Important Note on New Product Names

As part of Sun’s new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun’s core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

<table>
<thead>
<tr>
<th>Old Product Name</th>
<th>New Product Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun Visual WorkShop™ C++</td>
<td>Forte™ C++ Enterprise Edition 6</td>
</tr>
<tr>
<td>Sun Performance WorkShop™ Fortran</td>
<td>Forte™ for High Performance Computing 6</td>
</tr>
<tr>
<td>Sun Performance WorkShop™ Fortran Personal Edition</td>
<td>Forte™ Fortran Desktop Edition 6</td>
</tr>
<tr>
<td>Sun WorkShop Professional™ C</td>
<td>Forte™ C 6</td>
</tr>
<tr>
<td>Sun WorkShop™ University Edition</td>
<td>Forte™ Developer University Edition 6</td>
</tr>
</tbody>
</table>

In addition to the name changes, there have been major changes to two of the products:

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.

- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.
Contents

Preface 1

1. Elements of FORTRAN 11
   Standards Conformance 11
   Extensions 12
   Basic Terms 12
   Character Set 12
   Symbolic Names 14
   Program Units 16
   Statements 16
   Source Line Formats 17

2. Data Types and Data Items 23
   Types 23
   CHARACTER 26
   COMPLEX 26
   DOUBLE PRECISION 27
   INTEGER 28
   LOGICAL 29
   REAL 30
3. **Expressions** 69
   - Expressions, Operators, and Operands 69
   - Arithmetic Expressions 70
   - Character Expressions 76
   - Logical Expressions 80
   - Relational Operator 82
   - Constant Expressions 83
   - Record Assignment 84
   - Evaluation of Expressions 85

4. **Statements** 87
   - ACCEPT 88
   - ASSIGN 89
   - Assignment 91
   - AUTOMATIC 96
   - BACKSPACE 98
   - BLOCK DATA 100
   - BYTE 102
   - CALL 103
   - CHARACTER 107
   - CLOSE 110
   - COMMON 112
IF (Arithmetic)  169
IF (Block)  170
IF (Logical)  173
IMPLICIT  174
INCLUDE  177
INQUIRE  180
INTEGER  186
INTEGER  187
INTRINSIC  189
LOGICAL  191
LOGICAL  192
MAP  194
NAMELIST  195
OPEN  197
OPTIONS  205
PARAMETER  207
PAUSE  210
POINTER  212
PRINT  217
PROGRAM  220
READ  221
REAL  227
REAL  228
RECORD  230
RETURN  233
REWIND  235
SAVE  237
Statement Function  239
B. Sample Statements 355

C. Data Representations 365
   Real, Double, and Quadruple Precision 365
   Extreme Exponents 366
   IEEE Representation of Selected Numbers 367
   Arithmetic Operations on Extreme Values 367
   Bits and Bytes by Architecture 369

D. VMS Language Extensions 371
   Background 371
   VMS Language Features in Sun Fortran 371
   VMS Features Requiring -x1 or -vax=spec 375

Index 381
# Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 1-1</td>
<td>Special Character Usage</td>
<td>13</td>
</tr>
<tr>
<td>TABLE 1-2</td>
<td>Items with Symbolic Names</td>
<td>14</td>
</tr>
<tr>
<td>TABLE 1-3</td>
<td>Sample Symbolic Names</td>
<td>15</td>
</tr>
<tr>
<td>TABLE 1-4</td>
<td>FORTRAN Statements</td>
<td>17</td>
</tr>
<tr>
<td>TABLE 2-1</td>
<td>Default Data Sizes and Alignments (in Bytes)</td>
<td>31</td>
</tr>
<tr>
<td>TABLE 2-2</td>
<td>Data Defaults Changed by -i2, -r8, -dbl</td>
<td>32</td>
</tr>
<tr>
<td>TABLE 2-3</td>
<td>Backslash Escape Sequences</td>
<td>36</td>
</tr>
<tr>
<td>TABLE 3-1</td>
<td>Arithmetic Operators</td>
<td>70</td>
</tr>
<tr>
<td>TABLE 3-2</td>
<td>Arithmetic Expressions</td>
<td>71</td>
</tr>
<tr>
<td>TABLE 3-3</td>
<td>Arithmetic Operator Precedence</td>
<td>71</td>
</tr>
<tr>
<td>TABLE 3-4</td>
<td>Logical Operators</td>
<td>80</td>
</tr>
<tr>
<td>TABLE 3-5</td>
<td>Logical Operator Precedence</td>
<td>80</td>
</tr>
<tr>
<td>TABLE 3-6</td>
<td>Operator Precedence</td>
<td>80</td>
</tr>
<tr>
<td>TABLE 3-7</td>
<td>Logical Expressions and Their Meanings</td>
<td>81</td>
</tr>
<tr>
<td>TABLE 3-8</td>
<td>Relational Operators</td>
<td>82</td>
</tr>
<tr>
<td>TABLE 4-1</td>
<td>INQUIRE Statement Specifiers</td>
<td>181</td>
</tr>
<tr>
<td>TABLE 4-2</td>
<td>Intrinsics That Cannot Be Passed As Actual Arguments</td>
<td>190</td>
</tr>
<tr>
<td>TABLE 4-3</td>
<td>OPEN Statement Specifiers</td>
<td>198</td>
</tr>
<tr>
<td>TABLE 4-4</td>
<td>OPTIONS Statement Qualifiers</td>
<td>205</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Table 5-1</td>
<td>Summary of $\texttt{f77}$ Input and Output</td>
<td>265</td>
</tr>
<tr>
<td>Table 5-2</td>
<td>Format specifiers</td>
<td>275</td>
</tr>
<tr>
<td>Table 5-3</td>
<td>Default $w$, $d$, $e$ Values in Format Field Descriptors</td>
<td>276</td>
</tr>
<tr>
<td>Table 5-4</td>
<td>Carriage control with blank, 0, 1, and +</td>
<td>279</td>
</tr>
<tr>
<td>Table 5-5</td>
<td>Maximum characters in noncharacter type Hollerith (nHaaa)</td>
<td>283</td>
</tr>
<tr>
<td>Table 5-6</td>
<td>Sample Octal/Hex Input Values</td>
<td>288</td>
</tr>
<tr>
<td>Table 5-7</td>
<td>Sample Octal/Hex Output Value</td>
<td>289</td>
</tr>
<tr>
<td>Table 5-8</td>
<td>Default formats for list-directed output</td>
<td>314</td>
</tr>
<tr>
<td>Table 6-1</td>
<td>Arithmetic functions</td>
<td>327</td>
</tr>
<tr>
<td>Table 6-2</td>
<td>Type conversion functions</td>
<td>329</td>
</tr>
<tr>
<td>Table 6-3</td>
<td>Trigonometric functions</td>
<td>331</td>
</tr>
<tr>
<td>Table 6-4</td>
<td>Other mathematical functions</td>
<td>333</td>
</tr>
<tr>
<td>Table 6-5</td>
<td>Character functions</td>
<td>335</td>
</tr>
<tr>
<td>Table 6-6</td>
<td>Bitwise functions</td>
<td>336</td>
</tr>
<tr>
<td>Table 6-7</td>
<td>Environmental inquiry functions</td>
<td>337</td>
</tr>
<tr>
<td>Table 6-8</td>
<td>Memory functions</td>
<td>338</td>
</tr>
<tr>
<td>Table 6-9</td>
<td>VMS double-precision complex functions</td>
<td>345</td>
</tr>
<tr>
<td>Table 6-10</td>
<td>VMS degree-based trigonometric functions</td>
<td>345</td>
</tr>
<tr>
<td>Table 6-11</td>
<td>VMS bit-manipulation functions</td>
<td>346</td>
</tr>
<tr>
<td>Table 6-12</td>
<td>VMS integer functions</td>
<td>348</td>
</tr>
<tr>
<td>Table 6-13</td>
<td>Translated functions that VMS coerces to a particular type</td>
<td>349</td>
</tr>
<tr>
<td>Table 6-14</td>
<td>VMS functions that are translated into $\texttt{f77}$ generic names</td>
<td>350</td>
</tr>
<tr>
<td>Table 6-15</td>
<td>Zero-extend functions</td>
<td>350</td>
</tr>
<tr>
<td>Table A-1</td>
<td>ASCII character set</td>
<td>352</td>
</tr>
<tr>
<td>Table A-2</td>
<td>Control characters $^\wedge$ = Control Key $s^\wedge$ = Shift and Control Keys</td>
<td>353</td>
</tr>
<tr>
<td>Table B-1</td>
<td>FORTRAN statement samples</td>
<td>355</td>
</tr>
<tr>
<td>Table C-1</td>
<td>Floating-point representation</td>
<td>365</td>
</tr>
<tr>
<td>Table C-2</td>
<td>IEEE representation of selected numbers</td>
<td>367</td>
</tr>
<tr>
<td>Table C-3</td>
<td>Extreme value abbreviations</td>
<td>367</td>
</tr>
</tbody>
</table>
TABLE C-4  Extreme Values: Addition and Subtraction  368
TABLE C-5  Extreme Values: Multiplication  368
TABLE C-6  Extreme Values: Division  369
TABLE C-7  Extreme Values: Comparison  369
TABLE C-8  Bits and Bytes for Intel and VAX Computers  370
TABLE C-9  Bits and Bytes for 680x0 and SPARC Computers  370
Preface

The *FORTRAN 77 Language Reference* specifies the FORTRAN 77 programming language and extensions accepted by the Sun WorkShop™ 6 *f77* compiler.

This is a reference manual intended for programmers with a working knowledge of the Fortran language and the Solaris™ operating environment.

Multiplatform Release

This Sun WorkShop Fortran release supports versions 2.6, 7, and 8 of the Solaris™ SPARC™ Platform Edition Operating Environment.

See the Fortran 77 README file, *fortran_77*, in the Sun WorkShop READMEs directory for information regarding availability of this release of the *f77* compiler on specific platforms (or use the command *f77* -xhelp=readme).

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard /usr/bin/ and /usr/share/man directories, you must change your PATH and MANPATH environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your PATH environment variable:
1. Display the current value of the PATH variable by typing:

```
% echo $PATH
```

2. Review the output for a string of paths containing /opt/SUNWspro/bin/.
   
   If you find the paths, your PATH variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your PATH environment variable by following the instructions in this section.
   
   To determine if you need to set your MANPATH environment variable:
   
   1. Request the workshop man page by typing:

```
% man workshop
```

   2. Review the output, if any.
   
   If the workshop(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in this section for setting your MANPATH environment variable.

   **Note** – The information in this section assumes that your Sun WorkShop 6 products were installed in the /opt directory. Contact your system administrator if your Sun WorkShop software is not installed in /opt.

   The PATH and MANPATH variables should be set in your home .cshrc file if you are using the C shell or in your home .profile file if you are using the Bourne or Korn shells:
   
   - To use Sun WorkShop commands, add the following to your PATH variable:
     
     ```
     /opt/SUNWspro/bin
     ```
   
   - To access Sun WorkShop man pages with the man command, add the following to your MANPATH variable:
     
     ```
     /opt/SUNWspro/man
     ```
   
   For more information about the PATH variable, see the csh(1), sh(1), and ksh(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH and MANPATH variables to access this release, see the Sun WorkShop 6 Installation Guide or your system administrator.
How This Book Is Organized

The FORTRAN 77 Language Reference contains the following chapters and appendixes:

Chapter 1, “Elements of FORTRAN,” introduces the basic parts of Sun WorkShop FORTRAN 77, standards conformance, and elements of the language.

