`r_y1( x )`	`real`	Function	
`r_yn( n, x )`	`real`	Function	

See also: `intro`(3M) and the *Numerical Computation Guide.*

7.36 `link`, `symlnk`: *Make a Link to an Existing File*

Summary

`link`	Create a link to an existing file.
`symlnk`	Create a symbolic link to an existing file.

Functions

`status = link( name1, name2 )`			
`status = symlnk( name1, name2 )`			
`name1`	`character*`*`n`*	Input	Path name of an existing file
`name2`	`character*`*`n`*	Input	Path name to be linked to file *`name1`* *`name2`* must not already exist.
Return value	`integer`	Output	*`status`*=0: OK *`status`*>0: System error code

link: Create a Link to an Existing File

Example 1. `link`: Create a link named "`data1`" to file "`tlink.db.data.1`"

```
      character*34 name1/'tlink.db.data.1'/, name2/'data1'/
      integer link, status
      status = link( name1, name2 )
      if ( status .ne. 0 ) stop 'link: error'
      end
demo$ f77 -silent tlink.f
demo$ ls -l data1
data1 not found
demo$ a.out
demo$ ls -l data1
-rw-rw-r-- 2 generic 2 Aug 11 08:50 data1
demo$
```

symlnk: Create a Symbolic Link to an Existing File

Example 2. `symlnk`: Create a symbolic link named "`data1`" to the file "`tlink.db.data.1`"

```
      character*34 name1/'tlink.db.data.1'/, name2/'data1'/
      integer status, symlnk
      status = symlnk( name1, name2 )
      if ( status .ne. 0 ) stop 'symlnk: error'
      end
demo$ f77 -silent tsymlnk.f
demo$ ls -l data1
data1 not found
demo$ a.out
demo$ ls -l data1
lrwxrwxrwx 1 generic 15 Aug 11 11:09 data1 -> tlink.db.data.1
demo$
```

See also: `link`(2), `symlink`(2), `perror`(3F), `unlink`(3F).

Bug: Path names can be no longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

7.37 loc: *Return the Address of an Object*

Function

k = loc(*arg*)			
arg	Any type	Input	Name of any variable, array, or structure
Return value	integer	Output	Address of *arg*

Example: loc

```
      integer k, loc
      real arg / 9.0 /
      k = loc( arg )
      write(*,*) k
      end
```

7.38 long, short: *Integer Object Conversion*

long: *Convert a Short Integer to a Long Integer*

Function

call *ExpecLong*(long(*int2*))		
int2	integer*2	Input
Return value	integer*4	Output

short: *Convert a Long Integer to a Short Integer*

Function

call *ExpecShort*(short(*int4*))		
int4	integer*4	Input
Return value	integer*2	Output

Example (fragment): `long()` and `short()`.

```
      integer*4 int4/8/, long
      integer*2 int2/8/, short
      call ExpecLong( long(int2) )
      call ExpecShort( short(int4) )
      …
      end
```

`long` is useful if constants are used in calls to library routines and the code is compiled with the `-i2` option.

`short` is useful in similar context when an otherwise long object must be passed as a short integer.

7.39 longjmp, isetjmp: Return to location set by isetjmp

isetjmp: *Set the location for longjmp*

Function

ival = isetjmp(*env*)			
env	integer env(12)	Output	*env* is a 12 word integer array
Return value	integer	Output	*ival* = 0 if isetjmp is called explicitly *ival* ≠ 0 if isetjmp is called through longjmp

longjmp: *Return to the location set by* isetjmp

Subroutine

call longjmp(*env*, *ival*)			
env	integer env(12)	Input	*env* is the 12 word integer array initialized by isetjmp
ival	integer	Output	*ival* = 0 if isetjmp is called explicitly *ival* ≠ 0 if isetjmp is called through longjmp

Description

The isetjmp and longjmp routines are used to deal with errors and interrupts encountered in a low-level routine of a program.

These routines should be used only as a last resort. They require discipline. They are not portable. Read the man page setjmp (3V) for bugs and other details.

`isetjmp` saves the stack environment in *env*. It also saves the register environment.

`longjmp` restores the environment saved by the last call to `isetjmp` and returns in such a way that execution continues as if the call to `isetjmp` had just returned the value *ival*.

The integer expression *ival* returned from `isetjmp` is zero if `longjmp` is not called, and it is nonzero if `longjmp` is called.

Example: Code fragment using `isetjmp` and `longjmp`.

```
      integer  env(12)
      common /jmpblk/ env
      j = isetjmp( env )                    ! <-- isetjmp
      if ( j .eq. 0 ) then
           call  sbrtnA
      else
           call error_processor
      end if
      end
      subroutine sbrtnA
      integer  env(12)
      common /jmpblk/ env
      call longjmp( env, ival )                  ! <-- longjmp
      return
      end
```

Restrictions

- You must invoke `isetjmp` before calling `longjmp()`.
- The argument to `isetjmp` must be a 12 integer array.
- You must pass the *env* variable from the routine that calls `isetjmp` to the routine that calls `longjmp`, either by common or as an argument.
- `longjmp` attempts to clean up the stack. `longjmp` must be called from a lower call-level than `isetjmp`.
- Passing `isetjmp` as an argument that is a procedure name does not work.

See `setjmp` (3V).

7.40 `malloc`: *Allocate Memory and Get Address*

Function

k = malloc(*n*)			
n	integer	Input	Number of bytes of memory
Return value	integer	Output	*k*>0: *k*=address of *the start of the block of memory allocated* *k*=0: Error

The function `malloc` allocates an area of memory and returns the address of the start of that area. The region of memory is not initialized in any way — assume it is garbage.

Example (fragment): `malloc()`.

```
      pointer ( p1, X )
      …
      p1 = malloc( 1000 )
      if ( p1 .eq. 0 ) stop 'malloc: cannot allocate'
      …
      end
```

In the above example, we get 1000 bytes of memory.

See also Section 7.15, “free: Deallocate Memory Allocated by Malloc,” for more detail.

7.41 mvbits: *Move a Bit Field*

call mvbits(*src*, *ini1*, *nbits*, *des*, *ini2*)			
src	INTEGER	Input	Source
ini1	INTEGER	Input	Initial bit position in the source
nbits	INTEGER	Input	Number of bits to move
des	INTEGER	Output	Destination
ini2	INTEGER	Input	Initial bit position in the destination

Example: mvbits

```
demo$ cat mvb1.f
* mvb1.f -- From src, initial bit 0, move 3 bits to des, initial bit 3.
*    src    des
* 543210 543210 <-- Bit numbers (VMS convention)
* 000111 000001 <-- Values before move
* 000111 111001 <-- Values after move
     integer src, ini1, nbits, des, ini2
     data src, ini1, nbits, des, ini2
&         / 7,    0,     3,   1,    3 /
     call mvbits ( src, ini1, nbits, des, ini2 )
     write (*,"(5o3)") src, ini1, nbits, des, ini2
     end
demo$ f77 -silent mvb1.f -lV77                    ← Note the -lV77
demo$ a.out
  7  0  3 71  3
demo$
```

To get mvbits, use -lV77. If you use idate or time, you get VMS versions.

Remarks:

- Bits are numbered according to VMS convention: from low-ordered end (as in the example above).
- MVBITS changes only bits *ini2* through *ini2*+*nbits*-1 of the *des* location, and no bits of the *src* location.
- Restrictions
  - *ini1* + *nbits* ≤ 32
  - *ini2* + *nbits* ≤ 32

7.42 perror, gerror, ierrno: *Get System Error Messages*

perror	Print a message to FORTRAN logical unit 0, stderr.
gerror	Get a system error message (of the last detected system error)
ierrno	Get the error number of the last detected system error.

perror: *Print Message to Logical Unit 0, Stderr*

Subroutine

call perror(*string*)			
string	character*n	Input	The message. It will be written preceding the standard error message. It is for the last detected system error.

Example 1:

```
…
      call perror( "file is for formatted I/O" )
…
```

gerror: Get Message for Last Detected System Error

Subroutine or function

call gerror(*string*)			
string	character*n	Output	Message for the last detected system error

Example 2: gerror() as a subroutine.

```
      character string*30
      …
      call gerror ( string )
      write(*,*) string
      end
```

Example 3: `gerror`() as a function (In this case, *string* is not used.)

```
      character gerror*30, z*30
      …
      z = gerror( )
      write(*,*) z
      end
```

ierrno: Get Number for Last Detected System Error

Function

n = ierrno()			
Return value	integer	Output	Error number of last detected system error

This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

Example 4: `ierrno()`.

```
      integer ierrno, n
      …
      n = ierrno()
      write(*,*) n
      end
```

See also `intro`(2), `perror`(3).

Bugs:

- *`string`* in the call to `perror` can be no longer than 127 characters.
- The length of the string returned by `gerror` is determined by the calling program.

f77 *I/O Error Codes and Meanings*

If the error number is less than 1000, then it is a *system* error. See intro (2).

```
1000   ‘‘error in format’’
1001   ‘‘illegal unit number’’
1002   ‘‘formatted io not allowed’’
1003   ‘‘unformatted io not allowed’’
1004   ‘‘direct io not allowed’’
1005   ‘‘sequential io not allowed’’
1006   ‘‘can’t backspace file’’
1007   ‘‘off beginning of record’’
1008   ‘‘can’t stat file’’
1009   ‘‘no * after repeat count’’
1010   ‘‘off end of record’’
1011     <not used>
1012   ‘‘incomprehensible list input’’
1013   ‘‘out of free space’’
1014   ‘‘unit not connected’’
1015   ‘‘read unexpected character’’
1016   ‘‘illegal logical input field’’
1017   ‘’’new’ file exists’’
1018   ‘‘can’t find ’old’ file’’
1019   ‘‘unknown system error’’
1020   ‘‘requires seek ability’’
1021   ‘‘illegal argument’’
1022   ‘‘negative repeat count’’
1023   ‘‘illegal operation for unit’’
1024     <not used>
1025   ‘‘incompatible specifiers in open’’
1026   ‘‘illegal input for namelist’’
1027   ‘‘error in FILEOPT parameter’’
```

7.43 putc, fputc: *Write a Character to a Logical Unit*

`putc` writes to logical unit 6, normally the control terminal output.

`fputc` writes to a logical unit.

These functions write a character to the file associated with a FORTRAN logical unit bypassing normal FORTRAN I/O.

For any one unit, do not mix normal FORTRAN output with output by these functions.

putc: *Write to Logical Unit 6*

Function

status = putc(*char*)			
char	character	Input	The character to write to the unit
Return value	integer	Output	*status*=0: OK *status*>0: System error code

Example: `putc()`.

```
      character char, s*10 / 'OK by putc' /
      integer putc, status
      do i = 1, 10
           char = s(i:i)
           status = putc( char )
      end do
      status = putc( '\n' )
      end
demo$ f77 -silent tputc.f
demo$ a.out
OK by putc
demo$
```

fputc: *Write to Specified Logical Unit*

Function

status = fputc(*lunit*, *char*)			
lunit	*integer*	Input	The unit to write to
char	*character*	Input	The character to write to the unit
Return value	*integer*	Output	*status*=0: OK *status*>0: System error code

Example: fputc().

```
      character char, s*11 / 'OK by fputc' /
      integer fputc, status
      open( 1, file='tfputc.data')
      do i = 1, 11
           char = s(i:i)
           status = fputc( 1, char )
      end do
      status = fputc( 1, '\n' )
      end
demo$ f77 -silent tfputc.f
demo$ a.out
demo$ cat tfputc.data
OK by fputc
demo$
```

See also putc(3S), intro(2), perror(3F).