Chapter 2, “Data Types and Data Items,” describes the data types and data structures in the language, including arrays, substrings, structures, and pointers.

Chapter 3, “Expressions,” discusses FORTRAN expressions and how they are evaluated.

Chapter 4, “Statements,” details the statements in the FORTRAN 77 language and the extensions recognized by the Sun WorkShop compiler.

Chapter 5, “Input and Output,” describes the general concepts of FORTRAN input/output and provides details on the different I/O operations.

Chapter 6, “Intrinsic Functions,” tabulates and explains the intrinsic functions that are part of Sun WorkShop FORTRAN 77, including VAX VMS extensions.

Appendix A, “ASCII Character Set,” lists the standard ASCII character set.

Appendix B, “Sample Statements,” shows samples of selected FORTRAN 77 statements for quick reference.

Appendix C, “Data Representations,” introduces the way data is represented in FORTRAN.

Appendix D, “VMS Language Extensions,” describes the VAX VMS language extensions provided in Sun WorkShop FORTRAN 77.
Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. % You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, when contrasted with on-screen computer output</td>
<td>% su Password:</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, words to be emphasized</td>
<td>Read Chapter 6 in the User's Guide. These are called class options. You must be superuser to do this.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Command-line placeholder text; replace with a real name or value</td>
<td>To delete a file, type rm filename.</td>
</tr>
</tbody>
</table>

■ The symbol Δ stands for a blank space where a blank is significant:

ΔΔ36.001

■ Nonstandard features are tagged with the symbol “✠”. Standards are discussed in Chapter 1.

■ FORTRAN examples appear in tab format, not fixed columns. See the discussion of source line formats in the Fortran User's Guide for details.

■ The FORTRAN 77 standard uses an older convention of spelling the name "FORTRAN" capitalized. Sun documentation uses both FORTRAN and Fortran. The current convention is to use lower case: "Fortran 95".

ΔΔ36.001
Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell</td>
<td>%</td>
</tr>
<tr>
<td>Bourne shell and Korn shell</td>
<td>$</td>
</tr>
<tr>
<td>C shell, Bourne shell, and Korn shell superuser</td>
<td>#</td>
</tr>
</tbody>
</table>

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the docs.sun.com™ Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:

  [http://docs.sun.com](http://docs.sun.com)

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:

  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:

    `/opt/SUNWspro/docs/index.html`

(Contact your system administrator if your Sun WorkShop software is not installed in the `/opt` directory.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document’s title.
TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

<table>
<thead>
<tr>
<th>Document Collection</th>
<th>Document Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forte™ Developer 6 / Sun WorkShop 6</td>
<td>About Sun WorkShop 6 Documentation</td>
<td>Describes the documentation available with this Sun WorkShop release and how to access it.</td>
</tr>
<tr>
<td></td>
<td>What’s New in Sun WorkShop 6</td>
<td>Provides information about the new features in the current and previous release of Sun WorkShop.</td>
</tr>
<tr>
<td></td>
<td>Sun WorkShop 6 Release Notes</td>
<td>Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.</td>
</tr>
<tr>
<td>Forte Developer 6 / Sun WorkShop 6</td>
<td>Analyzing Program Performance With Sun WorkShop 6</td>
<td>Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool er_print, the LoopTool and LoopReport utilities, and UNIX profiling tools prof, gprof, and tcov.</td>
</tr>
<tr>
<td></td>
<td>Debugging a Program With dbx</td>
<td>Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.</td>
</tr>
<tr>
<td></td>
<td>Introduction to Sun WorkShop</td>
<td>Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.</td>
</tr>
<tr>
<td>Document Collection</td>
<td>Document Title</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Forte™ C 6 / Sun WorkShop 6 Compilers C</td>
<td>C User’s Guide</td>
<td>Describes the C compiler options, Sun-specific capabilities such as pragmas, the lint tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.</td>
</tr>
<tr>
<td>Forte™ C++ 6 / Sun WorkShop 6 Compilers C++</td>
<td>C++ Library Reference</td>
<td>Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.</td>
</tr>
<tr>
<td></td>
<td>C++ Migration Guide</td>
<td>Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.</td>
</tr>
<tr>
<td></td>
<td>C++ Programming Guide</td>
<td>Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.</td>
</tr>
<tr>
<td></td>
<td>C++ User’s Guide</td>
<td>Provides information on command-line options and how to use the compiler.</td>
</tr>
<tr>
<td></td>
<td>Sun WorkShop Memory Monitor User’s Manual</td>
<td>Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see /opt/SUNWpro/docs/index.html) and not at the docs.sun.com Web site.</td>
</tr>
<tr>
<td>Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95</td>
<td>Fortran Library Reference</td>
<td>Provides details about the library routines supplied with the Fortran compiler.</td>
</tr>
<tr>
<td>Document Collection</td>
<td>Document Title</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>FORTRAN 77 Language Reference</strong></td>
<td>Provides a complete language reference.</td>
<td></td>
</tr>
<tr>
<td><strong>Fortran Programming Guide</strong></td>
<td>Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.</td>
<td></td>
</tr>
<tr>
<td><strong>Fortran User’s Guide</strong></td>
<td>Provides information on command-line options and how to use the compilers.</td>
<td></td>
</tr>
<tr>
<td><strong>FORTRAN 77 Language Reference</strong></td>
<td>Provides a complete language reference.</td>
<td></td>
</tr>
<tr>
<td><strong>Interval Arithmetic Programming Reference</strong></td>
<td>Describes the intrinsic INTERVAL data type supported by the Fortran 95 compiler.</td>
<td></td>
</tr>
<tr>
<td><strong>Forte™ TeamWare 6 / Sun WorkShop TeamWare 6 User’s Guide</strong></td>
<td>Describes how to use the Sun WorkShop TeamWare code management tools.</td>
<td></td>
</tr>
<tr>
<td><strong>Forte™ / Sun Performance Library 6 User’s Guide</strong></td>
<td>Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.</td>
<td></td>
</tr>
<tr>
<td><strong>Sun Performance Library Reference</strong></td>
<td>Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.</td>
<td></td>
</tr>
<tr>
<td><strong>Numerical Computation Guide</strong></td>
<td>Describes issues regarding the numerical accuracy of floating-point computations.</td>
<td></td>
</tr>
<tr>
<td><strong>Standard Library 2 Standard C++ Class Library Reference</strong></td>
<td>Provides details on the Standard C++ Library.</td>
<td></td>
</tr>
<tr>
<td><strong>Standard C++ Library User’s Guide</strong></td>
<td>Describes how to use the Standard C++ Library.</td>
<td></td>
</tr>
<tr>
<td><strong>Tools.h++ 7 Tools.h++ Class Library Reference</strong></td>
<td>Provides details on the Tools.h++ class library.</td>
<td></td>
</tr>
<tr>
<td><strong>Tools.h++ User’s Guide</strong></td>
<td>Discusses use of the C++ classes for enhancing the efficiency of your programs.</td>
<td></td>
</tr>
</tbody>
</table>
TABLE P-4 describes related Solaris documentation available through the docs.sun.com Web site.

<table>
<thead>
<tr>
<th>Document Collection</th>
<th>Document Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris Software Developer</td>
<td><em>Linker and Libraries Guide</em></td>
<td>Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.</td>
</tr>
<tr>
<td></td>
<td><em>Programming Utilities Guide</em></td>
<td>Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.</td>
</tr>
</tbody>
</table>
Elements of FORTRAN

This chapter introduces the basic elements of Sun WorkShop FORTRAN 77.

Standards Conformance

- f77 was designed to be compatible with the ANSI X3.9-1978 FORTRAN standard and the corresponding International Organization for Standardization (ISO) 1539-1980, as well as standards FIPS 69-1, BS 6832, and MIL-STD-1753. (This is no longer the current Fortran standard.)

- Floating-point arithmetic for both compilers is based on IEEE standard 754-1985, and international standard IEC 60559:1989.


- In this document, "Standard" means conforming to the versions of the standards listed above. "Non-standard" or "Extension" refers to features that go beyond these versions of these standards.

The responsible standards bodies may revise these standards from time to time. The versions of the applicable standards to which these compilers conform may be revised or replaced, resulting in features in future releases of the Fortran compilers that create incompatibilities with earlier releases.
Extensions

Extensions to the standard FORTRAN 77 language include recursion, pointers, double-precision complex, quadruple-precision real, quadruple-precision complex, and many VAX™ and VMS™ FORTRAN 5.0 extensions, including NAMELIST, DO WHILE, structures, records, unions, maps, and variable formats. Multiprocessor FORTRAN includes automatic and explicit loop parallelization.

Sun FORTRAN 77 accepts many VMS extensions, so that programs originally written for VAX systems will port easily to Solaris.

Features implemented in Sun f77 that are not part of the applicable standards mentioned in “Standards Conformance” on page 11 are flagged with the special character mark ✠ throughout this manual.

Basic Terms

Some of the FORTRAN basic terms and concepts are:

- 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.

Character Set

The character set consists of the following:

- Uppercase and lowercase letters, A – Z and a – z
- Numerals 0 – 9
Special characters—The following table shows the special characters that are used for punctuation:

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>Space</td>
<td>Ignored in statements, except as part of a character constant</td>
</tr>
<tr>
<td>Tab</td>
<td>Tab</td>
<td>Establish the line as a tab-format source line *</td>
</tr>
<tr>
<td>=</td>
<td>Equals</td>
<td>Assignment</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
<td>Add, unary operator</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
<td>Subtract, unary operator</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
<td>Multiply, alternate returns, comments, exponentiation, stdin, stdout, list-directed I/O</td>
</tr>
<tr>
<td>/</td>
<td>Slash</td>
<td>Divide, delimit data, labeled commons, structures, end-of-record</td>
</tr>
<tr>
<td>()</td>
<td>Parentheses</td>
<td>Enclose expressions, complex constants, equivalence, parameter, or implicit groups, formats, argument lists, subscripts</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
<td>Separator for data, expressions, complex constants, equivalence groups, formats, argument lists, subscripts</td>
</tr>
<tr>
<td>.</td>
<td>Period</td>
<td>Radix point, delimiter for logical constants and operators, record fields</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe</td>
<td>Quoted character literals</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quote</td>
<td>Quoted character literals, octal constants *</td>
</tr>
<tr>
<td>$</td>
<td>Dollar sign</td>
<td>Delimit namelist input, edit descriptor, directives *</td>
</tr>
<tr>
<td>!</td>
<td>Exclamation</td>
<td>Comments *</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
<td>Array declarators, substrings, edit descriptor</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>Special functions: %REF, %VAL, %LOC *</td>
</tr>
<tr>
<td>&amp;</td>
<td>Ampersand</td>
<td>Continuation, alternate return, delimit namelist input; use in column 1 establishes the line as a tab-format source line *</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
<td>Request names in namelist group *</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
<td>Escape character *</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Angle brackets</td>
<td>Enclose variable expressions in formats *</td>
</tr>
</tbody>
</table>

Note the following usage and restrictions:
Uppercase or lowercase is not significant in the key words of FORTRAN statements or in symbolic names. The –U option of f77 makes case significant in symbolic names.

Most control characters are allowed as data even though they are not in the character set. The exceptions are: Control A, Control B, Control C, which are not allowed as data. These characters can be entered into the program in other ways, such as with the char() function.

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

For the backslash (\) character, you may need to use an escape sequence or use the –xl compiler option. For the newline (\n) character, you must use an escape sequence. See also TABLE 2-3.

Symbolic Names

The items in the following table can have symbolic names:

<table>
<thead>
<tr>
<th>TABLE 1-2</th>
<th>Items with Symbolic Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic constants</td>
<td>Labeled commons</td>
</tr>
</tbody>
</table>
| Variables | Namelist groups *
| Arrays | Main programs |
| Structures * | Block data |
| Records * | Subroutines |
| Record fields * | Functions |
|             | Entry points |

The following restrictions apply:

- Symbolic names can be any number of characters long. The standard is 6.
- Symbolic names consist of letters, digits, the dollar sign ($), and the underscore character (_). $ and _ are not standard.
- Symbolic names generally start with a letter—never with a digit or dollar sign ($). Names that start with an underscore (_) are allowed, but may conflict with names in the Fortran and system libraries.

Note – Procedure names that begin with exactly two underscores are considered special support functions internal to the compiler. Avoid naming functions or subroutines with exactly two initial underscores (for example _ _xfunc) as this will conflict with the compiler’s usage.
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 names are equivalent with the default in effect:

\[
\begin{align*}
\text{ATAD} &= 1.0E-6 \\
\text{Atad} &= 1.0e-6
\end{align*}
\]

The space character is not significant.

Example: These names are equivalent:

\[
\begin{align*}
\text{IF} ( \ X \ .LT. \ ATAD \ ) & \text{ GO TO } 9 \\
\text{IF} ( \ X \ .LT. \ A \ TAD \ ) & \text{ GO TO } 9 \\
\text{IF} (X.LT.ATAD) & \text{GOT09}
\end{align*}
\]

Here are some sample symbolic names:

**TABLE 1-3  Sample Symbolic Names**

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>X2</td>
<td>2X</td>
<td>Starts with a digit.</td>
</tr>
<tr>
<td>DELTA_TEMP</td>
<td>___DELTA_TEMP</td>
<td>Starts with an ___ (reserved for the compiler).</td>
</tr>
</tbody>
</table>
| Y$Dot    | Y|Dot    | There is an invalid character |.

In general, for any single program unit, different entities cannot have the same symbolic name. The exceptions are:

- 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
Program Units

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, which are invoked from other procedures or from the main program. The block data subprograms are handled by the loader.

Statements

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

Executable and Nonexecutable Statements

Every statement is either executable or nonexecutable. In general, if a statement specifies an action to be taken at runtime, it is executable. Otherwise, it is 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, nonexecutable statements are completed before execution of the first executable statement.
FORTRAN Statements

<table>
<thead>
<tr>
<th>TABLE 1-4 FORTRAN Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT*</td>
</tr>
<tr>
<td>ASSIGN*</td>
</tr>
<tr>
<td>Assignment*</td>
</tr>
<tr>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>BACKSPACE*</td>
</tr>
<tr>
<td>BLOCK DATA</td>
</tr>
<tr>
<td>BYTE</td>
</tr>
<tr>
<td>CALL*</td>
</tr>
<tr>
<td>CHARACTER</td>
</tr>
<tr>
<td>CLOSE*</td>
</tr>
<tr>
<td>COMMON</td>
</tr>
<tr>
<td>COMPLEX</td>
</tr>
<tr>
<td>CONTINUE*</td>
</tr>
<tr>
<td>DATA</td>
</tr>
<tr>
<td>DECODE*</td>
</tr>
<tr>
<td>DIMENSION</td>
</tr>
<tr>
<td>DO*</td>
</tr>
<tr>
<td>DO WHILE*</td>
</tr>
</tbody>
</table>

The asterisk (*) in the table indicates an executable statement.

Source Line Formats

A statement takes one or more lines; the first line is called the *initial line*; 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” on page 18.
- 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” on page 18.

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 can precede the tab.
- 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 –N1n option. *

Extended Lines

To extend 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 following DATA statement:

```
C   1         2         3         4         5         6         7
C23456789012345678901234567890123456789012345678901234567890123456789012
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 mark (!) 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
```

Directives

A directive passes information to a compiler in a special form of comment. *

Directives are also called compiler pragmas. There are two kinds of directives:
General directives

Parallel directives

See the Sun WorkShop Fortran User’s Guide and the Fortran Programming Guide for details on the specific directives available with f77.

General Directives

The form of a general directive is one of the following:

- `C$PRAGMA id`
- `C$PRAGMA id ( a [ , a ] ... ) [ , id ( a [ , a ] ... ) ] ...`
- `C$PRAGMA SUN id [=options]`

The variable `id` identifies the directive keyword; `a` is an argument.

Syntax

A directive has the following syntax:

- In column one, any of the comment-indicator characters `c, C, !`, or `*`
- In any column, the `!` comment-indicator character
- The next 7 characters are `$PRAGMA`, no blanks, any uppercase or lowercase

Rules and Restrictions

After the first eight characters, blanks are ignored, and uppercase and lowercase are equivalent, as in FORTRAN text.

Because it is a comment, a directive 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.

Parallelization Directives

Parallelization directives explicitly request the compiler attempt to parallelize the DO loop that follows the directive. The syntax differs from general directives.

Parallelization directives are only recognized when compilation options `-parallel` or `-explicitpar` are used. (f77 parallelization options are described in the Fortran Programming Guide.)

Parallelization directives have the following syntax:
■ The first character must be in column one.
■ The first character can be any one of c, C, *, or !.
■ The next four characters are $PAR, no blanks, either upper or lower case.
■ Next follows the directive keyword and options, separated by blanks.

The explicit parallelization directive keywords are:

TASKCOMMON, DOALL, DOSERIAL, and DOSERIAL*

Each parallelization directive has its own set of optional qualifiers that follow the keyword.

Example: Specifying a loop with a shared variable:

C$PAR DOALL SHARED(yvalue)

See the Fortran Programming Guide for details about parallelization and these directives. Fortran parallelization features require a Sun WorkShop HPC license.
Data Types and Data Items

This chapter describes the data types and data structures in Sun FORTRAN 77. Nonstandard features are tagged with a small black cross (✠).

Types

Except for specifically typeless constants, any constant, constant expression, variable, array, array element, substring, or function usually represents typed data. On the other hand, data types are not associated with the names of programs or subroutines, block data routines, common blocks, namelist groups, or structured records.

Rules for Data Typing

The name determines the 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 \texttt{IMPLICIT} statement, even to the extent of turning off all implicit typing with the \texttt{IMPLICIT NONE} statement. You can also turn off all implicit typing by specifying the \texttt{-u} compiler flag on the command line; this is equivalent to beginning each program unit with the \texttt{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 “Intrinsic Functions” on page 325.

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 \texttt{FUNCTION} statement, by preceding the word \texttt{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:

\begin{verbatim}
FUNCTION F ( X )
INTEGER F, X
F = X + 1
RETURN
END
\end{verbatim}

Example: Explicitly in its \texttt{FUNCTION} statement:

\begin{verbatim}
INTEGER FUNCTION F ( X )
INTEGER X
F = X + 1
RETURN
END
\end{verbatim}
Example: Implicitly by its name, as with variables:

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

Implicit typing can affect the type of a function, either by default implicit typing or by an `IMPLICIT` statement. You must make the data type of the function be the same within the function subprogram as it is in the calling program unit. The $f77$ compiler does no type checking across program units.

**Properties of Data Types**

This section describes the data types in Sun FORTRAN 77.

Default data declarations, those that do not explicitly declare a data size can have their meanings changed by certain compiler options. The next section, “Size and Alignment of Data Types” on page 31 summarizes data sizes and alignments and the effects of these options.

**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 eight-bit data item
- 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 the compiler expects `INTEGER` explicitly, it will not allow `BYTE`. 
Examples:

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

A BYTE item occupies 1 byte (8 bits) of storage, and is aligned on 1-byte boundaries.

**CHARACTER**

The character data type, CHARACTER, which has the synonym, CHARACTER*1, holds one character.

The character is enclosed in apostrophes (') or quotes ("), * Allowing quotes (") is nonstandard; if you compile with the -x1 option, quotes mean something else, and you must use apostrophes to enclose a string.

The data of type CHARACTER is always unsigned. A CHARACTER item occupies 1 byte (8 bits) of storage and is aligned on 1-byte boundaries.

**CHARACTER*\(n\)**

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

A CHARACTER*\(n\) data type occupies \(n\) bytes of storage and is aligned on 1-byte boundaries.

Every character string constant is aligned on 2-byte boundaries. 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 defaults to a synonym for 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.

The default size for a COMPLEX item (no size specified) is 8 bytes. The default alignment is on 4-byte boundaries. However, these defaults can be changed by compiling with certain special options (see “Size and Alignment of Data Types” on page 31).
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.

The size for COMPLEX*32 is 32 bytes.

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; the second represents the imaginary part.

The default size for DOUBLE COMPLEX with no size specified is 16.

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.

The default size for DOUBLE PRECISION with no size specified is 8 bytes.

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 “Data Representations” on page 365.
INTEGER

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

The default size for INTEGER with no size specified is 4, and is aligned on 4-byte boundaries. However, these defaults can be changed by compiling with certain special options (see “Size and Alignment of Data Types” on page 31).

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 is not recommended.

Generic functions return short or long integers depending on the default integer type. If a procedure is compiled with 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 compilation option is in effect. With -i2, the default length of LOGICAL quantities is 2 bytes.

Ordinary integers follow the FORTRAN rules about occupying the same space as a REAL variable. They are assumed to be equivalent to the C type long int, and 2-byte integers are of C type short int. These short integer and logical quantities do not obey the standard rules for storage association.

An INTEGER*2 occupies 2 bytes.

INTEGER*2 is aligned on 2-byte boundaries.

INTEGER*4 *

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

An INTEGER*4 occupies 4 bytes.

INTEGER*4 is aligned on 4-byte boundaries.

INTEGER*8 *

The integer data type, INTEGER*8, holds a signed 64-bit integer.

An INTEGER*8 occupies 8 bytes.

INTEGER*8 is aligned on 8-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.

The usual default size for an LOGICAL item with no size specified is 4, and is aligned on 4-byte boundaries. However, these defaults can be changed by compiling with certain special options.

LOGICAL*1

The one-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. An example:

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

A LOGICAL*1 item occupies one byte of storage.
LOGICAL*1 is aligned on one-byte boundaries.

LOGICAL*2

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

A LOGICAL*2 occupies 2 bytes.
LOGICAL*2 is aligned on 2-byte boundaries.

LOGICAL*4

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

A LOGICAL*4 occupies 4 bytes.
LOGICAL*4 is aligned on 4-byte boundaries.

LOGICAL*8

The logical data type, LOGICAL*8, holds the logical value .TRUE. or .FALSE. The value is defined the same way as for the LOGICAL data type.

A LOGICAL*8 occupies 8 bytes.

LOGICAL*8 is aligned on 8-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.

The usual default size for a REAL item with no size specified is 4 bytes, and is aligned on 4-byte boundaries. However, these defaults can be changed by compiling with certain special options.

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. The size for a REAL*16 item is 16 bytes. 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 of Data Types

Storage and alignment are always given in bytes. Values that can fit into a single byte are byte-aligned.