7.44 `qsort`: *Sort the Elements of a One-dimensional Array*

Subroutine

`call qsort(` *`array`*`,` *`len`*`,` *`isize`*`,` *`compar`* `)`			
`array`	`array`	Input	Contains the elements to be sorted
`len`	`integer`	Input	Number of elements in the array.
`isize`	`integer`	Input	Size of an element, typically: 4 for integer or real 8 for double precision or complex 16 for double complex Length of character object for character arrays
`compar`	function name	Input	Name of a user supplied `integer*2` function

The function `compar(`*`arg1`*`,`*`arg2`*`)` determines the sorting order. The two arguments are elements of *array*. The function must return:

Negative	If *`arg1`* is considered to precede *`arg2`*
Zero	If *`arg1`* is equivalent to *`arg2`*
Positive	If *`arg1`* is considered to follow *`arg2`*

Example: `qsort()`.

```
      external compar
      integer*2 compar
      integer array(10)/5,1,9,0,8,7,3,4,6,2/, len/10/, isize/4/
      call qsort( array, len, isize, compar )
      write(*,'(10i3)') array
      end

      integer*2 function compar( a, b )
      integer a, b
      if ( a .lt. b ) compar = -1
      if ( a .eq. b ) compar = 0
      if ( a .gt. b ) compar = 1
      return
      end
```

Compile and run of the above source:

```
demo$ f77 -silent tqsort.f
demo$ a.out
  0 1 2 3 4 5 6 7 8 9
demo$
```

See also `qsort`(3).

7.45 `ran`*: Generate a Random Number between 0 and 1*

Repeated calls to `ran` generate a sequence of random numbers with a uniform distribution.

r = ran(*i*)			
i	INTEGER*4	Input	Variable or array element
r	REAL	Output	Variable or array element

To use this VMS routine you need `-lV77`. If you use `-lV77` and invoke `idate()` or `time()`, then you get the VMS version.

This is an extremely poor algorithm. See `lcrans`(3m) instead.

Example: ran.

```
demo$ cat ran1.f
* ran1.f -- Generate random numbers.
      integer i, n
      real r(10)
      i = 760013
      do n = 1, 10
           r(n) = ran ( i )
      end do
      write ( *, "( 5 f11.6 )" ) r
      end
demo$ f77 -silent ran1.f -lV77
demo$ a.out
   0.222058 0.299851 0.390777 0.607055 0.653188
   0.060174 0.149466 0.444353 0.002982 0.976519
demo$
```

Remarks:

- The range includes 0.0 and excludes 1.0.
- The algorithm is a multiplicative, congruential type, general random number generator.
- In general, the value of i is set *once* during execution of the calling program.
- The initial value of i should be a large odd integer.
- Each call to RAN gets the next random number in the sequence.
- To get a different sequence of random numbers each time you run the program, you must set the argument to a different initial value for each run.
- The argument is used by RAN to store a value for the calculation of the next random number according to the following algorithm:

```
      SEED = 6909 * SEED + 1 (MOD 2**32)
```

- SEED contains a 32-bit number, and the high-order 24 bits are converted to floating point, and that value is returned.

7.46 rand, drand, irand: *Return Random Values*

Summary:

- `rand` returns real values in the range 0.0 through 1.0.
- `drand` returns double precision values in the range 0.0 through 1.0.
- `irand` returns positive integers in the range 0 through 2147483647.

These functions use `random`(3) to generate sequences of random numbers. The three functions share the same 256 byte state array. The only advantage of these functions is that they are widely available on UNIX systems. For better random number generators, compare `lcrans`, `addrans`, and `shufrans`, read the *Numerical Computation Guide.*

i = irand(*k*)			
r = rand(*k*)			
d = drand(*k*)			
k, r, d	integer*4	Input	*k*=0: Get next random number in the sequence *k*=1: Restart sequence, return first number *k*>0: Use as a seed for new sequence, return first number
rand	real*4	Output	
drand	real*8	Output	
irand	integer*4	Output	

Example: `irand()`.

```
      integer*4 v(5), iflag/0/
      do i = 1, 5
           v(i) = irand( iflag )
      end do
      write(*,*) v
      end
demo$ f77 -silent trand.f
demo$ a.out
   2078917053 143302914 1027100827 1953210302 755253631
demo$
```

See also `random`(3).

7.47 rename: *Rename a File*

Function

status = rename(*from*, *to*)			
from	character*n	Input	Path name of an existing file
to	character*n	Input	New path name for the file
Return value	integer	Output	*status*=0: OK *status*>0: System error code

If *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same filesystem.If *to* exists, it will be removed first.

Example: rename() – Rename file "trename.old" to "trename.new"

```
      integer rename, status
      character*18 from/'trename.old'/, to/'trename.new'/
      status = rename( from, to )
      if ( status .ne. 0 ) stop 'rename: error'
      end
demo$ f77 - silent trename.f
demo$ ls trename*
trename.f trename.old
demo$ a.out
demo$ ls trename*
trename.f trename.new
demo$
```

See also rename(2), perror(3F).

Bug: Path names can be no longer than MAXPATHLEN as defined in <sys/param.h>.

7.48 secnds: *Get System Time in Seconds, Minus Argument*

t = secnds(*t0*)			
t0	REAL	Input	Constant, variable, or array element
Return Value	REAL	Output	Number of seconds since midnight, minus *t0*

Example: secnds.

```
demo$ cat sec1.f
      real elapsed, t0, t1, x, y
      t0 = 0.0
      t1 = secnds( t0 )
      y = 0.1
      do i = 1, 1000
           x = asin( y )
      end do
      elapsed = secnds( t1 )
      write ( *, 1 ) elapsed
1     format ( ' 1000 arcsines: ', f12.6, ' sec' )
      end
demo$ f77 -silent sec1.f -lV77
demo$ a.out
 1000 arcsines: 6.699141 sec
demo$
```

To use this VMS routine you need `-lV77`. If you use `-lV77` and invoke `idate()` or `time()`, then you get the VMS version.

Remarks:

- The returned value from SECNDS is accurate to 0.01 second.
- The value is the system time, as the number of seconds from midnight, and it correctly spans midnight.
- Some precision may be lost for small time intervals near the end of the day.

7.49 sh: *Fast Execution of an* sh *Command*

Function

status = sh(*string*)			
string	character*n	Input	String containing command to do
Return value	integer	Output	Exit status of the shell executed. See wait(2) for an explanation of this value.

Example: sh().

```
      character*18 string / 'ls > MyOwnFile.names' /
      integer status, sh
      status = sh( string )
      if ( status .ne. 0 ) stop 'sh: error'
      ...
      end
```

The function sh passes *string* to the sh shell as input, as if the string had been typed as a command.

The current process waits until the command terminates.

The forked process flushes all open files.

- For output files, the buffer is flushed to the actual file.
- For input files, the position of the pointer is unpredictable.

The sh() function is not mt-safe. Do not call it from multi-threaded programs, that is do not call it from FORTRAN MP programs.

See also: execve(2), wait(2), system(3).

Bug: *string* can not be longer than 1024 characters.

7.50 `signal`: *Change the Action for a Signal*

Function

n = signal(*signum*, *proc*, *flag*)			
signum	integer	Input	Signal number. See signal(3)
proc	Routine name	Input	Name of user signal handling routine (must be in an external statement)
flag	integer	Input	*flag*<0: Use *proc* as the signal handling routine *flag*≥0: Ignore *proc*; pass *flag* as the action *flag*=0: Use the default action *flag*=1: Ignore this signal
Return value	integer	Output	*n*=-1: System error *n*>0: Definition of previous action *n*>1: *n*=Address of routine that would have been called *n*<-1: If *signum* is a valid signal number, then *n*=address of routine that would have been called. If *signum* is a *not* a valid signal number, then *n* is an error number.

If *proc* is called, it will be passed the signal number as an integer argument.

If a process incurs a signal, the default action is usually to clean up and abort. You can change the action by writing an alternative signal handling routine, and then telling the system to use it.

You tell the system to use alternate action by calling `signal`.

The returned value can be used in subsequent calls to `signal` in order to restore a previous action definition.

Note that you can get a negative return value even though there is no error. In fact, if you pass a *valid* signal number to `signal()` and you get a return value less than -1, then it is OK.

`f77` arranges to trap certain signals when a process is started. The only way to restore the default `f77` action is to save the returned value from the first call to `signal`.

Example (fragment): `signal()`—If illegal instruction signal, then call `MyAct`.

```
#include <signal.h>
      integer flag/-1/, n, signal
      external MyAct
      …
      n = signal( SIGILL, MyAct, flag )
      if ( n .eq. -1 ) stop 'Error from signal()'
      if ( n .lt. -1 ) write(*,*) 'From signal: n = ', -n
      …
      end

      subroutine MyAct( signum )
      integer signum
      …
      return
      end
```

See also `kill`(1), `signal`(3), `kill`(3F).

7.51 `sleep`*: Suspend Execution for an Interval*

Subroutine

`subroutine sleep(` *`itime`* `)`			
`itime`	`integer`	Input	Number of seconds to sleep

The actual time can be up to 1 second less than *`itime`* due to granularity in system timekeeping.

Example: `sleep()`.

```
      integer time / 5 /
      write(*,*) 'Start'
      call sleep( time )
      write(*,*) 'End'
      end
```

See also `sleep`(3).

7.52 `stat`, `lstat`, `fstat`: *Get File Status*

These functions return the following information:

> device, inode's number, protection, number of hard links,
> user ID, group ID, device type, size, access time, modify time,
> status change time, optimal blocksize, blocks allocated

Both `stat` and `lstat` query by file name. `fstat` queries by logical unit.

`stat`: *Get Status for File, by File Name*

Function

ierr = stat (*name*, *statb*)			
name	character*n	Input	Name of the file
statb	integer	Output	Status structure for the file, 13-element array
Return value	integer	Output	*ierr*=0: OK *ierr*>0: Error code

Example 1: `stat()`.

```
      character name*18 /'MyFile'/
      integer ierr, stat, lunit/1/, statb(13)
      open( unit=lunit, file=name )
      ierr = stat ( name, statb )
      if ( ierr .ne. 0 ) stop 'stat: error'
      write(*,*)'UID of owner = ',statb(5),', blocks = ',statb(13)
      end
```

`fstat` *Get Status for File, by Logical Unit*

Function

ierr = fstat (*lunit*, *statb*)			
lunit	integer	Input	Logical unit number
statb	integer	Output	Status structure for the file, 13-element array
Return value	integer	Output	*ierr*=0: OK *ierr*>0: Error code

Example 2: `fstat()`.

```
      character name*18 /'MyFile'/
      integer fstat, lunit/1/, statb(13)
      open( unit=lunit, file=name )
      ierr = fstat ( lunit, statb )
      if ( ierr .ne. 0 ) stop 'fstat: error'
      write(*,*)'UID of owner = ',statb(5),', blocks = ',statb(13)
      end
```

`lstat`: *Get Status for File, by File Name*

Function

ierr = lstat (*name*, *statb*)			
name	character*n	Input	File name
statb	integer	Output	Status array of file, 13 elements
Return value	integer	Output	*ierr*=0: OK *ierr*>0: Error code

Example 3: `lstat()`.

```
      character name*18 /'MyFile'/
      integer lstat, lunit/1/, statb(13)
      open( unit=lunit, file=name )
      ierr = lstat ( name, statb )
      if ( ierr .ne. 0 ) stop 'lstat: error'
      write(*,*)'UID of owner = ',statb(5),', blocks = ',statb(13)
      end
```

Detail of Status Array for Files

The meaning of the information returned in array *statb* is as described for the structure *stat* under `stat`(2).

Spare values are not included. The order is shown below:

```
statb(1)    Device inode resides on
statb(2)    This inode's number
statb(3)    Protection
statb(4)    Number of hard links to the file
statb(5)    User ID of owner
statb(6)    Group ID of owner
statb(7)    Device type, for inode that is device
statb(8)    Total size of file
statb(9)    File last access time
statb(10)   File last modify time
statb(11)   File last status change time
statb(12)   Optimal blocksize for file system I/O ops
statb(13)   Actual number of blocks allocated
```

See also `stat`(2), `access`(3F), `perror`(3F), `time`(3F).

Bug: Path names can be no longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

7.53 system: *Execute a System Command*

Function

status = system(*string*)			
string	character*n	Input	String containing command to do
Return value	integer	Output	Exit status of the shell executed. See wait(2) for an explanation of this value.