The size and alignment of types depends on various compiler options and platforms, and how variables are declared. The maximum alignment in COMMON blocks is to 4-byte boundaries.

Default data alignment and storage allocation can be changed by compiling with special options, such as -aligncommon, -f, -dalign, -dbl_align_all, -dbl, -xmemalign, -r8, -i2, and -xtypemap. The default descriptions in this manual assume that these options are not in force.

Refer to the Fortran User's Guide for details of specific compiler options.

The following table summarizes the default size and alignment, ignoring other aspects of types and options.

<table>
<thead>
<tr>
<th>Fortran 77 Data Type</th>
<th>Size</th>
<th>Default Alignment</th>
<th>Alignment in COMMON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SPARC</td>
<td>x86</td>
</tr>
<tr>
<td>BYTE X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHARACTER X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHARACTER*8 X</td>
<td>n</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COMPLEX X</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX*8 X</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DOUBLE COMPLEX X</td>
<td>16</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX*16 X</td>
<td>16</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX*32 X</td>
<td>32</td>
<td>8/16</td>
<td>—</td>
</tr>
<tr>
<td>DOUBLE PRECISION X</td>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>REAL X</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>REAL*4 X</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>REAL*8 X</td>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>REAL*16 X</td>
<td>16</td>
<td>8/16</td>
<td>—</td>
</tr>
<tr>
<td>INTEGER X</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*2 X</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER*4 X</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*8 X</td>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL X</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*1 X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL*2 X</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL*4 X</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*8 X</td>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
Note the following:

- REAL*16 and COMPLEX*32 are only available on SPARC only. In 64-bit environments (compiling with -xarch=v9 or v9a) the default alignment is on 16-byte (rather than 8-byte) boundaries, as indicated by 8/16 in the table.
- 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.

Compiling with options -i2,-r8, or -dbl changes the defaults for certain data declarations that appear without an explicit size:

### TABLE 2-2 Data Defaults Changed by -i2, -r8, -dbl

<table>
<thead>
<tr>
<th>Default Type</th>
<th>With -i2</th>
<th>With -r8 or -dbl</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>INTEGER*2</td>
<td>INTEGER*8</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>LOGICAL*2</td>
<td>LOGICAL*8</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL*4</td>
<td>REAL*8</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>REAL*8</td>
<td>REAL*16</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>COMPLEX*8</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>DOUBLE COMPLEX</td>
<td>COMPLEX*16</td>
<td>COMPLEX*32</td>
</tr>
</tbody>
</table>

Do not combine -i2 with -r8 as this can produce unexpected results. REAL*16 and COMPLEX*32 are SPARC only.

With -dbl or -r8, INTEGER and LOGICAL are allocated the larger space indicated above. This is done to maintain the FORTRAN requirement that an integer item and a real item have the same amount of storage. However, with -r8 8 bytes are allocated but only 4-byte arithmetic is done. With -dbl, 8 bytes are allocated and full 8-byte arithmetic is done. In all other ways, -dbl and -r8 produce the same results. A disadvantage of using -r8 or -dbl is that it also promotes DOUBLE PRECISION data to QUAD PRECISION, possibly degrading performance.

Use of the more flexible -xtypemap option is preferred over the older -r8 and -dbl options. Both -dbl and -r8 have their -xtypemap equivalents:

- On SPARC:
  
  -dbl  same as: -xtypemap=real:64,double:128,integer:64
  -r8   same as: -xtypemap=real:64,double:128,integer:mixed

- On x86:

  -dbl  same as: -xtypemap=real:64,double:64,integer:64
  -r8   same as: -xtypemap=real:64,double:64,integer:mixed

The mapping integer:mixed indicates 8 byte integers but only 4 byte arithmetic.
There are two additional possibilities on SPARC:

-xtypemap=real:64,double:64,integer:mixed
-xtypemap=real:64,double:64,integer:64

which map both default REAL and DOUBLE to 8 bytes, and should be preferable over using -r8 or -dbl.

Note that INTEGER and LOGICAL are treated the same, and COMPLEX is mapped as two REAL values. Also, DOUBLE COMPLEX will be treated the way DOUBLE is mapped.

Options -f or -dalign (SPARC only) force alignment of all 8, 16, or 32-byte data onto 8-byte boundaries. Option -dbl_align_all causes all data to be aligned on 8-byte boundaries. Programs that depend on the use of these options may not be portable.

See the Fortran User's Guide for details on these compiler options.

---

## Constants

A literal 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. (For a named constant, defined by a PARAMETER statement, the name defines the data type.)

There are three general kinds of constants:

- Arithmetic
- Logical
- Character

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

Here are the different kinds of arithmetic constants:

<table>
<thead>
<tr>
<th>Typed Constants</th>
<th>Typeless Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Binary</td>
</tr>
<tr>
<td>Double complex</td>
<td>Octal</td>
</tr>
<tr>
<td>Double precision</td>
<td>Hexadecimal</td>
</tr>
<tr>
<td>Integer</td>
<td>Hollerith</td>
</tr>
<tr>
<td>Real</td>
<td></td>
</tr>
</tbody>
</table>

---
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.

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.

Compiling with any of the options -i2, -dbl, -r8, or -xtypemap alters the default size of integer, real, complex, and double precision constants. These options are described in Chapter 2, and in the Fortran User’s Guide.

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 2-3.

Example: 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:

```f77
! Example: Null character string:

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

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:

```f77
! 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
```

```
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>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>\0</td>
<td>Null</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe, which does not terminate a string</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation mark, which does not terminate a string</td>
</tr>
<tr>
<td>\ \</td>
<td>\</td>
</tr>
<tr>
<td>\x</td>
<td>$x$, where $x$ is any other character</td>
</tr>
</tbody>
</table>

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 (or PARAMETER 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.
Example: Complex constants:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 9.01, .603 )</td>
<td></td>
</tr>
<tr>
<td>( +1.0, -2.0 )</td>
<td></td>
</tr>
<tr>
<td>( +1.0, -2 )</td>
<td></td>
</tr>
<tr>
<td>( 1, 2 )</td>
<td></td>
</tr>
<tr>
<td>( 4.51, )</td>
<td>Invalid — need second part</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 9.01D6, .603 )</td>
<td></td>
</tr>
<tr>
<td>( +1.0, -2.0D0 )</td>
<td></td>
</tr>
<tr>
<td>( 1D0, 2 )</td>
<td></td>
</tr>
<tr>
<td>( 4.51D6, )</td>
<td>Invalid — need second part</td>
</tr>
<tr>
<td>( +1.0, -2.0 )</td>
<td>Not DOUBLE COMPLEX — need a REAL*8</td>
</tr>
</tbody>
</table>

**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.
Example: 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 INTEGER*4 range (-2147483648, 2147483647), unless compiled with an option that promotes integers to 64 bits, in which case the range becomes INTEGER*8 (-9223372036854775808, 9223372036854775807). See “Size and Alignment of Data Types” on page 31.

Example: Integer constants:

-2147483648 Invalid—too small, error message
-2147483649
-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 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 43.

Long Integers *

Compiling with an option that promotes the range from INTEGER*4 (-21474836, 21474836) to INTEGER*8 (-9223372036854775808, 9223372036854775807). The integer constant is stored or passed as an 8-byte integer, data type INTEGER*8.

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 compiling with the –i2 option will cause it to be 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 compiled with the –i2 option, in which case 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 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.

Example: 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.

Example: 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.
Example: 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

The restrictions are:

- 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 are:

```
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
```

The restrictions are:
- 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 or an integer constant, such that it is followed by a quadruple-precision exponent. See “Real Constants” on page 39.*

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.

Example: Quadruple-precision constants:

```
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.
The restrictions are:

- 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 8-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.

These constants are not converted before use. However, in f77, they 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
```

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'FFFF99A'
WRITE(*,*) M, JCOUNT, TEMP
END
```
In the above example, 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:

```fortran
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:

```fortran
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.

For an octal notation, enclose a string of octal digits in apostrophes and append the letter O.

Example: Octal alternate notation for typeless constants:

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

For hexadecimals, enclose a string of hex digits in apostrophes and append the letter X.
Example: Hex alternate notation for typeless constants:

<table>
<thead>
<tr>
<th>Hex alternate notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ab'X</td>
<td>Invalid—missing trailing apostrophe</td>
</tr>
<tr>
<td>3fff'X</td>
<td>Not numeric—missing X</td>
</tr>
<tr>
<td>'1f'X</td>
<td>Invalid—invalid digit g</td>
</tr>
<tr>
<td>'3f'X</td>
<td>Invalid—missing X</td>
</tr>
<tr>
<td>'3g7'X</td>
<td>Invalid—invalid digit g</td>
</tr>
</tbody>
</table>

Here are the 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.

A Hollerith constant occupies 1 byte of storage for each character.
A Hollerith constant is aligned on 2-byte boundaries.

The FORTRAN standard does not have this old Hollerith notation, although the standard recommends implementing the Hollerith feature 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
```

The rules and restrictions on Hollerith constants are:

- 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.

### Fortran 95-Style Constants *

The Sun WorkShop Fortran 77 compiler recognizes the Fortran 95-style syntax for integer and real constants that allows literal specification of the size of the data item. In Fortran 95 terminology, a constant literal may include an optional trailing underscore followed by a “kind type parameter”. *
In the Sun Fortran 77 implementation, the “kind type parameter” is limited to the digits 1, 2, 4, 8, or 16, and its use specifies the data size, in bytes, of the literal constant. For example:

<table>
<thead>
<tr>
<th>Kind Type Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12_8</td>
<td>specifies an 8-byte integer constant, value = 12</td>
</tr>
<tr>
<td>12.012_16</td>
<td>specifies a 16-byte real constant, value = 12.012</td>
</tr>
<tr>
<td>1.345E-10_8</td>
<td>specifies an 8-byte real constant, value = 1.345E-10</td>
</tr>
<tr>
<td>(-1.5_8,.895E-3_8)</td>
<td>specifies a complex constant with 8-byte real and imaginary parts</td>
</tr>
</tbody>
</table>

With complex constants, the real and imaginary parts may be specified with different kind type parameters, (1.0_8,2.0_4), but the resulting data item will have the real and imaginary parts with the same size, taking the larger one specified.

This construction is valuable when calling subprograms with constant arguments when a specific data type is required, as in the following example:

```fortran
call suby(A,1.5_8,0_8,Y)
... 
subroutine suby(H0, M, N, W)
INTEGER *8 M, N,
... 
```

---

**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. This does not include arrays, array elements, records, or record fields, so this definition is more restrictive than the usual usage of the word “variable.”

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 “Types” on page 23 for more details on the rules for data typing.

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.
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.

---

**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” on page 51 for details.

You can declare an array in any of the following statements:

- DIMENSION statement
- COMMON statement
- Type statements: BYTE, CHARACTER, INTEGER, REAL, and so forth

---

**Array Declarators**

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

The syntax of an array declarator is:

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

where:

- **a** is the name of the array
- **d** is a dimension declarator

A dimension declarator has the form:

```
  [  dl :  ]  du
```

where:

- **dl** is the lower dimension bound
- **du** is the upper dimension bound

An array must appear only once in an array declarator within a program unit (main program, subroutine, function, or block common). The compiler flags multiple or duplicate array declarations within the same unit as errors.