Example: system().

```
      character*8 string / 'ls s*' /
      integer status, system
      status = system( string )
      if ( status .ne. 0 ) stop 'system: error'
      end
```

The function system passes *string* to your shell as input, as if the string had been typed as a command.

If system can find the environment variable SHELL, then system uses the value of SHELL as the command interpreter (shell), otherwise it uses sh(1).

The current process waits until the command terminates.

Historically cc and f77 developed with different assumptions:

- If cc calls system, the shell is always the Bourne shell.
- If f77 calls system, then which shell gets called depends on the environment variable SHELL.

The system function flushes all open files.

- For output files, the buffer is flushed to the actual file.
- For input files, the position of the pointer is unpredictable.

See also: execve(2), wait(2), system(3).

The system() function is not mt-safe. Do not call it from multi-threaded programs, that is do not call it from FORTRAN MP programs.

Bug: *string* can not be longer than 1024 characters.

7.54 `time`, `ctime`, `ltime`, `gmtime`: *Get System Time*

`time`	Get system time as integer (seconds since 0 GMT 1/1/70).
`ctime`	Convert a system time to an ASCII string.
`ltime`	Dissect a system time into month, day, and so forth, local time.
`gmtime`	Dissect a system time into month, day, and so forth, GMT.

Alternate:

`time`	VMS Version: Get the system time as character (hh:mm:ss).

`time`: *Get System Time*

For `time`() there are two versions, a standard version and a VMS version. If you use the `f77` command-line option `-lV77`, then you get the VMS version for `time()` and for `idate()`; otherwise you get the standard versions.

`time()`—*Version Standard with Operating System*

Function

n = `time()`			
Return value	`integer`	Output	Time, in seconds, since 0:0:0, GMT, 1/1/70

The function `time()` returns an integer with the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the operating system clock.

Example: `time()`, version standard with the operating system.

```
      integer  n, time
      n = time()
      write(*,*) 'Seconds since 0 1/1/70 GMT = ', n
      end
demo$ f77 -silent ttime.f
demo$ a.out
 The time is:  771967850
demo$
```

Note: Do not use `-lV77`.

time()—*VMS Version*

This function time gets the current system time as a character string.

Function

call time(*t*)			
t	character*8	Output	Time, in the form *hh:mm:ss* *hh*, *mm*, *ss* are each 2-digits *hh* is the hour *mm* is the minute *ss* is the second

Example: time(*t*), VMS version, ctime, convert System Time to ASCII.

```
      character  t*8
      call time( t )
      write(*, "(' The current time is ', A8 )")  t
      end
demo$ f77 -silent ttimeV.f -lV77
demo$ a.out
 The current time is 08:14:13
demo$
```

Note: Use - lV77.

ctime: *Convert System Time to Character*

The function ctime converts a system time, *stime*, and returns it as a 24-character ASCII string.

Function

string = ctime(*stime*)			
stime	integer*4	Input	System time from time() (standard version)
Return value	character*24	Output	System time as character string. You must type ctime and *string* as character*24.

The format of the `ctime` returned value is shown in the example below. It is described in the `man` page `ctime`, section 3C in Solaris 2.x, 3V in Solaris 1.x.

Example: `ctime()`.

```
      character*24 ctime, string
      integer  n, time
      n = time()
      string = ctime( n )
      write(*,*) 'ctime: ', string
      end
demo$ f77 -silent tctime.f
demo$ a.out
 ctime: Mon Aug 12 10:35:38 1991
demo$
```

`ltime`: *Split System Time to Month, Day,... (Local)*

This dissects a system time into month, day, and so forth, for local time zone.

Subroutine

call ltime(*stime*, *tarray*)			
stime	integer*4	Input	System time from `time()` (standard version)
tarray	integer*4(9)	Output	System time, local, as day, month, year, ...

For the meaning of the elements in `tarray`, see.“tarray() Values,” page 404.

Example: `ltime()`.

```
      integer*4  stime, tarray(9), time
      stime = time()
      call ltime( stime, tarray )
      write(*,*) 'ltime: ', tarray
      end
demo$ f77 -silent tltime.f
demo$ a.out
 ltime: 25 49 10 12 7 91 1 223 1
demo$
```

7

gmtime: *Split System Time to Month, Day, … (GMT)*

This dissects a system time into month, day, etc, for GMT.

Subroutine

call gmtime(*stime*, *tarray*)			
stime	integer*4	Input	System time from time() (standard version)
tarray	integer*4(9)	Output	System time, GMT, as day, month, year, …

For the meaning of the elements in tarray, see.“tarray() Values,” below.

Example: gmtime.

```
     integer*4  stime, tarray(9), time
     stime = time()
     call gmtime( stime, tarray )
     write(*,*) 'gmtime: ', tarray
     end
demo$ f77 -silent tgmtime.f
demo$ a.out
 gmtime:   12  44  19  18  5  94  6  168  0
demo$
```

tarray() *Values*

The tarray() values, from ctime: Index, units, range.

tarray()

1	Seconds (0 - 61)	6	Year - 1900
2	Minutes (0 - 59)	7	Day of week (Sunday = 0)
3	Hours (0 - 23)	8	Day of year (0 - 365)
4	Day of month (1 - 1)	9	Daylight Saving Time, 1 if DST in effect
5	Months since January (0 - 11)		

For Solaris1.x, the range for seconds is (0 - 59)

These are described in the man page ctime, section 3C in Solaris 2.x, 3V in Solaris 1.x. See also: ctime, idate(3F), fdate(3F).

7.55 topen, tclose, tread,..., tstate: *Do Tape I/O*

You can manipulate magnetic tape from FORTRAN using these functions:

topen	Associate a device name with a tape logical unit.
tclose	Write EOF, close tape device channel, and remove association with *tlu*.
tread	Read next physical record from tape into buffer.
twrite	Write the next physical record from buffer to tape.
trewin	Rewind the tape to the beginning of the first data file.
tskipf	Skip forward over files and/or records, and reset EOF status.
tstate	Determine the logical state of the tape I/O channel.

On any one unit, do not mix these functions with standard FORTRAN I/O.

You must first use topen() to open a *tape logical unit*, *tlu*, for the specified device. Then you do all other operations on the specified *tlu*. The *tlu* has no relationship at all to any normal FORTRAN logical unit.

Note that before you use one of these functions, its name must be in an INTEGER type statement.

topen: *Associate a Device with a Tape Logical Unit*

n = topen(*tlu*, *devnam*, *islabeled*)			
tlu	integer	Input	Tape logical unit. It must be in the range 0 to 7.
islabeled	logical	Input	true=the tape is labeled Note: a label is the first file on the tape.
Return value	integer	Output	*n*=0: OK *n*<0: Error.

This does *not* move tape. See perror(3f) for details.

Example: `topen()`–Open a 1/4" tape file.

```
      CHARACTER devnam*9 / '/dev/rst0' /
      INTEGER n / 0 /, tlu / 1 /, topen
      LOGICAL islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      IF ( n .LT. 0 ) STOP "topen: cannot open"
      WRITE(*,'("topen ok:", 2I3, 1X, A10)') n, tlu,  devnam
      END
```

The displayed output is

```
 topen ok: 0 1 /dev/rst0
```

`tclose`: *Write Eof, Close Tape Channel, Disconnect tlu*

n = tclose (*tlu*)			
tlu	integer	Input	Tape logical unit, in range 0 to 7.
n	integer	Return value	*n*=0: OK *n*<0: Error

Caution: `tclose()` places an EOF marker immediately after the current location of the unit pointer and then closes the unit. So if you `trewin()` a unit before you use `tclose()` it, its contents are thrown away.

Example: `tclose()`–Close an opened 1/4" tape file.

```
      CHARACTER devnam*9 / '/dev/rst0' /
      INTEGER n / 0 /, tlu / 1 /, tclose, topen
      LOGICAL islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      n = tclose( tlu )
      IF ( n .LT. 0 ) STOP "tclose: cannot close"
      WRITE(*, '("tclose ok:", 2I3, 1X, A10)')  n, tlu,  devnam
      END
```

The displayed output is:

```
 tclose ok: 0 1  /dev/rst0
```

`twrite`: *Write Next Physical Record to Tape*

n = twrite(*tlu*, *buffer*)			
tlu	integer	Input	Tape logical unit, in range 0 to 7.
buffer	character	Input	Must be sized at a multiple of 512
n	integer	Return value	*n*>0: OK, and *n* = the number of bytes written *n*=0: End of Tape *n*<0: Error

The physical record length will be the size of `buffer`.

Example: `twrite()`–Write a 2-record file.

```
      CHARACTER devnam*9 / '/dev/rst0' /, rec1*512 / "abcd" /,
     &             rec2*512 / "wxyz" /
      INTEGER n / 0 /, tlu / 1 /, tclose, topen, twrite
      LOGICAL islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      IF ( n .LT. 0 ) STOP "topen: cannot open"
      n = twrite( tlu, rec1 )
      IF ( n .LT. 0 ) STOP "twrite: cannot write 1"
      n = twrite( tlu, rec2 )
      IF ( n .LT. 0 ) STOP "twrite: cannot write 2"
      WRITE(*, '("twrite ok:", 2I4, 1X, A10)')  n, tlu, devnam
      END
```

The displayed output is:

```
 twrite ok: 512 1 /dev/rst0
```

7

`tread`: *Read Next Physical Record from Tape*

n = tread(*tlu*, *buffer*)			
tlu	integer	Input	Tape logical unit, in range 0 to 7.
buffer	character	Input	Must be sized at a multiple of 512, and must be large enough to hold the largest physical record to be read.
n	integer	Return value	*n*>0: OK, and *n* is the number of bytes read. *n*<0: error. *n*=0: EOF

If the tape is at EOF or EOT, then tread does a return; it does not read tape.

Example: `tread()`–Read the first record of the file written above.

```
      CHARACTER devnam*9 / '/dev/rst0' /, onerec*512 / " " /
      INTEGER n / 0 /, tlu / 1 /, topen, tread
      LOGICAL islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      IF ( n .LT. 0 ) STOP "topen: cannot open"
      n = tread( tlu, onerec )
      IF ( n .LT. 0 ) STOP "tread: cannot read"
      WRITE(*,'("tread ok:", 2I4, 1X, A10)')  n, tlu,  devnam
      WRITE(*,'( A4)')  onerec
      END
```

The displayed output is:

```
tread ok: 512 1 /dev/rst0
abcd
```

`trewin`: *Rewind Tape to Beginning of First Data File*

n = trewin (*tlu*)			
tlu	integer	Input	Tape logical unit, in range 0 to 7.
n	integer	Return value	*n*=0: OK *n*<0: Error.

If the tape is labeled then the label is skipped over after rewinding.

Example 1: `trewin()`–Typical fragment.

```
      CHARACTER devnam*9 / '/dev/rst0' /
      INTEGER n /0/, tlu /1/, tclose, topen, tread, trewin
      …
      n = trewin( tlu )
      IF ( n .LT. 0 ) STOP "trewin: cannot rewind"
      WRITE(*, '("trewin ok:", 2I4, 1X, A10)') n, tlu, devnam
      …
      END
```

Example 2: `trewin()`–In a 2-record file, try to read 3 records; rewind; read 1 record.

```
      CHARACTER devnam*9 / '/dev/rst0' /, onerec*512 / " " /
      INTEGER n / 0 /, r, tlu / 1 /, topen, tread, trewin
      LOGICAL islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      IF ( n .LT. 0 ) STOP "topen: cannot open"
      DO r = 1, 3
         n = tread( tlu, onerec )
         WRITE(*,'(1X, I2, 1X, A4)')  r, onerec
      END DO
      n = trewin( tlu )
      IF ( n .LT. 0 ) STOP "trewin: cannot rewind"
      WRITE(*, '("trewin ok:" 2I4, 1X, A10)')  n, tlu, devnam
      n = tread( tlu, onerec )
      IF ( n .LT. 0 ) STOP "tread: cannot read after rewind"
      WRITE(*,'(A4)')  onerec
      END
```

The displayed output is:

```
1 abcd
2 wxyz
3 wxyz
trewin ok: 0 1 /dev/rst0
abcd
```

`tskipf`: *Skip Files and Records; Reset EoF Status*

n = tskipf(*tlu*, *nf*, *nr*)			
tlu	integer	Input	Tape logical unit, in range 0 to 7.
nf	integer	Input	Number of end-of-file marks to skip over first
nr	integer	Input	Number of physical records to skip over after skipping files
n	integer	Return value	*n*=0: OK *n*<0: Error.

It does *not* skip backward.

First it skips forward over *nf* end-of-file marks. Then it skips forward over *nr* physical records. If the current file is at EOF, this counts as 1 file to skip. This also resets the EOF status. Compare `tstate` below.

Example: `tskipf()`–Typical fragment, skip 4 files and then skip 1 record. Compare `tstate`, second example.

```
      INTEGER nfiles / 4 /, nrecords / 1 /, tskipf, tlu / 1 /
      …
      n = tskipf( tlu, nfiles, nrecords )
      IF ( n .LT. 0 ) STOP "tskipf: cannot skip"
      …
```

`tstate`: *Get Logical State of Tape I/O Channel*

n = tstate(*tlu*, *fileno*, *recno*, *errf*, *eoff*, *eotf*, *tcsr*)			
tlu	integer	Input	Tape logical unit, in range 0 to 7.
fileno	integer	Output	Current file number
recno	integer	Output	Current record number
errf	logical	Output	true=an error occurred
eoff	logical	Output	true=the current file is at EOF
eotf	logical	Output	true=tape has reached logical end-of-tape
tcsr	integer	Output	true=hardware errors on the device. It contains the tape drive control status register. If the error is software, then *tcsr* is returned as zero. The values returned in this status register vary grossly with the brand and size of tape drive.

For details, see `st`(4s).

While *eoff* is true, you cannot read from that *tlu*. You can set this EOF status flag to false by using `tskipf()` to skip one file and zero records:

```
      n = tskipf( tlu, 1, 0).
```

Then you can read any valid record that follows.

End-Of-Tape (EOT) is indicated by an empty file, often referred to as a double EOF mark. You cannot read past EOT, but you can write past EOT.

Example: Write 3 files of 2 records each. The next example uses `tstate()` to trap EOF and get at all files.

```
      CHARACTER devnam*10 / '/dev/nrst0' /,
&                f0rec1*512 / "eins" /, f0rec2*512 / "zwei" /,
&                f1rec1*512 / "ichi" /, f1rec2*512 / "ni__" /,
&                f2rec1*512 / "un__" /, f2rec2*512 / "deux" /
      INTEGER n / 0 /, tlu / 1 /, tclose, topen, trewin, twrite
      LOGICAL islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      n = trewin( tlu )
      n = twrite( tlu, f0rec1 )
      n = twrite( tlu, f0rec2 )
      n = tclose( tlu )
      n = topen( tlu, devnam, islabeled )
      n = twrite( tlu, f1rec1 )
      n = twrite( tlu, f1rec2 )
      n = tclose( tlu )
      n = topen( tlu, devnam, islabeled )
      n = twrite( tlu, f2rec1 )
      n = twrite( tlu, f2rec2 )
      n = tclose( tlu )
      END
```

Example: Use `tstate()` in a loop that reads all records of the 3 files written in the previous example.

```
      CHARACTER devnam*10 / '/dev/nrst0' /, onerec*512 / " " /
      INTEGER f, n / 0 /, tlu / 1 /, tcsr, topen, tread,
&          trewin, tskipf, tstate
      LOGICAL errf, eoff, eotf, islabeled / .false. /
      n = topen( tlu, devnam, islabeled )
      n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
      WRITE(*,1)  'open:', fn, rn, errf, eoff, eotf, tcsr
1     FORMAT(1X, A10, 2I2, 1X, 1L, 1X, 1L,1X, 1L, 1X, I2 )
2     FORMAT(1X, A10,1X,A4,1X,2I2,1X,1L,1X,1L,1X,1L,1X,I2)
      n = trewin( tlu )
      n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
      WRITE(*,1)  'rewind:', fn, rn, errf, eoff, eotf, tcsr
      DO f = 1, 3
         eoff = .false.
         DO WHILE ( .NOT. eoff )
            n = tread( tlu, onerec )
            n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
            IF (.NOT. eoff) WRITE(*,2) 'read:', onerec,
&               fn, rn, errf, eoff, eotf, tcsr
         END DO
         n = tskipf( tlu, 1, 0 )
         n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
         WRITE(*,1)  'tskip: ', fn, rn, errf, eoff, eotf, tcsr
      END DO
      END
```

The displayed output is:

```
open: 0 0 F F F 0
rewind: 0 0 F F F 0
read: eins 0 1 F F F 0
read: zwei 0 2 F F F 0
tskip: 1 0 F F F 0
read: ichi 1 1 F F F 0
read: ni__ 1 2 F F F 0
tskip: 2 0 F F F 0
read: un__ 2 1 F F F 0
read: deux 2 2 F F F 0
tskip: 3 0 F F F 0
```

EOF and EOT summary:

- If at either EOF or EOT, then:
  - Any tread() will just return; it will not read the tape.
  - A successful tskipf(tlu,1,0) resets the EOF status to false, and returns; it does not advance the tape pointer.
- A successful twrite() resets the EOF and EOT status flags to false.
- A successful tclose() resets all those flags to false.
- tclose() truncates:

  tclose() places an EOF marker immediately after the current location of the unit pointer and then closes the unit. So if you use trewin() to rewind a unit before you use tclose() to close it, its contents are thrown away. This behavior of tclose() is inherited from the Berkeley code.

See also: ioctl(2), mtio(4s), perror(3f), read(2), st(4s), write(2).

7.56 ttynam, isatty: *Get Name of a Terminal Port*

ttynam: *Get Name of a Terminal Port*

The function ttynam returns a blank padded path name of the terminal device associated with logical unit *lunit*.

Function

name = ttynam(*lunit*)			
lunit	integer	Input	Logical unit
Return value	character*n	Output	*name* is nonblank: *name*=path name of device on *lunit*. *name* is an empty string (all blanks): *lunit* is not associated with a terminal device in directory /dev
n	integer	size of *name*	Must be large enough for the longest path name

`isatty`: *Is this Unit a Terminal?*

Function

terminal = isatty(*lunit*)			
lunit	integer	Input	Logical unit
Return value	logical	Output	*terminal*=true: It is a terminal device *terminal*=false: It is *not* a terminal device.

Example: Determine if *lunit* is a tty.

```
      character*12 name, ttynam
      integer lunit /5/
      logical isatty, terminal
      terminal = isatty( lunit )
      name = ttynam( lunit )
      write(*,*) 'terminal = ', terminal, ', name = "', name, '"'
      end
```

The displayed output is:

```
  terminal = T, name = "/dev/ttyp1  "
```

7.57 `unlink`: *Remove a File*

Function

n = unlink (*patnam*)			
patnam	character**n*	Input	File name
Return value	integer	Output	*n*=0: OK *n*>0: Error

The function `unlink` removes the file specified by path name *patnam*.

If this was the last link to the file, the contents of the file are lost.

Example: `unlink()`–Remove the `tunlink.data` file.

```
      call unlink( 'tunlink.data' )
      end
demo$ f77 -silent tunlink.f
demo$ ls tunl*
tunlink.f tunlink.data
demo$ a.out
demo$ ls tunl*
tunlink.f
demo$
```

See also: `unlink`(2), `link`(3F), `perror`(3F).

Bug: Path names can be no longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

7.58 `wait`: *Wait for a Process to Terminate*

Function

n = wait(*status*)			
status	integer	Output	Termination status of the child process
Return value	integer	Output	*n*>0: Process ID of the child process *n*<0: *n*=-(system error code). See `wait`(2).

`wait` suspends the caller until a signal is received or one of its child processes terminates. If any child has terminated since the last `wait`, return is immediate. If there are no children, return is immediate with an error code.

Example: (fragment) `wait()`.

```
      integer n, status, wait
      …
      n = wait( status )
      if ( n .lt. 0 ) stop 'wait: error'
      …
      end
```

See also: `wait`(2), `signal`(3F), `kill`(3F), `perror`(3F).

VMS Routines 8

This chapter is organized into the following sections.

These functions are nonstandard. ♦ The *quad*, `REAL*16`, and `COMPLEX*32` are *SPARC only.*

8.1 VMS Intrinsic Functions

Double-Precision Complex Functions

Table 8-1 Double-Precision Complex Function

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
CDABS	Specific	Absolute value	`COMPLEX*16`	`REAL*8`
CDEXP	Specific	Exponential, ea	`COMPLEX*16`	`COMPLEX*16`
CDLOG	Specific	Natural log	`COMPLEX*16`	`COMPLEX*16`
CDSQRT	Specific	Square root	`COMPLEX*16`	`COMPLEX*16`
CDSIN	Specific	Sine	`COMPLEX*16`	`COMPLEX*16`
CDCOS	Specific	Cosine	`COMPLEX*16`	`COMPLEX*16`
DCMPLX	Specific	Convert to double complex	Any numeric	`COMPLEX*16`
DCONJG	Specific	Complex conjugate	`COMPLEX*16`	`COMPLEX*16`
DIMAG	Specific	Imaginary part of complex	`COMPLEX*16`	`REAL*8`
DREAL	Specific	Real part of complex	`COMPLEX*16`	`REAL*8`

8

Degree-Based Trigonometric Functions

Table 8-2 Degree-based Trigonometric Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
SIND	Generic	Sine	-	-
SIND	Specific	Sine	REAL*4	REAL*4
DSIND	Specific	Sine	REAL*8	REAL*8
QSIND	Specific	Sine	REAL*8	REAL*16
COSD	Generic	Cosine	-	-
COSD	Specific	Cosine	REAL*4	REAL*4
DCOSD	Specific	Cosine	REAL*8	REAL*8
QCOSD	Specific	Cosine	REAL*16	REAL*16
TAND	Generic	Tangent	-	-
TAND	Specific	Tangent	REAL*4	REAL*4
DTAND	Specific	Tangent	REAL*8	REAL*8
QTAND	Specific	Tangent	REAL*16	REAL*16
ASIND	Generic	Arc sine	-	-
ASIND	Specific	Arc sine	REAL*4	REAL*4
DASIND	Specific	Arc sine	REAL*8	REAL*8
QASIND	Specific	Arc sine	REAL*16	REAL*16
ACOSD	Generic	Arc cosine	-	-
ACOSD	Specific	Arc cosine	REAL*4	REAL*4
DACOSD	Specific	Arc cosine	REAL*8	REAL*8
QACOSD	Specific	Arc cosine	REAL*16	REAL*16
ATAND	Generic	Arc tangent	-	-
ATAND	Specific	Arc tangent	REAL*4	REAL*4
DATAND	Specific	Arc tangent	REAL*8	REAL*8
QATAND	Specific	Arc tangent	REAL*16	REAL*16
ATAN2D	Generic	Arc tangent of a1/a2	-	-
ATAN2D	Specific	Arc tangent of a1/a2	REAL*4	REAL*4
DATAN2D	Specific	Arc tangent of a1/a2	REAL*8	REAL*8
QATAN2D	Specific	Arc tangent of a1/a2	REAL*16	REAL*16