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

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:

```plaintext
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:

```plaintext
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:

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

Example: Lower bounds other than one:

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

Example: Character arrays:

```plaintext
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.

The following restrictions on bounds apply:
- Both the upper and the lower bounds can be negative, zero, or positive.
- The upper bound must be greater than or equal to the lower bound.
- 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 that is a dummy argument or local array* with one or more of its dimensions or bounds as an expression of integer variables that are either themselves dummy arguments, or are in a common block.

You can declare adjustable arrays in the usual **DIMENSION** 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 arrays;**

```
SUBROUTINE POPUP ( A, B, N )
COMMON / DEFS / M, L
REAL A(3:5, L, M:N), B(N+1:2*N)  ! These arrays are dummy args
REAL C(N+1,2*N)  ! This array is local
```

The restrictions are:
- 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.
- You cannot declare an adjustable array in **COMMON**.

If the array is local to the routine, memory is allocated on entry to the routine and deallocated on return to the caller.*

### 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.

The following f77 extensions allow you to:
- declare assumed-size arrays in a RECORD statement, if that RECORD statement is not inside a structure declaration block.
- use an assumed-size array as a unit identifier for an internal file in an I/O statement.
- use an assumed-size array as a runtime format specifier in an I/O statement.

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

```
SUBROUTINE PULLDOWN ( A, B, C )
   INTEGER A(5, *), B(*), C(0:1, 2:*)
```

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. 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:

\( (s[,s]...) \)

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 declaration, you can assign a value to a particular element, as follows:

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

The above code 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, complex, logical, or byte expression. According to the FORTRAN Standard, it must be an integer expression.
- A subscript expression can 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 \((L_1, ..., L_n)\), where each \(L_i\) is the lower bound of the respective dimension, references the first element of the array.
- A subscript of the form \((U_1, ..., U_n)\), where each \(U_i\) 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.
Subscript expressions cannot exceed the range of INTEGER*4 in 32-bit environments. It is not controlled, but if the subscript expression is not in the range (-2147483648, 2147483647), then the results are unpredictable. When compiled for 64-bit environments, INTEGER*8 subscript expressions are allowed.

Array Ordering

Array elements are usually considered as being arranged with the first subscript as the row number and the second subscript as the column number. This corresponds to traditional mathematical $n \times m$ matrix notation:

\[
\begin{array}{cccc}
  a_{1,1} & a_{1,2} & a_{1,3} & \ldots & a_{1,m} \\
  a_{2,1} & a_{2,2} & \ldots & & a_{2,m} \\
  \vdots & \vdots & \ddots & \ddots & \vdots \\
  a_{n,1} & a_{n,2} & \ldots & & a_{n,m} \\
\end{array}
\]

Element $a_{i,j}$ is located in row $i$, column $j$.

For example:

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

The elements of $A$ are conceptually arranged 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.
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. They cannot exceed the range of \texttt{INTEGER*4} on 32-bit environments. If the expression is not in the range \((-2147483648, 2147483647)\), then the results are unpredictable. When compiled for 64-bit environments, the substring character position expressions can be in the range of \texttt{INTEGER*8}.

Example: The string with initial character from the \( I \)th character of \( S \) and with the last character from the \( L \)th character of \( S \):

\[ S(I:L) \]

In the above example, there are \( L-I+1 \) characters in the substring.

The following string has an initial character from the \( M \)th character of the array element \( A(J,K) \), with the last character from the \( N \)th character of that element.

\[ A(J,K)(M:N) \]

In the above example, there are \( N-M+1 \) characters in the substring.

Here are the 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 can be used on the left and right sides of assignments and as procedure actual arguments.
• Substrings must not be overlapping. $\text{ASTR}(2:4) = \text{ASTR}(3:5)$ is illegal.

Examples: Substrings—the value of the element in column 2, row 3 is e23:

```
demo% cat sub.f

character v*8 / 'abcdefg' /,
&   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% f77 sub.f
sub.f:
MAIN:
demo% a.out
  cde
  abcdefgh
  abcdefgh
  abcdefgh
e11
e21
e12
e22
e13
e23
  13
demo%
```
Structures

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

Just as an array is a collection of elements of the same type, 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 in the following subsections.

Syntax

The structure declaration has the following 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*  
Defines a field of the record.  
*field-declaration* is defined in the next section.

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, which is described later
- 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 61.

Rules and Restrictions for Structures

Note the following:

- 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. 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. %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 may be a useful feature if you
want 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.

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 “Data Representations” on page 365 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:

```
RECORD /structure-name/ record-list
    [./structure-name/ record-list]
...    [./structure-name/ record-list]
```

<table>
<thead>
<tr>
<th>structure-name</th>
<th>Name of a previously declared structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>record-list</td>
<td>List of variables, arrays, or arrays with dimensioning and index ranges, separated by commas.</td>
</tr>
</tbody>
</table>

Example: A RECORD that uses 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; LINE is an array of 10 such records.

Note the following 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 62.

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

\[ \text{record-name} \cdot \text{field-name} \cdot \ldots \cdot \text{field-name} \]

- `record-name`: Name of a previously defined record variable
- `field-name`: Name of a field in the record immediately to the left.

Example: References that are based on structure and records of the 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.
- 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 one 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 \texttt{SALE} contains three fields, \texttt{BUYER}, \texttt{QUANTITY}, and \texttt{ITEM}, where \texttt{ITEM} is a record with the structure, \texttt{/PRODUCT/}.

**Structure Within a Structure**

You can nest a declaration within a declaration.

Example: If \texttt{/PRODUCT/} is \textit{not} declared previously, then you can declare it within the declaration of \texttt{SALE}:

```plaintext
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 \texttt{SALE} still contains the same three fields as in the prior example: \texttt{BUYER}, \texttt{QUANTITY}, and \texttt{ITEM}. The field \texttt{ITEM} is an example of a \textit{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 (\texttt{PRODUCT} and \texttt{SALE}, from the previous examples, are defined in the current program unit):

```plaintext
... RECORD /SALE/ JAPAN ...
... N = JAPAN.QUANTITY
I = JAPAN.ITEM.ID ...
```
Rules and Restrictions for Substructures

Note the following:
- 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; however, doing so might be questionable programming practice.
- You can use the pseudo-name, %FILL, to align fields in a record, and create 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.

Syntaxes

The syntax of a union declaration is:

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

The syntax of a map declaration is as follows.

```fortran
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:
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`:

```plaintext
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.

---

**Pointers**

The `POINTER` statement establishes pairs of variables and pointers. * Each pointer contains the address of its paired variable.
Syntax Rules

The POINTER statement has the following syntax:

\[
\text{POINTER ( } p1, v1 \ [ , ( p2, v2 ) \ldots ]\]
\]

where:
- \( v1, v2 \) are pointer-based variables, also called pointees.
- \( p1, p2 \) 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 just called a based variable, or a pointee. The pointer is the integer variable that contains the address. (Variable names appearing on a POINTER statement are considered volatile by the compiler.)

Example: A simple POINTER statement:

\[
\text{POINTER ( } P, V \text{ )}
\]

Here, \( V \) is a pointee, and \( P \) is its associated pointer.

See “POINTER” on page 212, for more examples.

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 and should not appear in a type declaration.
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 is always from its associated pointer.
Address and Memory

No storage for the variable is allocated when a pointer-based variable is defined, so you must 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.

The `loc()`, `malloc()`, and `free()` routines associate and deassociate memory addresses with pointers. (These routines are described in Chapter 6.)

When compiled for 64-bit environments, pointers declared by the `POINTER` statement are `INTEGER*8` values.

Address by `LOC()` Function

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

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

```fortran
* 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 prints an `E`.

When compiled for 64-bit environments, `LOC()` returns an `INTEGER*8` value. The receiving variable must be either a pointer or an `INTEGER*8` variable to avoid possible address truncation.

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.
Example: Memory allocation for pointers, by MALLOC:

```fortran
COMPLEX Z
REAL X, Y
POINTER ( P1, X ), ( P2, Y ), ( P3, Z )

P1 = MALLOC ( 10000 )
```

In the above example, MALLOC() allocates 10,000 bytes of memory and associates the address of that block of memory with the pointer P1.

Deallocation of Memory by FREE() Subroutine

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:

```fortran
POINTER ( P1, X ), ( P2, Y ), ( P3, Z )

P1 = MALLOC ( 10000 )

CALL FREE ( P1 )
```

In the above example, MALLOC() allocates 10,000 bytes of memory, which are associated with pointer P1. FREE() later returns those same 10,000 bytes to the memory manager.

Special Considerations

Here are some special considerations when working with pointers and memory allocation with malloc(), loc(), and free():

- 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.

Address expressions cannot exceed the range of INTEGER*4 on 32-bit environments. If the expression is not in the range (-2147483648, 2147483647), then the results are unpredictable.

When compiling for 64-bit environments, use malloc64() to access the 64-bit address space. Routine malloc64() takes an INTEGER*8 argument and returns a 64-bit pointer value. In 64-bit programs, pointers defined by the POINTER statement are 64-bit INTEGER*8 values. See the Fortran Library Reference Manual and the malloc(3F) man pages.

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 -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)   This programming unit does
POINTER ( P, V )       nothing else with P other than
P = MALLOC(10000)      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 -O4, then avoid using pointers in CALC.

Some Problematic Code Practices

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: Code that could cause trouble with -O3 or -O4:

```
COMMON A, B, C
POINTER ( P, V )
P = LOC(A) + 4   Possible problems here if optimized
```

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

This chapter discusses Fortran expressions and how they are evaluated.

Expressions, Operators, and Operands

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

There are three kinds of expressions:

- 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)
- An expression
Arithmetic Expressions

An arithmetic expression evaluates to a single arithmetic value, and its operands have the following types. * 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>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or Unary Minus</td>
</tr>
<tr>
<td>+</td>
<td>Addition or Unary Plus</td>
</tr>
</tbody>
</table>

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:

\[ a \odot b \]

where \(a\) and \(b\) are operands, and \(\odot\) 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:

\[ \oplus \ b \]

where \( b \) is an operand, and \( \oplus \) is either of the - or + operators.

Examples: Unary operators:

- \(-z\)
- \(+B\)

## Basic Arithmetic Expressions

Each arithmetic operator is shown in its basic expression in the following table:

### TABLE 3-2  Arithmetic Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^z )</td>
<td>Raise ( a ) to the power ( z )</td>
</tr>
<tr>
<td>( a/z )</td>
<td>Divide ( a ) by ( z )</td>
</tr>
<tr>
<td>( a*z )</td>
<td>Multiply ( a ) by ( z )</td>
</tr>
<tr>
<td>( a-z )</td>
<td>Subtract ( z ) from ( a )</td>
</tr>
<tr>
<td>(-z)</td>
<td>Negate ( z )</td>
</tr>
<tr>
<td>( +z)</td>
<td>Add ( z ) to ( a )</td>
</tr>
<tr>
<td>(+z)</td>
<td>Same as ( z )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>First</td>
</tr>
<tr>
<td>(* /)</td>
<td>Second</td>
</tr>
<tr>
<td>+ -</td>
<td>Last</td>
</tr>
</tbody>
</table>

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

\( F \ ** S \ ** Z \)
The above is evaluated as:

\[ F \; ** \; (S \; ** \; Z) \]

`$`77 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 `*`. It first does the `*`, then the `−`, then the `**`.

**Mixed Mode**

Except for BYTE operands, if both operands have the same type, then the resulting value has that type. (The result of an arithmetic operation with both operands type BYTE is default integer.) 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. The ranking is summarized in the following table:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE or LOGICAL*1</td>
<td>1 (Weakest)</td>
</tr>
<tr>
<td>LOGICAL*2</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>3</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>5</td>
</tr>
<tr>
<td>INTEGER*8</td>
<td>6</td>
</tr>
<tr>
<td>LOGICAL*8</td>
<td>6</td>
</tr>
<tr>
<td>REAL*4 (REAL)</td>
<td>6</td>
</tr>
<tr>
<td>REAL*8 (DOUBLE PRECISION)</td>
<td>7</td>
</tr>
<tr>
<td>REAL*16 (QUAD PRECISION) (SPARC only)</td>
<td>8</td>
</tr>
<tr>
<td>COMPLEX*8 (COMPLEX)</td>
<td>9</td>
</tr>
<tr>
<td>COMPLEX*16 (DOUBLE COMPLEX)</td>
<td>10</td>
</tr>
<tr>
<td>COMPLEX*32 (QUAD COMPLEX) (SPARC only)</td>
<td>11 (Strongest)</td>
</tr>
</tbody>
</table>
Note – REAL*4, INTEGER*8, and LOGICAL*8 are of the same rank, but they can be the results of different pairs of operands. For example, INTEGER*8 results if you combine INTEGER*8 and any of the types between 1-5. Likewise, REAL*4 results if one of the operands is REAL*4, and the other is any of the types between 1-5. LOGICAL*8 dictates only the 8-byte size of the result.

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

\[ R \times I \]

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

Rules

Note these 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: An expression that evaluates to zero:

\[ \frac{2}{3} + \frac{3}{4} \]

- When an INTEGER*8 operand is mixed with REAL*4 operands, the result is REAL*8.

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. However, \((2/3) \times 4.0\) 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 in 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. 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 logics are converted to integers, and the result is an integer.

Arithmetic Assignment

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