REAL*16 is *SPARC only.*

Bit-Manipulation Functions

Table 8-3 Bit Manipulation Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IBITS	Generic	From a1, initial bit a2, extract a3 bits	-	-
IIBITS	Specific	From a1, initial bit a2, extract a3 bits	INTEGER*2	INTEGER*2
JIBITS	Specific	From a1, initial bit a2, extract a3 bits	INTEGER*4	INTEGER*4
ISHFT	Generic	Shift a1 logically by a2 bits *	-	-
ISHFTC	Generic	In a1, circular shift by a2 places, of right a3 bits	-	-
IISHFTC	Specific	In a1, circular shift by a2 places, of right a3 bits	INTEGER*2	INTEGER*2
JISHFTC	Specific	In a1, circular shift by a2 places, of right a3 bits	INTEGER*4	INTEGER*4
IISHFT	Specific	Shift a1 logically left by a2 bits	INTEGER*2	INTEGER*2
JISHFT	Specific	Shift a1 logically left by a2 bits	INTEGER*4	INTEGER*4
IAND	Generic	Bitwise AND of a1, a2	-	-
IIAND	Specific	Bitwise AND of a1, a2	INTEGER*2	INTEGER*2
JIAND	Specific	Bitwise AND of a1, a2	INTEGER*4	INTEGER*4
IOR	Generic	Bitwise OR of a1, a2	-	-
IIOR	Specific	Bitwise OR of a1, a2	INTEGER*2	INTEGER*2
JIOR	Specific	Bitwise OR of a1, a2	INTEGER*4	INTEGER*4
IEOR	Generic	Bitwise exclusive OR of a1, a2	-	-
IIEOR	Specific	Bitwise exclusive OR of a1, a2	INTEGER*2	INTEGER*2
JIEOR	Specific	Bitwise exclusive OR of a1, a2	INTEGER*4	INTEGER*4
NOT	Generic	Bitwise complement	-	-
INOT	Specific	Bitwise complement	INTEGER*2	INTEGER*2
JNOT	Specific	Bitwise complement	INTEGER*4	INTEGER*4
IBSET	Generic	In a1, set bit a2 to 1	-	-
IIBSET	Specific	In a1, set bit a2 to 1; return new a1	INTEGER*2	INTEGER*2
JIBSET	Specific	In a1, set bit a2 to 1; return new a1	INTEGER*4	INTEGER*4
BTEST	Generic	If bit a2 of a1 is 1, return .TRUE.	-	-
BITEST	Specific	If bit a2 of a1 is 1, return .TRUE.	INTEGER*2	LOGICAL*2
BJTEST	Specific	If bit a2 of a1 is 1, return .TRUE.	INTEGER*4	INTEGER*4
IBCLR	Generic	In a1, set bit a2 to 0; return new a1	-	-
IIBCLR	Specific	In a1, set bit a2 to 0; return new a1	INTEGER*2	INTEGER*2
JIBCLR	Specific	In a1, set bit a2 to 0; return new a1	INTEGER*4	INTEGER*4

* ISHFT — If *a2* is positive, then shift left; if negative, then shift right.

Multiple Integer Types

The possibility of multiple integer types is not addressed by the FORTRAN Standard. f77 copes with their existence by treating a specific INTEGER → INTEGER function name (IABS, and so forth) as a special sort of generic. The argument type is used to select the appropriate runtime routine name, which is not accessible to the programmer. VMS FORTRAN takes a similar approach but makes the specific names available.

Table 8-4 Integer Functions

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IIABS	Specific	Absolute value	INTEGER*2	INTEGER*2
JIABS	Specific	Absolute value	INTEGER*4	INTEGER*4
IMAX0	Specific	Maximum [1]	INTEGER*2	INTEGER*2
JMAX0	Specific	Maximum [1]	INTEGER*4	INTEGER*4
IMIN0	Specific	Minimum [1]	INTEGER*2	INTEGER*2
JMIN0	Specific	Minimum [1]	INTEGER*4	INTEGER*4
IIDIM	Specific	Positive difference [2]	INTEGER*2	INTEGER*2
JIDIM	Specific	Positive difference [2]	INTEGER*4	INTEGER*4
IMOD	Specific	Remainder of a1/a2	INTEGER*2	INTEGER*2
JMOD	Specific	Remainder of a1/a2	INTEGER*4	INTEGER*4
IISIGN	Specific	Transfer sign, |a1|* sign(a2)	INTEGER*2	INTEGER*2
JISIGN	Specific	Transfer sign, |a1|* sign(a2)	INTEGER*4	INTEGER*4

1. At least two arguments

2. Positive difference: a1-min(a1,a2)

Functions Coerced to a Particular Type

Some VMS FORTRAN functions coerce to a particular `INTEGER` type.

Table 8-5 Translated Functions that VMS Coerces to a Particular Type

Name	*Gen/Spec*	*Function*	*Arg Type*	*Result Type*
IINT	Specific	Truncation toward zero	`REAL*4`	`INTEGER*2`
JINT	Specific	Truncation toward zero	`REAL*4`	`INTEGER*4`
IIDINT	Specific	Truncation toward zero	`REAL*8`	`INTEGER*2`
JIDINT	Specific	Truncation toward zero	`REAL*8`	`INTEGER*4`
IQINT	Generic	Truncation toward zero	`REAL*16`	`INTEGER`
IIQINT	Specific	Truncation toward zero	`REAL*16`	`INTEGER*2`
JIQINT	Specific	Truncation toward zero	`REAL*16`	`INTEGER*4`
ININT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*4`	`INTEGER*2`
JNINT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*4`	`INTEGER*4`
IIDNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*8`	`INTEGER*2`
JIDNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*8`	`INTEGER*4`
IQNINT	Generic	Nearest integer, `INT(a+.5*sign(a))`	`REAL*16`	`INTEGER`
IIQNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*16`	`INTEGER*2`
JIQNNT	Specific	Nearest integer, `INT(a+.5*sign(a))`	`REAL*16`	`INTEGER*4`
IIFIX	Specific	Fix	`REAL*4`	`INTEGER*2`
JIFIX	Specific	Fix	`REAL*4`	`INTEGER*4`
IMAX1	Specific	Maximum	`REAL*4`	`INTEGER*2`
JMAX1	Specific	Maximum	`REAL*4`	`INTEGER*4`
IMIN1	Specific	Minimum	`READ*4`	`INTEGER*2`
JMIN1	Specific	Minimum	`READ*4`	`INTEGER*4`

`REAL*16` is *SPARC only.*

Functions Translated to a Generic Name

In some cases, each VMS-specific name is translated into an f77 generic name.

Table 8-6 Other Conversions by f77

Name	Gen/Spec	Function	Arg Type	Result Type
FLOATI	Specific	Convert to REAL*4	INTEGER*2	REAL*4
FLOATJ	Specific	Convert to REAL*4	INTEGER*4	REAL*4
DFLOAT	Generic	Convert to REAL*8	INTEGER	REAL*8
DFLOTI	Specific	Convert to REAL*8	INTEGER*2	REAL*8
DFLOTJ	Specific	Convert to REAL*8	INTEGER*4	REAL*8
AIMAX0	Specific	Maximum	INTEGER*2	REAL*4
AJMAX0	Specific	Maximum	INTEGER*4	REAL*4
AIMIN0	Specific	Minimum	INTEGER*2	REAL*4
AJMIN0	Specific	Minimum	INTEGER*4	REAL*4

Zero Extend

The following zero-extend functions are recognized by f77. The first unused high-order bit is set to zero and extended toward the higher-order end to the width indicated in the table.

Table 8-7 Zero Extend Functions

Name	Gen/Spec	Function	Arg Type	Result Type
ZEXT	Generic	Zero-extend	-	-
IZEXT	Specific	Zero-extend	BYTE LOGICAL*1 LOGICAL*2 INTEGER*2	INTEGER*2
JZEXT	Specific	Zero-extend	BYTE LOGICAL*1 LOGICAL*2 LOGICAL*4 INTEGER INTEGER*2 INTEGER*4	INTEGER*4

8.2 VMS System Routines

These routines provide compatibility with VMS FORTRAN system routines.

Summary

To use these system routines, you must do the following:

- Include the `-lV77` option on the `f77` command line.
- Be aware that with `-lV77`, you get the VMS versions of `DATE, IDATE` and `TIME`, instead of the standard versions.

Example: Compile using the `libV77` library.

```
demo$ f77 myprog.f -lV77
```

Table 8-8 Sumary of VMS FORTRAN System Routines

Name	*Definition*	*Calling Sequence*	*Argument Type*	*Returned Type*
date	Date: *dd-mmm-yy*	`call date( c )`	`CHARACTER*9`	n/a
idate	Date: *d, m, y*	`call idate( d, m, y )`	`INTEGER`	n/a
mvbits	Move bit field	`call mvbits( src, ini1, nbits, des, ini2)`	`INTEGER`	n/a
ran	Random number	`r = ran( s )`	`INTEGER*4`	`REAL`
secnds	Elapsed time	`t = secnds( t0 )`	`REAL`	`REAL`
time	Time: *hh:mm:ss*	`call time( t )`	`CHARACTER*8`	n/a

The VMS error condition subroutine `ERRSNS` is *not* provided on SPARCsystems because it is totally specific to the VMS operating system. The terminate program subroutine `EXIT` was already provided by the operating system.

See Chapter 7, "FORTRAN Library Routines," where each of the above routines is described in detail.

8

VMS Language Extensions 9

This chapter is organized into the following sections.

These are all, of course, nonstandard. ♦

9.1 Background

This FORTRAN compiler includes the VMS extensions to make it as easy as possible to port FORTRAN programs from VMS environments to Solaris environments. The compiler provides almost complete compatibility with VMS FORTRAN. These extensions are included in two systems.

- Compiler, command: `f77`
- Debugger, commands: `debugger, dbx`

9.2 VMS Language Features You Get Automatically

This list includes the ones you get automatically. This is only a summary. Details are elsewhere in this manual.

- Namelist I/O
- Unlabeled DO … END DO
- Indefinite DO WHILE … END DO
- BYTE data type
- Logical operations on integers, and arithmetic operations on logicals
- Additional field and edit descriptors for FORMAT statements:
  - Remaining characters (Q)
  - Carriage Control ($)
  - Octal (O)
  - Hexadecimal (X)
  - Hexadecimal (Z)
- Default field indicators for *w*, *d*, and *e* fields in FORMAT statements
- Reading into Hollerith edit descriptors
- APPEND option for OPEN
- Long names (32 characters)
- "_" and "$" in names
- Long source lines (132-character), if the -e option is on.
- Records, structures, unions, and maps
- Getting addresses by the %LOC function
- Passing arguments by the %VAL function
- End-of-line comments
- OPTIONS statement
- VMS Tab-format source lines are valid.
- Initialize in common

  You can initialize variables in common blocks outside of BLOCK DATA subprograms. You can initialize portions of common blocks, but you cannot initialize portions of one common block in more than one subprogram.

- Radix-50

  Radix-50 constants are implemented as f77 bit-string constants, that is, no type is assumed.
- IMPLICIT NONE is treated as IMPLICIT UNDEFINED (A-Z)
- VIRTUAL is treated as DIMENSION.
- Initialize in declarations

Initialization of variables in declaration statements is allowed. Example.

```
CHARACTER*10 NAME /'Nell'/
```

- Noncharacter format specifiers

  If a runtime format specifier is not of type `CHARACTER`, the compiler accepts that too, even though the FORTRAN Standard requires the `CHARACTER` type.

- Omitted arguments in subprogram calls

  The compiler accepts omitted actual argument in a subroutine call, that is, two consecutive commas compile to a null pointer. Reference to that dummy argument gives a segmentation fault.

- `REAL*16`

  (SPARC only) The compiler treats variables of type `REAL*16` as quadruple precision.

- Noncharacter variables

  The FORTRAN Standard requires the "`FILE=`" specifier for `OPEN` and `INQUIRE` to be an expression of type `CHARACTER`. `f77` accepts a numeric variable or array element reference.

- Consecutive operators

  `f77` allows two consecutive arithmetic operators when the second operator is a unary `+` or `-`. Example: Two consecutive operators.

```
X = A ** -B
```

The above statement is treated as follows:

```
X = A ** (-B)
```

- Illegal real expressions

  When the compiler finds a REAL expression where it expects an integer expression, it makes an explicit type conversion to INTEGER (truncates).

  Examples: Contexts for illegal real expressions that f77 converts to integer:
  - Alternate RETURN
  - Dimension declarators and array subscripts
  - Substring selectors
  - Computed GO TO
  - Logical unit number, record number, and record length

- Typeless numeric constants

  Binary, hexadecimal and octal constants are accepted in VMS form. Example: Constants–Binary (B), Octal (O), Hexadecimal (X or Z).

```
      DATA N1 /B'0011111'/, N2/O'37'/, N3/X'1f'/, N4/Z'1f'/
```

- Function length on function name, rather than on the word FUNCTION.

  The compiler accepts nonstandard length specifiers in function declarations. Example: Size on function name, rather than on the word FUNCTION.

```
      INTEGER FUNCTION FCN*2 ( A, B, C )
```

- TYPE and ACCEPT statements are allowed.

- Alternate return

  The nonstandard "&" syntax for alternate-return actual arguments is treated as the standard FORTRAN "*" syntax. Example

```
      CALL SUBX ( I, *100, Z)  ! Standard (OK)
      CALL SUBX ( I, &100, Z ) ! Nonstandard (OK)
```

- The ENCODE and DECODE statements are accepted.

- Direct I/O with 'N record specifier

  The nonstandard record specifier 'N for direct-access I/O statements is OK.

  Example: A nonstandard form for record specifier.

```
      READ ( K ' N ) LIST
```

  The above is treated as:

```
      READ ( UNIT=K, REC=N ) LIST
```

  The logical unit number is K and the number of the record is N.

- NAME, RECORDSIZE, and TYPE options—OPEN has the following alternative options.
  - NAME is treated as FILE.
  - RECORDSIZE is treated as RECL.
  - TYPE is treated as STATUS.

- DISPOSE=*p*

  The DISPOSE=*p* clause in the CLOSE statement is treated as STATUS=*p*.

- Special Intrinsics

  The compiler processes certain special intrinsic functions.
  - %VAL is ok as is.
  - %LOC is treated as LOC.
  - %REF(*expr*) is treated as *expr* (with a warning if *expr* is CHARACTER)
  - %DESCR is reported as an untranslatable feature.

- Variable Expressions in FORMAT Statements

  In general, inside a FORMAT statement, any integer constant can be replaced by an arbitrary expression; the single exception is the "*n*" in an "*n*H…" edit descriptor. The expression itself must be enclosed in angle brackets.
  Example: The "6" in the following statement is a constant.

```
1     FORMAT( 3F6.1 )
```

  The "6" can be replaced by the variable "N", as in:

```
1     FORMAT( 3F<N>.1 )
```

9

9.3 VMS Language Features that Require `-xl`

Although you get most VMS features automatically, without any special options, for a few VMS features you must use the `-xl` option on the `f77` command line.

In general, you need this `-xl` option if a source statement can be interpreted for either a VMS way of behavior or an `f77` way of behavior, and you want the VMS way of behavior. The `-xl` option forces the compiler to interpret it as VMS FORTRAN.

Summary of Features that Require `-xl[d]`

- 1. Unformatted record `size` in words rather than bytes (`-xl`)
- 2. VMS style logical file names (`-xl`)
- 3. Quote (") character introducing octal constants (`-xl`)
- 4. Backslash (\ ) as ordinary character within character constants (`-xl`)
- 5. Nonstandard form of the `PARAMETER` statement (`-xl`)
- 6. Debugging lines as comment lines or FORTRAN statements (`-xld`)
- 7. Align structures as in VMS FORTRAN (`-xl`)

Details of Features that Require `-xl[d]`

1. Unformatted record `size` in words rather than bytes

   In `f77`, direct-access, unformatted files are always opened with the logical record size in *bytes*.

   If the `–xl[d]` option is *not* set:

   then the argument *n* in the `OPEN` option "`RECL=`*n*" is assumed to be the number of bytes to use for the record size.

   If the `–xl[d]` option is set:

   then the argument *n* in the `OPEN` option "`RECL=`*n*" is assumed to be the number of *words*, so the compiler uses *n*`*4` as the number of bytes for the record size.

   If the `–xl[d]` option is set, and if the compiler cannot determine if the file is formatted or unformatted, then it issues a warning message that the record size may need to be adjusted. This could happen if the information is passed in variable character strings.

The record size returned by an INQUIRE statement is *not* adjusted by the compiler; that is, INQUIRE always returns the number of *bytes*.

Note that these record sizes apply to direct-access, unformatted files only.

2. VMS style logical file names

   If the –xl[d] compiler option is set, then the compiler will interpret VMS logical file names on the INCLUDE statement if it finds the environment variable LOGICALNAMEMAPPING to define the mapping between the logical names and the UNIX path name.

   You set the environment variable to a string of the form:

```
"lname1=path1; lname2=path2; … "
```

   Rules: VMS style logical file names

   - Each *lname* is a logical name and each *path1*, *path2*, and so forth, is the path name of a directory (without a trailing '/').
   - It ignores all blanks when parsing this string.
   - It strips any trailing "/[no]list" from the file name in the INCLUDE statement.
   - Logical names in a file name are delimited by the first ":" in the VMS file name.
   - It converts file names from "*lname1:file*" to "*path1/file*" form.
   - For logical names, uppercase/lowercase is significant. If a logical name is encountered on the INCLUDE statement which is not specified in the LOGICALNAMEMAPPING, the file name is used unchanged.

3. Quote (") character introducing octal constants

   If the –xl[d] compiler option is on, a VMS FORTRAN octal *integer* constant is treated as its decimal form.

   Example: VMS octal *integer* constant.

```
JCOUNT = ICOUNT + "703
```

   The above statement is treated as:

```
JCOUNT = ICOUNT + 451
```

   If the –xl[d] compiler option is *not* on, then the "703 is an error.

With the `-xl[d]` compiler option on, the VMS FORTRAN notation `"703` signals `f77` to convert from the integer *octal* constant to its integer *decimal* equivalent, 451 in this case. Note that in VMS FORTRAN the `"703` cannot be the start of a character constant, because VMS FORTRAN character constants are delimited by apostrophes, not quotes.

4. Backslash (\ ) as ordinary character within character constants

   If the `-xl[d]` option is on, a backslash in a character string is treated as an ordinary character; otherwise, it is treated as an escape character.

5. Nonstandard form of the `PARAMETER` statement

   The alternate `PARAMETER` statement syntax is allowed, if the `-xl` [*d*] option is on. Example: VMS alternate form of `PARAMETER` statement omits the parentheses.

```
      PARAMETER FLAG1 = .TRUE.
```

- Debugging lines as comment lines or FORTRAN statements (`-xld`)

  The compiler interprets debugging lines as comment lines or FORTRAN statements, depending on whether the `-xld` option is set. If set, they are compiled; otherwise they are comments.

  Example: Debugging lines.

```
      REAL A(5) / 5.0, 6.0, 7.0, 8.0, 9.0 /
      DO I = 1, 5
           X = A(I)**2
D          PRINT *, I, X
      END DO
      PRINT *, 'done'
      END
```

  With `-xld`, this prints `I` and `X`. Without `-xld`, it does not print them.

6. Align structures as in VMS FORTRAN.

   Use this if your program has some detailed knowledge of how VMS structures are implemented. If you use both `-oldstruct` and `-xl`, then you get `-oldstruct`. If you need to share structures with C, you should use the default: no `-xl` and no `-oldstruct`.

9.4 Unsupported VMS FORTRAN

Most VMS FORTRAN extensions are incorporated into the `f77` compiler. This section lists the few VMS statements that are *not* supported. The compiler writes messages to standard error for any unsupported statements in the source file.

The following VMS FORTRAN features are *not* supported in `f77`.

1. `DEFINE FILE` statement
2. `DELETE` statement
3. `UNLOCK` statement
4. `FIND` statement
5. `REWRITE` statement
6. `KEYID` and key specifiers in `READ` statements
7. Nonstandard `INQUIRE` specifiers
   - `CARRIAGECONTROL`
   - `DEFAULTFILE`
   - `KEYED`
   - `ORGANIZATION`
   - `RECORDTYPE`
8. Nonstandard `OPEN` specifiers
   - `ASSOCIATEVARIABLE`
   - `BLOCKSIZE`
   - `BUFFERCOUNT`
   - `CARRIAGECONTROL`
   - `DEFAULTFILE`
   - `DISP[OSE]`
   - `EXTENDSIZE`
   - `INITIALSIZE`
   - `KEY`
   - `MAXREC`
   - `NOSPANBLOCKS`
   - `ORGANIZATION`
   - `READONLY`

- RECORDTYPE
- SHARED
- USEROPEN

9. The intrinsic function %DESCR.

10. The following parameters on the OPTIONS statement:
    - [NO]G_FLOATING
    - [NO]F77
    - CHECK=[NO]OVERFLOW
    - CHECK=[NO]UNDERFLOW

11. Some of the INCLUDE statement

    Some aspects of the INCLUDE statement are converted. The INCLUDE statement is operating system–dependent, so it cannot be completely converted automatically. The VMS version allows a module-name and a LIST control directive that are indistinguishable from a continuation of a UNIX file name. Also, VMS ignores alphabetic case, so if the programmer is inconsistent about capitalization, distinctions are made where none are intended.

12. Getting a long integer — expecting a short

    In VMS FORTRAN you can pass a long integer argument to a subroutine that expects a short integer. This will work if the long integer fits in 16 bits, because the VAX addresses an integer by its low-order byte. This does *not* work on SPARC computers.

13. Those VMS system calls that are directly tied to that operating system.

14. Initializing a common block in more than one subprogram.

15. Alphabetizing common blocks so you can rely/depend on the order in which blocks are loaded. You can specify the older with the -M *mapfile* option to ld.

16. If you use the defaults for both of the following:
    - The OPEN option BLANK=
    - The BN/BZ/B format edit specifiers

    then formatted numeric input ignores imbedded and trailing blanks. The corresponding VMS defaults treat them as zeros.

ASCII Character Set A

Table A-1 ASCII Character Set

Dec	*Oct*	*Hex*	*Name*	*Dec*	*Oct*	*Hex*	*Name*	*Dec*	*Oct*	*Hex*	*Name*	*Dec*	*Oct*	*Hex*	*Name*
0	000	00	NUL	**32**	040	20	SP	**64**	100	40	@	**96**	140	60	‘
1	001	01	SOH	**33**	041	21	!	**65**	101	41	A	**97**	141	61	a
2	002	02	STX	**34**	042	22	"	**66**	102	42	B	**98**	142	62	b
3	003	03	ETX	**35**	043	23	#	**67**	103	43	C	**99**	143	63	c
4	004	04	EOT	**36**	044	24	$	**68**	104	44	D	**100**	144	64	d
5	005	05	ENQ	**37**	045	25	%	**69**	105	45	E	**101**	145	65	e
6	006	06	ACK	**38**	046	26	&	**70**	106	46	F	**102**	146	66	f
7	007	07	BEL	**39**	047	27	’	**71**	107	47	G	**103**	147	67	g
8	010	08	BS	**40**	050	28	(	**72**	110	48	H	**104**	150	68	h
9	011	09	HT	**41**	051	29	)	**73**	111	49	I	**105**	151	69	i
10	012	0A	LF	**42**	052	2A	*	**74**	112	4A	J	**106**	152	6A	j
11	013	0B	VT	**43**	053	2B	+	**75**	113	4B	K	**107**	153	6B	k
12	014	0C	FF	**44**	054	2C	,	**76**	114	4C	L	**108**	154	6C	l
13	015	0D	CR	**45**	055	2D	–	**77**	115	4D	M	**109**	155	6D	m
14	016	0E	SO	**46**	056	2E	.	**78**	116	4E	N	**110**	156	6E	n
15	017	0F	SI	**47**	057	2F	/	**79**	117	4F	O	**111**	157	6F	o
16	020	10	DLE	**48**	060	30	0	**80**	120	50	P	**112**	160	70	p
17	021	11	DC1	**49**	061	31	1	**81**	121	51	Q	**113**	161	71	q
18	022	12	DC2	**50**	062	32	2	**82**	122	52	R	**114**	162	72	r
19	023	13	DC3	**51**	063	33	3	**83**	123	53	S	**115**	163	73	s
20	024	14	DC4	**52**	064	34	4	**84**	124	54	T	**116**	164	74	t