```
v = e
```

<table>
<thead>
<tr>
<th>v</th>
<th>Numeric variable, array element, or record field</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Arithmetic expression, a character constant, or a logical expression</td>
</tr>
</tbody>
</table>
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 practice is nonstandard and may not be portable.

<table>
<thead>
<tr>
<th>Type of ( v )</th>
<th>Conversion of ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER<em>2, INTEGER</em>4, or INTEGER*8</td>
<td>INT(( e ))</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL(( e ))</td>
</tr>
<tr>
<td>REAL*8</td>
<td>DBLE(( e ))</td>
</tr>
<tr>
<td>REAL*16 (SPARC only)</td>
<td>QREAL(( e )) (SPARC only)</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>DBLE(( e ))</td>
</tr>
<tr>
<td>COMPLEX*8</td>
<td>CMPLX(( e ))</td>
</tr>
<tr>
<td>COMPLEX*16</td>
<td>DCMPLX(( e ))</td>
</tr>
<tr>
<td>COMPLEX*32 (SPARC only)</td>
<td>QCMPPLX(( e )) (SPARC only)</td>
</tr>
</tbody>
</table>

\textbf{Note} – Compiling with any of the options \( -i2, -dbl, -r8, \) or \( -xtypemap \) will have an effect on the assumed type of \( e \). This is discussed in Chapter 2. See also the \textit{Fortran User’s Guide} for a description of these options.
Character Expressions

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a // z</td>
<td>Concatenate a with z.</td>
</tr>
</tbody>
</table>

The result of concatenating two strings is a third string that contains 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, assuming C, S, and R.C are characters:

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

Note the following (nonstandard) exceptions:*

■ **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:

```plaintext
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
```

<table>
<thead>
<tr>
<th>v</th>
<th>Expression giving the value to be assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Expression giving the value to be assigned</td>
</tr>
<tr>
<td>v</td>
<td>Variable, array element, substring, or character record field</td>
</tr>
</tbody>
</table>

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 following program below displays joinedΔΔ:

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

Also, this program displays the equal string:

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

Example: Character assignment:

```fortran
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 are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Receiving Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>'z\Delta'</td>
<td>A trailing blank</td>
</tr>
<tr>
<td>C3</td>
<td>'uvw'</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>'ABxyz'</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>'ABxyzzz'</td>
<td>The final 'z' comes from C2</td>
</tr>
</tbody>
</table>

Control Character (^G)
Example 4: A Hollerith assignment:

```
CHARACTER S*4
INTEGER I2*2, I4*4
REAL R
S = 'wxyz'
I2 = 'yz'
I4 = 'wxyz'
R = 'wxyz'
```

Rules of Assignment

Here are the rules for character assignments:

- 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 substring assignment cannot overlap. See the “Substrings” on page 54.
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**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Standard Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>Logical conjunction</td>
</tr>
<tr>
<td>.OR.</td>
<td>Logical disjunction (inclusive OR)</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>Logical nonequivalence</td>
</tr>
<tr>
<td>.XOR.</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>.EQV.</td>
<td>Logical equivalence</td>
</tr>
<tr>
<td>.NOT.</td>
<td>Logical negation</td>
</tr>
</tbody>
</table>

The period delimiters are necessary.

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

Logical operators are evaluated according to the following precedence:

**TABLE 3-5  Logical Operator Precedence**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.</td>
<td>Highest</td>
</tr>
<tr>
<td>.AND.</td>
<td></td>
</tr>
<tr>
<td>.OR.</td>
<td></td>
</tr>
<tr>
<td>.NEQV., .XOR., .EQV.</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

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

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

**TABLE 3-6  Operator Precedence**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Highest</td>
</tr>
<tr>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following table shows the meanings of simple expressions:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X .AND. Y</td>
<td>Both X and Y are true.</td>
</tr>
<tr>
<td>X .OR. Y</td>
<td>Either X or Y, or both, are true.</td>
</tr>
<tr>
<td>X .NEQV. Y</td>
<td>X and Y are not both true and not both false.</td>
</tr>
<tr>
<td>X .XOR. Y</td>
<td>Either X or Y is true, but not both.</td>
</tr>
<tr>
<td>X .EQV. Y</td>
<td>X and Y are both true or both false.</td>
</tr>
<tr>
<td>.NOT. X</td>
<td>Logical negation.</td>
</tr>
</tbody>
</table>

Logical operations on operands of different sizes result in a logical value promoted to the size of the largest operand. For example, L2 .AND. L4 gives a LOGICAL*4 result if L2 is LOGICAL*2 and L4 is LOGICAL*4.

BYTE * variables are treated the same as LOGICAL*1.

**Logical Assignment**

This is the syntax for the assignment of the value of a logical expression to a logical variable:

\[ v = e \]

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

Execution of a logical assignment statement causes evaluation of the logical expression \( e \) and assignment of the resulting value to \( v \). 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. (All non-zero values are treated as .TRUE., and zero is .FALSE.) This practice is nonstandard, however, and is not portable.
Example: A logical assignment:

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

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**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT.</td>
<td>Less than</td>
</tr>
<tr>
<td>.LE.</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>.EQ.</td>
<td>Equal</td>
</tr>
<tr>
<td>.NE.</td>
<td>Not equal</td>
</tr>
<tr>
<td>.GT.</td>
<td>Greater than</td>
</tr>
<tr>
<td>.GE.</td>
<td>Greater than or equal</td>
</tr>
</tbody>
</table>

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:

```fortran
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  STR1 and STR2 are character type
S .EQ. 'a'      S is character type
```
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.

### 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 intrinsic functions called with constant arguments.

Examples: Constant expressions:

```fortran
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 a 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; a warning is issued.
Example: Exponentiation to a floating-point power is not allowed:

```fortran
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%
```

---

**Record Assignment**

The general form of record assignment is:

\[ v = e \]

<table>
<thead>
<tr>
<th>( v )</th>
<th>A record or record field</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e )</td>
<td>A record or record field</td>
</tr>
</tbody>
</table>

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: A record assignment and a record-field assignment:

```plaintext
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.

---

### 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 that affects the value of any other function reference in the same statement.
This chapter describes the statements recognized by the Sun WorkShop
FORTRAN 77 compiler, f77. Nonstandard features are indicated by the symbol "*".
(See Chapter 1 for a discussion of the conforming standards). A table of sample
statements appears in Appendix B.
ACCEPT

The ACCEPT * statement reads from standard input and requires the following syntax:

ACCEPT f [, iolist]

ACCEPT grname

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Format identifier</td>
</tr>
<tr>
<td>iolist</td>
<td>List of variables, substrings, arrays, and records</td>
</tr>
<tr>
<td>grname</td>
<td>Name of the namelist group</td>
</tr>
</tbody>
</table>

Description

ACCESS f [, iolist] is equivalent to READ f [, iolist] and is for compatibility with older versions of FORTRAN. An example of list-directed input:

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

The ASSIGN statement assigns a statement label to a variable.

\texttt{ASSIGN \ s \ TO \ i}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{s}</td>
<td>Statement label</td>
</tr>
<tr>
<td>\texttt{i}</td>
<td>Integer variable</td>
</tr>
</tbody>
</table>

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. It can not be declared INTEGER*2.

After assigning a statement label to a variable, you can reference it in:

- An assigned \texttt{GO TO} statement
- An input/output statement, as a format identifier

Restrictions

The variable must be assigned a statement label \textit{before} referencing it as a label in an assigned \texttt{GO TO} statement, or as a format identifier.

While \(i\) is assigned a statement label value, do \textit{no} arithmetic with \(i\).

On 64-bit platforms, the actual value stored in variable \(i\) by the ASSIGN statement is not available to the program, except by the assigned \texttt{GO TO} statement, or as a format identifier in an I/O statement. Also, only variables set by an ASSIGN statement can be used in an assigned \texttt{GO TO} or as a format identifier.
Examples

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

```fortran
IF(LB.EQ.0) ASSIGN 9 TO K
...
GO TO K
...
9 AKX = 0.0
```

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

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

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

\[ v = e \]

**Parameter** | **Description**
---|---
\( v \) | Variable, substring, array element, record, or record field
\( e \) | Expression giving the value to be assigned

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 following table.

<table>
<thead>
<tr>
<th>Type of ( v )</th>
<th>Type of ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER<em>2, INTEGER</em>4, or INTEGER*8</td>
<td>INT(e)</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL(e)</td>
</tr>
<tr>
<td>REAL*8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>REAL*16 (SPARC only)</td>
<td>QREAL(e) (SPARC only)</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>DBLE(e)</td>
</tr>
<tr>
<td>COMPLEX*8</td>
<td>CMPLX(e)</td>
</tr>
<tr>
<td>COMPLEX*16</td>
<td>DCMPLX(e)</td>
</tr>
<tr>
<td>COMPLEX*32 (SPARC only)</td>
<td>QCMLPX(e) (SPARC only)</td>
</tr>
</tbody>
</table>
Note – Compiling with any of the options -i2, -dbl, -r8, or -xtypemap can alter the default data size of variables and expressions. This is discussed in Chapter 2. See also the *Fortran User’s Guide* for a description of these options.