A

Table A-1 ASCII Character Set

Dec	*Oct*	*Hex*	*Name*	*Dec*	*Oct*	*Hex*	*Name*	*Dec*	*Oct*	*Hex*	*Name*	*Dec*	*Oct*	*Hex*	*Name*
21	025	15	NAK	**53**	065	35	5	**85**	125	55	U	**117**	165	75	u
22	026	16	SYN	**54**	066	36	6	**86**	126	56	V	**118**	166	76	v
23	027	17	ETB	**55**	067	37	7	**87**	127	57	W	**119**	167	77	w
24	030	18	CAN	**56**	070	38	8	**88**	130	58	X	**120**	170	78	x
25	031	19	EM	**57**	071	39	9	**89**	131	59	Y	**121**	171	79	y
26	032	1A	SUB	**58**	072	3A	:	**90**	132	5A	Z	**122**	172	7A	z
27	033	1B	ESC	**59**	073	3B	;	**91**	133	5B	[	**123**	173	7B	{
28	034	1C	FS	**60**	074	3C	<	**92**	134	5C	\	**124**	174	7C	|
29	035	1D	GS	**61**	075	3D	=	**93**	135	5D	]	**125**	175	7D	}
30	036	1E	RS	**62**	076	3E	>	**94**	136	5E	^	**126**	176	7E	~
31	037	1F	US	**63**	077	3F	?	**95**	137	5F	_	**127**	177	7F	DEL

Table A-2 Control Character Meanings

^=Control Key
s^=Shift and Control Keys

Dec	*Oct*	*Hex*	*Name*	*Keys*	*Meaning*
0	000	00	NUL	s^P	Null or time fill character
1	001	01	SOH	^A	Start of Heading
2	002	02	STX	^B	Start of Text
3	003	03	ETX	^C	End of Text (EOM)
4	004	04	EOT	^D	End of Transmission
5	005	05	ENQ	^E	Enquiry (WRU)
6	006	06	ACK	^F	Acknowledge (RU)
7	007	07	BEL	^G	Bell
8	010	08	BS	^H	Backspace
9	011	09	HT	^I	Horizontal Tab
10	012	0A	LF	^J	Line Feed (Newline)
11	013	0B	VT	^K	Vertical Tab
12	014	0C	FF	^L	Form Feed
13	015	0D	CR	^M	Carriage Return
14	016	0E	SO	^N	Shift Out
15	017	0F	SI	^O	Shift In
16	020	10	DLE	^P	Data Link Escape
17	021	11	DC1	^Q	Device Control 1 (X-ON)
18	022	12	DC2	^R	Device Control 2 (TAPE)
19	023	13	DC3	^S	Device Control 3 (X-OFF)
20	024	14	DC4	^T	Device Control 4 (TAPE)
21	025	15	NAK	^U	Negative Acknowledge
22	026	16	SYN	^V	Synchronous Idle
23	027	17	ETB	^W	End of Transmission Blocks
24	030	18	CAN	^X	Cancel
25	031	19	EM	^Y	End Of Medium
26	032	1A	SS	^Z	Special Sequence
27	033	1B	ESC	s^K	Escape (^ [ )
28	034	1C	FS	s^L	File Separator (^ \ )
29	035	1D	GS	s^M	Group Separator (^])
30	036	1E	RS	s^N	Record Separator (^ ‘)
31	037	1F	US	s^O	Unit Separator (^ /)
127	177	7F	DEL	s^0	Delete or Rubout (^ _)

A

Sample Statements B

This appendix shows selected samples of all `f77` statement types, all in the following table. The purpose of the table is to provide a quick reference for syntax details of the more common variations of each statement type.

In this table, the following conventions are used:

`C`	is a character variable	`R`	is an real variable
`CA`	is a character array	`N`	is a numeric variable
`I`	is an integer variable	`L`	is a logical variable
`U`	is an external unit	`S`	is a switch variable
		♦	indicates nonstandard feature

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***
Accept ♦	`ACCEPT *, A, I`	Compare Read.
Assign	`ASSIGN 9 TO I`	

Assignment	`C = 'abc'`
	`C = "abc"`
	`C = S // 'abc'`
	`C = S(I:M)`
		Character
		♦

	`L = L1 .OR. L2` `L = I .LE. 80`	Logical
	`N = N+1` `X = '7FF00000'x`	Arithmetic Hex ♦
	`CURR = NEXT` `NEXT.ID = 82`	Compare Record.

B

Table B-1 FORTRAN Statement Samples

Name	*Examples*	*Comments*

Automatic ♦	`AUTOMATIC A, B, C`
	`AUTOMATIC REAL P, D, Q`
	`IMPLICIT AUTOMATIC REAL (X-Z)`	

Backspace	`BACKSPACE U` `BACKSPACE( UNIT=U, IOSTAT=I, ERR=9 )`	
Block Data	`BLOCK DATA` `BLOCK DATA COEFFS`	

Byte ♦	`BYTE A, B, C`
	`BYTE A, B, C(10)`
	`BYTE A /'x'/, B /255/, C(10)`
		
		Initialize A and B

Call	`CALL P( A, B )`
	`CALL P( A, B, *9 )`
	`CALL P( A, B, &9 )`
	`CALL P`
		
		Alternate return
		Alternate return ♦

Character	`CHARACTER C*80, D*1(4)`
	`CHARACTER*18 A, B, C`
	`CHARACTER A, B*3 /'xyz'/, C /'z'/`
		
		Initialize B and C ♦

Close	`CLOSE ( UNIT=I )` `CLOSE( UNIT=U, ERR=90, IOSTAT=I )`	

Common	`COMMON / DELTAS / H, P, T`
	`COMMON X, Y, Z`
	`COMMON P, D, Q(10,100)`	

Complex	`COMPLEX U, V, U(3,6)`
	`COMPLEX U*16`
	`COMPLEX U*32`
	`COMPLEX U / (1.0,1.0) /, V /(1.0,10.0) /`
		
		Double complex ♦
		Quad complex ♦ (SPARC)
		Initialize U and V ♦

Continue	`100 CONTINUE`	

Data	`DATA A, C / 4.01, 'z' /`
	`DATA (V(I),I=1,3) /.7, .8, .9/`
	`DATA ARRAY(4,4) / 1.0 /`
	`DATA B,O,X,Y /B'0011111', O'37', X'1f', Z'1f'/`
		♦

Decode ♦	`DECODE ( 4, 1, S ) V`	
Dimension	`DIMENSION ARRAY(4, 4)` `DIMENSION V(1000), W(3)`	

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***

Do	`DO 100 I = INIT, LAST, INCR`
	…
	`100 CONTINUE`	

	`DO I = INIT, LAST`
	…
	`END DO`
		Unlabeled `do` ♦

	`DO WHILE ( DIFF .LE. DELTA )`
	…
	`END DO`
		Do While ♦

	`DO 100 WHILE ( DIFF .LE. DELTA )`
	…
	`100 CONTINUE`
		♦

Double Complex ♦	`DOUBLE COMPLEX U, V`
	`DOUBLE COMPLEX U, V`
	`COMPLEX U / (1.0,1.0D0) /, V / (1.0,1.0D0) /`
		`Complex*16` ♦
		`Complex` ♦
		Initialize `U` and `V`

Double Precision	`DOUBLE PRECISION A, D, Y(2)` `DOUBLE PRECISION A, D / 1.2D3 /, Y(2)`	`Real*8` ♦ Initialize `D` ♦
Else	`ELSE`	Compare If (Block).
Else If	`ELSE IF`	
Encode ♦	`ENCODE( 4, 1, T ) A, B, C`	
End	`END`	
End Do ♦	`END DO`	Compare `Do`.

Endfile	`ENDFILE ( UNIT=I )`
	`ENDFILE I`
	`ENDFILE( UNIT=U, IOSTAT=I, ERR=9 )`	

End If	`END IF`	
End Map ♦	`END MAP`	Compare `Map`.
End Structure ♦	`END STRUCTURE`	Compare `Structure`.
End Union ♦	`END UNION`	Compare `Union`.

Entry	`ENTRY SCHLEP( X, Y )`
	`ENTRY SCHLEP( A1, A2, *4 )`
	`ENTRY SCHLEP`	

B

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***

Equivalence	`EQUIVALENCE ( V(1), A(1,1) )`
	`EQUIVALENCE ( V, A )`
	`EQUIVALENCE (X,V(10)), (P,D,Q)`	

External	`EXTERNAL RNGKTA, FIT`	
Format	`10 FORMAT(// 2X, 2I3, 3F6.1, 4E12.2, 2A6,3L2 )`	`X I F E A L`
	`10 FORMAT(// 2D6.1, 3G12.2 )`	`D G`
	`10 FORMAT( 2I3.3, 3G6.1E3, 4E12.2E3 )`	*w*
	`10 FORMAT('a quoted string', " another", I2)`	Strings ♦
	`10 FORMAT( 18Ha hollerith string, I2)`	Hollerith
	`10 FORMAT( 1X, T10, A1, T20, A1 )`	Tabs
	`10 FORMAT( 5X, TR10, A1, TR10, A1, TL5, A1 )`	Tab right, left
	`10 FORMAT(" Init=", I2, :, 3X, "Last=", I2)`	`:`
	`10 FORMAT( 1X, "Enter path name ", $ )`	`$`
	`10 FORMAT( F4.2, Q, 80 A1 )`	`Q`♦
	`10 FORMAT( 'Octal ', O6, ', Hex ' Z6 )`	Octal, hex ♦
	`10 FORMAT( 3F<N>.2 )`	Variable expression ♦
Function	`FUNCTION Z( A, B )`	
	`FUNCTION W( P,D, *9 )`	
	`CHARACTER FUNCTION R*4(P,D,*9 )`	
	`INTEGER*2 FUNCTION M( I, J )`	Short integer ♦
Go To	`GO TO 99`	Unconditional
	`GO TO I, ( 10, 50, 99 )` `GO TO I`	Assigned
	`GO TO ( 10, 50, 99), I`	Computed
If	`IF ( I -K ) 10, 50, 90`	Arithmetic if
	`IF ( L ) RETURN`	Logical if

	`IF ( L ) THEN`
	`  N=N+1`
	`  CALL CALC`
	`ELSE`
	`  K=K+1`
	`  CALL DISP`
	`ENDIF`
		Block if

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***

	`IF ( C .EQ. 'a' ) THEN`
	`  NA=NA+1`
	`  CALL APPEND`
	`ELSE IF ( C .EQ. 'b' ) THEN`
	`  NB=NB+1`
	`  CALL BEFORE`
	`ELSE IF ( C .EQ. 'c' ) THEN`
	`  NC=NC+1`
	`  CALL CENTER`
	`END IF`
		Block if
		with else if

Implicit	`IMPLICIT COMPLEX (U-W,Z)` `IMPLICIT UNDEFINED (A-Z)`	
Include ♦	`INCLUDE 'project02/header'`	

Inquire	`INQUIRE( UNIT=3, OPENED=OK )`
	`INQUIRE( FILE='mydata', EXIST=OK )`
	`INQUIRE( UNIT=3, OPENED=OK, IOSTAT=ERRNO )`	

Integer	`INTEGER C, D(4)`
	`INTEGER C*2`
	`INTEGER*4 A, B, C`
	`INTEGER A/ 100 /, B, C / 9 /`
		
		Short integer ♦
		
		Initialize A and C ♦

Intrinsic	`INTRINSIC SQRT, EXP`	

Logical	`LOGICAL C`
	`LOGICAL B*1, C*1`
	`LOGICAL*1 B, C`
	`LOGICAL*4 A, B, C`
	`LOGICAL B / .FALSE. /, C`
		
		♦
		♦
		♦
		Initialize B ♦

Map ♦	`MAP`
	`  CHARACTER *18 MAJOR`
	`END MAP`
	`MAP`
	`  INTEGER*2 CREDITS`
	`  CHARACTER*8 GRAD_DATE`
	`END MAP`
		Compare Structure and Union.

Namelist ♦	`NAMELIST /CASE/ S, N, D`	
Open	`OPEN( UNIT=3, FILE="data.test" )` `OPEN( UNIT=3, IOSTAT=ERRNO )`	

B

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***
Options ♦	`OPTIONS /CHECK /EXTEND_SOURCE`	

Parameter	`PARAMETER (A="xyz"), (PI=3.14)`
	`PARAMETER (A="z", PI=3.14)`
	`PARAMETER X=11, Y=X/3`
		
		♦

Pause	`PAUSE`	
Pointer ♦	`POINTER ( P, V ), ( I, X )`	
Pragma ♦	`EXTERNAL RNGKTA, FIT !$PRAGMA C(RNGKTA, FIT)`	`C()` directive
Program	`PROGRAM FIDDLE`	
Print	`PRINT *, A, I`	List-directed
	`PRINT 10, A, I`	Formatted
	`PRINT 10, M`	Array M
	`PRINT 10, (M(I),I=J,K)`	Implied-DO
	`PRINT 10, C(I:K)`	Substring
	`PRINT '(A6,I3)', A, I` `PRINT FMT='(A6,I3)', A, I`	Character constant format
	`PRINT S, I` `PRINT FMT=S, I`	Switch variable has format number
	`PRINT G`	Namelist ♦
Read	`READ *, A, I`	List-directed
	`READ 1, A, I`	Formatted
	`READ 10, M`	Array `M`
	`READ 10, (M(I),I=J,K)`	Implied-DO
	`READ 10, C(I:K)`	Substring
	`READ '(A6,I3)', A, I`	Character constant format

	`READ( 1, 2 ) X, Y`
	`READ( UNIT=1, FMT=2) X,Y`
	`READ( 1, 2, ERR=8,END=9) X,Y`
	`READ( UNIT=1, FMT=2, ERR=8,END=9) X,Y`
		Formatted read from a file

	`READ( *, 2 ) X, Y`	Formatted read from standard input
	`READ( *, 10 ) M`	Array `M`
	`READ( *, 10 ) (M(I),I=J,K)`	Implied-DO

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***
	`READ( *, 10) C(I:K)`	Substring
	`READ( 1, * ) X, Y` `READ( *, * ) X, Y`	List-directed from file — from standard input
	`READ( 1, '(A6,I3)') X, Y` `READ( 1, FMT='(A6,I3)') X, Y`	Character constant format
	`READ( 1, C ) X, Y` `READ( 1, FMT=C ) X, Y`	
	`READ( 1, S ) X, Y` `READ( 1, FMT=S ) X, Y`	Switch variable has format number
	`READ( *, G )` `READ( 1, G )`	Namelist read ♦ Namelist read from a file ♦
	`READ( 1, END=8, ERR=9 ) X, Y`	Unformatted direct access
	`READ( 1, REC=3 ) V` `READ( 1 ' 3 ) V`	Unformatted direct access
	`READ( 1, 2, REC=3 ) V`	Formatted direct access
	`READ( CA, 1, END=8, ERR=9 ) X, Y`	Internal formatted sequential
	`READ( CA, *, END=8, ERR=9 ) X, Y`	Internal list-directed sequential access ♦
	`READ( CA, REC=4, END=8, ERR=9 ) X, Y`	Internal direct access ♦

Real	`REAL R, M(4)`
	`REAL R*4`
	`REAL*8 A, B, C`
	`REAL*16 A, B, C`
	`REAL A / 3.14 /, B, C / 100.0 /`
		
		♦
		Double Precision ♦
		Quad Precision ♦ *(SPARC)*
		Initialize `A` and `C` ♦

Record ♦	`RECORD /PROD/ CURR,PRIOR,NEXT`	
Return	`RETURN` `RETURN 2`	Standard return Alternate return

Rewind	`REWIND 1`
	`REWIND I`
	`REWIND ( UNIT=U, IOSTAT=I, ERR=9 )`	

Save	`SAVE A, /B/, C` `SAVE`	

B

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***

Static ♦	`STATIC A, B, C`
	`STATIC REAL P, D, Q`
	`IMPLICIT STATIC REAL (X-Z)`	

Stop	`STOP` `STOP "all gone"`	

Structure	`STRUCTURE /PROD/`
	`  INTEGER*4  ID / 99 /`
	`  CHARACTER*18 NAME`
	`  CHARACTER*8  MODEL / 'XL' /`
	`  REAL*4  COST`
	`  REAL*4  PRICE`
	`END STRUCTURE`	

Subroutine	`SUBROUTINE SHR( A, B, *9 )`
	`SUBROUTINE SHR( A, B, &9 )`
	`SUBROUTINE SHR( A, B )`
	`SUBROUTINE SHR`
		Alternate Return
		♦

Type ♦	`TYPE *,  A, I`	Compare `PRINT`

Union ♦	`UNION`
	`  MAP`
	`  CHARACTER*18 MAJOR`
	`  END MAP`
	`  MAP`
	`  INTEGER*2 CREDITS`
	`  CHARACTER*8 GRAD_DATE`
	`  END MAP`
	`END UNION`
		Compare Structure

Virtual ♦	`VIRTUAL M(10,10), Y(100)`	
Volatile ♦	`VOLATILE V, Z, MAT, /INI/`	

Write	`WRITE( 1, 2 ) X, Y }`
	`WRITE( UNIT=1, FMT=2 ) X, Y`
	`WRITE( 1, 2, ERR=8, END=9 ) X, Y`
	`WRITE( UNIT=1, FMT=2, ERR=8, END=9 ) X, Y`
		Formatted write to a file

	`WRITE( *, 2 ) X, Y` `WRITE( *, 10 ) M`	Formatted write to stdout Array `M`
	`WRITE( *, 10 ) (M(I),I=J,K)`	Implied-DO
	`WRITE( *, 10) C(I:K)`	Substring

Table B-1 FORTRAN Statement Samples

Name	***Examples***	***Comments***
	`WRITE( 1, * ) X, Y`	List-directed write to a file
	`WRITE( *, * ) X, Y`	List-directed write to standard output
	`WRITE( 1, '(A6,I3)') X, Y` `WRITE( 1, FMT='(A6,I3)') X, Y`	Character constant format
	`WRITE( 1, C ) X, Y` `WRITE( 1, FMT=C ) X, Y`	Character variable format
	`WRITE( 1, S ) X, Y` `WRITE( 1, FMT=S ) X, Y`	Switch variable has format number
	`WRITE( *, CASE )` `WRITE( 1, CASE )`	Namelist write ♦ Namelist write to a file ♦
	`WRITE( 1, END=8, ERR=9 ) X, Y`	Unformatted sequential access
	`WRITE( 1, REC=3 ) V` `WRITE( 1 ' 3 ) V`	Unformatted direct access
	`WRITE( 1, 2, REC=3 ) V`	Formatted direct access
	`WRITE( CA, 1, END=8, ERR=9 ) X, Y`	Internal formatted sequential
	`WRITE( CA, *, END=8, ERR=9 ) X, Y`	Internal list-directed sequential access ♦
	`WRITE( CA, REC=4, END=8, ERR=9 ) X, Y`	Internal direct access ♦

B

Data Representations C

This appendix is organized into the following sections.

This appendix is a brief introduction to data representation. For some detail and explanation, read the *FORTRAN User's Guide*; for even more, read the *Numerical Computation Guide.* Whatever the size of the data element in question, the most significant bit of the data element is always stored in the lowest-numbered byte of the byte sequence required to represent that object.

C.1 Real, Double, and Quadruple Precision

Real, double precision, and quadruple precision number data elements are represented according to the IEEE standard by the following form, where *f* is the bits in the fraction. The *quad* is *SPARC only.*

$$(-1)^{sign} * 2^{exponent\text{-}bias} * 1.f$$

C

Table C-1 Floating-point Representation

	Single	***Double***	***Quadruple***
Sign	Bit 31	Bit 63	Bit 127
Exponent	Bits 30–23 Bias 127	Bits 62–52 Bias 1023	Bits 126-112 Bias 16583
Fraction	Bits 22–0	Bits 51–0	Bits 111-0
Range approx.	3.402823e+38 1.175494e-38	1.797693e+308 2.225074e-308	3.362E-4932 1.20E+4932

C.2 Extreme Exponents

Zero (signed)

Zero (signed) is represented by an exponent of zero and a fraction of zero.

Subnormal Number

The form of a subnormal number is

$$(-1)^{sign} * 2^{1-bias} * 0.f$$

where *f* is the bits in the significand.

Signed Infinity

Signed infinity (that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero fraction.

Not a Number (NaN)

Not a Number (NaN) is represented by the largest value that the exponent can assume (all ones), and a nonzero fraction.

Normalized REAL and DOUBLE PRECISION numbers have an implicit leading bit that provides one more bit of precision than is stored in memory. For example, IEEE double precision provides 53 bits of precision: 52 bits stored in the fraction, plus the implicit leading 1.

C.3 IEEE Representation of Selected Numbers

The values here are as shown by dbx, in hexadecimal.

Table C-2 IEEE Representation of Selected Numbers

Value	***Single-Precision***	***Double-Precision***
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7Fxxxxxx	7FFxxxxxxxxxxxx

C.4 Arithmetic Operations on Extreme Values

This section describes the results of basic arithmetic operations with extreme and ordinary values. We assume all inputs are positive, and no traps, overflow, underflow, or other exceptions happen.

Table C-3 Extreme Value Abbreviations

Abbreviation	***Meaning***
Sub	Subnormal number
Num	Normalized number
Inf	Infinity (positive or negative)
NaN	Not a Number
Uno	Unordered

Table C-4 Extreme Values: Addition and Subtraction

Left Operand	***Right Operand***				
	0	***Sub***	***Num***	***Inf***	***NaN***
0	0	Sub	Num	Inf	NaN
Sub	Sub	Sub	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Read Note	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Note: Above, for Inf ± Inf: Inf + Inf = Inf, and Inf - Inf = NaN.

Table C-5 Extreme Values: Multiplication

Left Operand	***Right Operand***				
	0	***Sub***	***Num***	***Inf***	***NaN***
0	0	0	0	NaN	NaN
Sub	0	0	NS	Inf	NaN
Num	0	NS	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Above, NS means either Num or Sub result possible.

Table C-6 Extreme Values: Division

Left Operand	***Right Operand***				
	0	***Sub***	***Num***	***Inf***	***NaN***
0	NaN	0	0	0	NaN
Sub	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Table C-7 Extreme Values: Comparison

Left Operand	***Right Operand***				
	0	***Sub***	***Num***	***Inf***	***NaN***
0	=	<	<	<	Uno
Sub	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>	=	Uno
NaN	Uno	Uno	Uno	Uno	Uno

- If either `X` or `Y` is `NaN`, then "`X.NE.Y`" is `.TRUE.` and the others (`.EQ.`, `.GT.`, `.GE.`, `.LT.`, `.LE.`) are `.FALSE.`.
- +0 compares equal to -0.
- If any argument is `NaN`, then the results of `MAX` or `MIN` are undefined.

C.5 Bits and Bytes by Architecture

The order in which the data—the bits and bytes—are arranged differs between VAX computers on the one hand and SPARC computers on the other.

The bytes in a 32-bit integer, when read from address *n*, end up in the register as shown below.

Table C-8 Bits and Bytes for Intel[1] and VAX Computers

Byte n+3	***Byte n+2***	***Byte n+1***	***Byte n***
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07 06 05 04 03 02 01 00
Most Significant			Least significant

Table C-9 Bits and Bytes for 680x0 and SPARC Computers

Byte n	***Byte n+1***	***Byte n+2***	***Byte n+3***
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07 06 05 04 03 02 01 00
Most Significant			Least significant

The bits are numbered the same on these systems, even though the bytes are numbered differently.

Possible Problem Area

- Passing binary data over the network. Use External Data Representation (XDR) format, or another standard network format to avoid problems.
- Porting raster graphics images between architectures. If your program uses graphics images in binary form, and they have byte ordering that is not the same as for images produced by SPARCsystem routines, you need to convert them.
- If you convert character-to-integer or integer-to-character between architectures, you should use XDR.

1. Intel is a trademark of Intel Corporation.

- If you read binary data created on an architecture with a different byte order, then you need to filter it to correct the byte order.

Again, for more detail and explanation, read the *Numerical Computation Guide* or the *FORTRAN User's Guide*. See also the `man` page `xdr`(3N).

C

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

R

S

T

U

V

W

X

Y

Z