Example: An assignment statement:

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

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

```fortran
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 $v$. 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 provides more details on the size of logical variables.

**Character Assignment**

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
-ax1 option, because, in that case, a quote introduces an octal constant. The
characters are transferred to the variables without any conversion of data, and 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:

- Both e and v are fields with the same elementary data type.
- Both e and v are records with the same number of fields such that corresponding
  fields are the same elementary data type.
- Both e and v are records 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 details on the
structure of records.
Examples

Example 1: Arithmetic assignment:

```fortran
INTEGER I2*2, J2*2, I4*4
REAL R1, QP*16 ! (REAL*16 is SPARC only)
DOUBLE PRECISION DP
COMPLEX C8, C16*16, QC*32 ! (COMPLEX*32 is 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:

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

Example 3: Hollerith assignment:

```fortran
CHARACTER S*4
INTEGER I2*2, I4*4
REAL R
S = 4Hwxyz
I2 = 2Hyz
I4 = 4Hwxyz
R = 4Hwxyz
```
Example 4: Character assignment:

```plaintext
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
BELL = CHAR(7) Control Character (^G)
```

The results of the above are:

<table>
<thead>
<tr>
<th>C2</th>
<th>receives 'z\Delta' a trailing blank</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>receives 'uvw'</td>
</tr>
<tr>
<td>C5</td>
<td>receives 'ABxyz'</td>
</tr>
<tr>
<td>C6</td>
<td>receives 'ABxyzz' an extra z left over from C5</td>
</tr>
<tr>
<td>BELL</td>
<td>receives 07 hex Control-G, a bell</td>
</tr>
</tbody>
</table>

Example 5: Record assignment and record field assignment:

```plaintext
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
```
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.

AUTOMATIC vlist

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlist</td>
<td>List of variables and arrays</td>
</tr>
</tbody>
</table>

**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. See also the discussion of the -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:

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

Local variables and arrays are static by default, so in general, there is no need to use SAVE. You should use SAVE to ensure portability. Also, SAVE is safer if you leave a subprogram by some way other than a RETURN.

**Restrictions**

Automatic variables and arrays cannot appear in DATA or SAVE statements.
Arguments and function values cannot appear in DATA, RECORD, STATIC, or SAVE statements because f77 always makes them automatic.

Examples

Example: Some other uses of AUTOMATIC:

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

Example: Structures are unpredictable if AUTOMATIC:

```fortran
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%
```

Restrictions

An AUTOMATIC statement and a type statement cannot be combined to make an AUTOMATIC type statement. For example, AUTOMATIC REAL X does not declare the variable X to be both AUTOMATIC and REAL; it declares the variable REALX to be AUTOMATIC.
BACKSPACE

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

BACKSPACE  u

BACKSPACE ( [UNIT= ] u [, IOSTAT=ios ] [, ERR=s ] )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>Unit identifier of the external unit connected to the file</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier, integer variable, or an integer array element</td>
</tr>
<tr>
<td>s</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 77 Standard, and is unpredictable. We do not recommend using a BACKSPACE statement on a direct-access file or an append access file. BACKSPACE on a file opened as FORM=’BINARY’ is not allowed and causes a runtime error.

Execution of the BACKSPACE statement modifies the file position, as follows:

<table>
<thead>
<tr>
<th>Prior to Execution</th>
<th>After Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning of the file</td>
<td>Remains unchanged</td>
</tr>
<tr>
<td>Beyond the endfile record</td>
<td>Before the endfile record</td>
</tr>
<tr>
<td>Beginning of the previous record</td>
<td>Start of the same record</td>
</tr>
</tbody>
</table>
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
```
BLOCK DATA

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

```
BLOCK DATA  [name]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Symbolic name of the block data subprogram in which the BLOCK DATA statement appears. This parameter is optional. It is a global name.</td>
</tr>
</tbody>
</table>

**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
```
BYTE

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

BYTE v [/c/] ...

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>c</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

Description

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

Example

```
BYTE BIT3 /8/, C1/'W'/, M/127/, SWITCH/.FALSE./
```
CALL

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

CALL sub ([ [ar[, ar]...]])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>Name of the subroutine to be called</td>
</tr>
<tr>
<td>ar</td>
<td>Actual argument to be passed to the subroutine</td>
</tr>
</tbody>
</table>

Description

Arguments are separated by commas.

The FORTRAN 77 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 checks this only when the -XlistE option is on.

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; for a list of the intrinsics that cannot be actual arguments, see TABLE 4-2.
- 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 constructs as:

- Constant
- 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 that references 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.

**Note** – A CALL to a subprogram defined as a FUNCTION rather than as a SUBROUTINE will cause unexpected results and is not recommended. The compiler does not automatically detect such inappropriate CALLs and no warning is issued unless the -Xlist option is specified.

**Examples**

Example 1: Character string:

```
CHARACTER *25 TEXT
TEXT = 'Some kind of text string'
CALL OOPS ( TEXT )
END

SUBROUTINE OOPS ( S )
   CHARACTER S*(*)
   WRITE (*,*) S
END
```
Example 2: Alternate return:

```fortran
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:

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

Example 4: Array, array element, and variable:

```fortran
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
```

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.
Example 5: A structured record and field; the record is nonstandard:

```fortran
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.
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.

```
CHARACTER [* len [, ] ] v [* len /c/] ] ...  
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>len</td>
<td>Length in characters of the symbolic constant, variable, array element, or function</td>
</tr>
<tr>
<td>c</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

**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 length is defined in referencing the program unit, and the dummy argument has the length of the actual 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 in Example 5 in the next section.

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 two examples are equivalent to the nonstandard variation: *

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

There are no null (zero-length) character-string variables. A one-byte character string assigned a null constant has the length zero.

Example 2: No null character-string variables:

```
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; because of the declaration, the intrinsic function LEN(S) 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 SWAN( A )
CHARACTER A*32
```

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

```
SUBROUTINE SWAN( A )
CHARACTER A*(*)
...
Example 5: Symbolic constant with parenthesized asterisk:

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

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

Example 6: The `LEN` intrinsic function:

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

The above program displays 17, not 3.
CLOSE

The CLOSE statement disconnects a file from a unit.

CLOSE([UNIT=] u [, STATUS= sta] [, IOSTAT= ios] [, ERR= s])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>Unit identifier for an external unit. If UNIT= is not used, then u must be first.</td>
</tr>
<tr>
<td>sta</td>
<td>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, the default sta is KEEP.</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier—ios must be an integer variable or an integer array element.</td>
</tr>
<tr>
<td>s</td>
<td>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.</td>
</tr>
</tbody>
</table>

Description

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 status at normal program termination. Regardless of the specified status, 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 or file disconnected by the execution of a CLOSE statement can be connected again to the same, or a different, file or unit.

**Note** – For tape I/O, use the **OPEN()** routines.

Examples

Example 1: Close and keep:

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

Example 2: Close and delete:

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

Example 3: Close and keep a scratch file even though the status is SCRATCH:

```fortran
OPEN ( 2, STATUS='SCRATCH')
...
CLOSE ( 2, STATUS='KEEP', IOSTAT=I )
```
COMMON

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

```
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cb</td>
<td>Common block name</td>
</tr>
<tr>
<td>nlist</td>
<td>List of variable names, array names, and array declarators</td>
</tr>
</tbody>
</table>

**Description**

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.

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. However, blank common blocks within a program are not required to 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. See Example 2.

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

Example 1: Unlabeled common and labeled common:

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

In the above example, V and M are in the unlabeled common block; 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:

```fortran
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:

```fortran
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:

```fortran
COMMON /X/ A
REAL B(2)
EQUIVALENCE ( A, B(2))  Not allowed
```
COMPLEX

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

\[
\text{COMPLEX} \ [\ast\text{len}[,]] \ v \ [\ast\text{len}[/c/]] \ [, v [\ast\text{len}[/c/]] \ldots
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v)</td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>(len)</td>
<td>Either 8, 16, or 32, the length in bytes of the symbolic constant, variable, array element, or function (32 is SPARC only)</td>
</tr>
<tr>
<td>(c)</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

Description

The declarations can be: COMPLEX, COMPLEX*8, COMPLEX*16, or COMPLEX*32. Specifying the size is nonstandard.*
**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.

If you do not specify the size, a default size is used.

The default size for a declaration such as `COMPLEX W` can be altered by compiling with any of the options `-dbl`, `-r8`, or `-xtypemap`. See the discussion in Chapter 2 for details.

**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.

**Comments**

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.

There are specific complex functions for quad precision (SPARC only). In general, where there is a specific `REAL` 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 are: `SIN()`, `CSIN()`, `CDSIN()`, `CQSIN()`. 
Examples

Example 1: Complex variables. These statements are equivalent.

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

Example 2: Initialize complex variables:

```
COMPLEX U/(1.0, 9.0)/, V/(4.0, 5)/
```

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

Example 3: Double complex, with initialization:

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

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, with initialization (SPARC only):

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

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 which are nonstandard (SPARC only):

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

The CONTINUE statement is a “do-nothing” statement.

\[ \text{[label]} \ \text{CONTINUE} \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Executable statement number</td>
</tr>
</tbody>
</table>

Description

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

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
   . . .
```
DATA

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

\texttt{DATA nlist / clist / [ [ , ] nlist / clist / ] ...}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{nlist}</td>
<td>List of variables, arrays, array elements, substrings, and implied \texttt{DO} lists separated by commas</td>
</tr>
<tr>
<td>\texttt{clist}</td>
<td>List of the form: \texttt{c [ , c ] ...}</td>
</tr>
<tr>
<td>\texttt{c}</td>
<td>One of the forms: \texttt{c} or \texttt{r*c}, and \texttt{c} is a constant or the symbolic name of a constant.</td>
</tr>
<tr>
<td>\texttt{r}</td>
<td>Nonzero, unsigned integer constant or the symbolic name of such constant</td>
</tr>
</tbody>
</table>

Description

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

\texttt{r*c} is equivalent to \texttt{r} successive occurrences of the constant \texttt{c}.

A DATA statement is a nonexecutable statement, and must appear after all specification statements, but it can be interspersed with statement functions and executable statements, although this is non-standard.*

\textbf{Note} – Initializing a local variable in a DATA statement after an executable reference to that variable is flagged as an error when compiling with the \texttt{-stackvar} option. See the Sun WorkShop Fortran User’s Guide.

Taking into account the repeat factor, the number of constants in \texttt{clist} must be equal to the number of items in \texttt{nlist}. The appearance of an array in \texttt{nlist} is equivalent to specifying a list of all elements in that array. Array elements can be indexed by constant subscripts only.

Automatic variables or arrays cannot appear on a DATA statement.

Normal type conversion takes place for each noncharacter member of the \texttt{clist}. 
Character Constants in the DATA Statement

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

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

If the constant in \textit{clist} is of integer type and the item of \textit{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 intrinsic function.

If the constant of \textit{clist} is a character constant or a Hollerith constant, and the item of \textit{nlist} is of type \texttt{INTEGER}, then the number of characters that can be assigned is 2 or 4 for \texttt{INTEGER*2} and \texttt{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 \textit{nlist} can specify an implied DO list for initialization of array elements.

The form of an implied DO list is:
\[(dlist, iv=m1, m2 [, m3])\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{dlist}</td>
<td>List of array element names and implied DO lists</td>
</tr>
<tr>
<td>\textit{iv}</td>
<td>Integer variable, called the implied DO variable</td>
</tr>
<tr>
<td>\textit{m1}</td>
<td>Integer constant expression specifying the initial value of \textit{iv}</td>
</tr>
<tr>
<td>\textit{m2}</td>
<td>Integer constant expression specifying the limit value of \textit{iv}</td>
</tr>
<tr>
<td>\textit{m3}</td>
<td>Integer constant expression specifying the increment value of \textit{iv}; If \textit{m3} is omitted, then a default value of 1 is assumed.</td>
</tr>
</tbody>
</table>

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

Implied DO lists may also appear within the variables lists on I/O statements \texttt{PRINT}, \texttt{READ}, and \texttt{WRITE}.  

Chapter 4  Statements  119
Variables

Variables can also be initialized in type statements. This is an extension of the FORTRAN 77 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 an integer and a character:

```
CHARACTER CR*1
INTEGER I*2, N*4
DATA I /'00'/, N/4Hs12t/., CR/13/
...
```
**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 77 Standard, and are provided for compatibility with older versions of FORTRAN.

**ENCODE** \((size, f, buf [, IOSTAT=ios] [, ERR=s]) [iolist]\)

**DECODE** \((size, f, buf [, IOSTAT=ios] [, ERR=s]) [iolist]\)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Number of characters to be translated, an integer expression</td>
</tr>
<tr>
<td>f</td>
<td>Format identifier, either the label of a FORMAT statement, or a character expression specifying the format string, or an asterisk.</td>
</tr>
<tr>
<td>buf</td>
<td>Variable, array, or array element</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier, (ios) must be an integer variable or an integer array element.</td>
</tr>
<tr>
<td>s</td>
<td>The error specifier (statement label) (s) must be the label of executable statement in the same program unit in which the <strong>ENCODE</strong> and <strong>DECODE</strong> statement occurs.</td>
</tr>
<tr>
<td>iolist</td>
<td>List of input/output items.</td>
</tr>
</tbody>
</table>

**Description**

The entities in the I/O list can be: variables, substrings, arrays, array elements, 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

A program using DECODE/ENCODE:

```fortran
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'.

122 FORTRAN 77 Language Reference • May 2000
DIMENSION

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

Optionally, the DIMENSION statement initializes items with values.

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

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Name of an array</td>
</tr>
<tr>
<td>d</td>
<td>Specifies the dimensions of the array. It is a list of 1 to 7 declarators separated by commas.</td>
</tr>
</tbody>
</table>

### Description

This section contains descriptions for the dimension declarator and the arrays.

#### 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:

```fortran
DIMENSION M(4,4), V(1000)
...
END
```

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

Example 2: An adjustable array in a subroutine:

```fortran
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 $N \times N$.

Example 3: Lower and upper bounds:

```fortran
DIMENSION HELIO (-3:3, 4, 3:9)
...
END
```
In the above example, 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:

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

Example 5: Noninteger bounds:

```fortran
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 non-integer bounds:

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

Example 7: Assumed-size arrays:

```fortran
SUBROUTINE RUN(A,B,N)
    DIMENSION A(*), B(N,*)
    ...
```
**DO**

The DO statement repeatedly executes a set of statements.

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

or

```
DO  loop-control *
```

where \( s \) is a statement number. The form of loop-control is

\[
variable = e1, e2 [, e3]
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( variable )</td>
<td>Variable of type integer, real, or double precision.</td>
</tr>
<tr>
<td>( e1, e2, e3 )</td>
<td>Expressions of type integer, real or double precision, specifying initial, limit, and increment values respectively.</td>
</tr>
</tbody>
</table>

**Description**

The DO statement contains the following constructs.

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

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

**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 loop 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:
   \[
   \text{MAX} \left( \text{INT} \left( \frac{(e2 - e1 + e3)}{}, e3 \right), e3 0 \right)
   \]
   The iteration count is zero if either of the following is true:
   - `e1 > e2` and `e3 > 0`.
   - `e1 < e2` and `e3 < 0`.

   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.

Control 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.

If there is a jump into the range of a `DO` loop from outside its range, a warning is issued, but execution continues anyway.
When the jump is from outside to the terminal statement that is \texttt{CONTINUE}, and this statement is the terminal statement of several nested \texttt{DO} loops, then the most inner \texttt{DO} loop is always executed.

**Examples**

Example 1: Nested \texttt{DO} loops:

```fortran
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
```

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

Example 2: The program \texttt{DoNest2.f} (DO variable always defined):

```fortran
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
```
DO WHILE

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

\[
\text{DO [ \{ s, \} ] WHILE (e)}
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>Label of an executable statement</td>
</tr>
<tr>
<td>( e )</td>
<td>Logical expression</td>
</tr>
</tbody>
</table>

**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 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.

Restrictions

Jumping into the range of a DO WHILE loop from outside its range can produce unpredictable results.

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
```
DOUBLE COMPLEX

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

DOUBLE COMPLEX \( v / c / \) \( , v / c / \) ...

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v )</td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>( c )</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

Description

The declaration can be: DOUBLE COMPLEX or COMPLEX*16.

DOUBLE COMPLEX *

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

If you do not specify the size, a default size is used.

The default size, for a declaration such as DOUBLE COMPLEX \( Z \), can be altered by compiling with any of the options -dbl, -r8, or -xtypemap. See the discussion in Chapter 2 for details.

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.
Comments

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 are: 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
```
DOUBLE PRECISION

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

DOUBLE PRECISION v[/c/] [, v[/c/] ...]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>c</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

Description

The declaration can be: DOUBLE PRECISION or REAL*8.

DOUBLE PRECISION

For a declaration such as DOUBLE PRECISION X, the variable X is a REAL*8 element in memory, interpreted as one double-width real number.

If you do not specify the size, a default size is used. The default size, for a declaration such as DOUBLE PRECISION X, can be altered by compiling with any of the options -dbl, -r8, or -xtypemap. See the discussion in Chapter 2 for details.

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.

Example

```
DOUBLE PRECISION R, S(3,6)
REAL*8 T(-1:0,5)
```
**ELSE**

The **ELSE** statement indicates the beginning of an **ELSE** block.

```plaintext
IF (e) THEN
  ...
ELSE
  ...
END IF
```

where \( e \) is a logical expression.

**Description**

Execution of an **ELSE** statement has no effect on the program.

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 “**IF (Block)**” on page 170 for more information.

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
...
```
**ELSE IF**

The *ELSE IF* provides a multiple alternative decision structure.

```
ELSE IF (e2) THEN
IF (e1) THEN
END IF...
```

where *e1* and *e2* are 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.

Execution of the *ELSE IF* (*e*) proceeds as follows, depending on the value of the logical expression, *e*:

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.

**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.
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
```
ENCODE/DECODE

The ENCODE * statement writes data from a list to memory.

ENCODE(size, f, buf [, IOSTAT=ios] [, ERR=s]) [iolist]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Number of characters to be translated</td>
</tr>
<tr>
<td>f</td>
<td>Format identifier</td>
</tr>
<tr>
<td>buf</td>
<td>Variable, array, or array element</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier</td>
</tr>
<tr>
<td>s</td>
<td>Error specifier (statement label)</td>
</tr>
<tr>
<td>iolist</td>
<td>List of I/O items, each a character variable, array, or array element</td>
</tr>
</tbody>
</table>

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 77 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 “DECODE/ENCODE” on page 121 for more information and a full example.
END

The END statement indicates the end of a program unit with the following 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.

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 77 Standard, the END statement cannot be continued, but f77 allows this practice.*

No other statement, such as an END IF statement, can have an initial line that appears to be an END statement.

Example

Example: END:

```fortran
PROGRAM MAIN
WRITE( *, * ) 'Very little'
END
```
END DO

The END DO statement terminates a DO loop and requires the following 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 loop with a statement number:

```
  DO 10 N = 1, 100
    ...
  10    END DO
```

Example 2: A DO loop without statement number:

```
  DO N = 1, 100
    ...
  END DO
```
**END FILE**

The **END FILE** statement writes an end-of-file record as the next record of the file connected to the specified unit.

```
END FILE u
```

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>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.</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier, an integer variable or an integer array element.</td>
</tr>
<tr>
<td>s</td>
<td>Error specifier, s must be the label of an executable statement in the same program in which the <strong>END FILE</strong> statement occurs. The program control is transferred to the label in the event of an error during the execution of the <strong>END FILE</strong> statement.</td>
</tr>
</tbody>
</table>

**Description**

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.

**Restrictions**

u must be connected for *sequential* access. Execution of an **END FILE** statement on a direct-access file is not defined in the FORTRAN 77 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
```
END IF

The END IF statement ends the block IF that the IF began and requires the following 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
The END MAP statement terminates the MAP declaration and requires the following syntax:

```
END MAP
```

**Description**

See “UNION and MAP” on page 253 for more information.

**Restrictions**

The MAP statement must be within a UNION statement.

**Example**

```
... 
MAP
  CHARACTER *16 MAJOR
END MAP
END MAP
... 
```
END STRUCTURE

The END STRUCTURE statement terminates the STRUCTURE statement and requires the following syntax:

   END STRUCTURE

Description

See “STRUCTURE” on page 244 for more information.

Example

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

The `END UNION` statement terminates the `UNION` statement and requires the following syntax:

```
END UNION
```

**Description**

See “UNION and MAP” on page 253 for more information.

**Example**

```
UNION
MAP
   CHARACTER*16
END MAP
MAP
   INTEGER*2    CREDITS
   CHARACTER *8 GRAD_DATE
END MAP
END UNION
```
ENTRY

The ENTRY statement defines an alternate entry point within a subprogram.

ENTRY en [(fa[, , fa]...)]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>en</td>
<td>Symbolic name of an entry point in a function or subroutine subprogram</td>
</tr>
<tr>
<td>fa</td>
<td>Formal argument—it can be a variable name, array name, formal procedure name, or an asterisk specifying an alternate return label</td>
</tr>
</tbody>
</table>

Description

Note these nuances for the ENTRY statement:

Procedure References 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.

Argument 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 FIN( A, B, C )
INTEGER A, B
CHARACTER C*4
...
RETURN
ENTRY HLEP( A, B, C )
...
RETURN
ENTRY MOOZ
...
RETURN
END
```

In the above example, the subroutine FIN has two alternate entries: the entry HLEP
has an argument list; the entry MOOZ has no argument list.
Example 2: In the calling routine, you can call the above subroutine and entries as follows:

```fortran
INTEGER A, B
CHARACTER C*4
...
CALL FIN( A, B, C )
...
CALL MOOZ
...
CALL HLEP( A, B, C )
...
```

In the above example, the order of the call statements need not match the order of the entry statements.

Example 3: Multiple entry points in a function:

```fortran
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
```
EQUIVALENCE

The EQUIVALENCE statement specifies that two or more variables or arrays in a program unit share the same memory.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlist</td>
<td>List of variable names, array element names, array names, and character substring names separated by commas</td>
</tr>
</tbody>
</table>

Description

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.

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 less than or equal to the number of dimensions specified in the array declarator for the array name.

Restrictions

In nlist, 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.

EQUIVALENCE can associate automatic variables only with other automatic variables or undefined storage classes. These classes must be ones which are not in any of the COMMON, STATIC, SAVE, DATA, or dummy arguments.

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 statement 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 statement is not allowed:

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A(1), D(1)), (A(2), D(2))
```

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. The first seven character positions are arranged in memory as follows:

```
  01  02  03  04  05  06  07
  A  A(1) A(2) A(3) A(4)
  B  B(1) B(2) B(3) B(4)
```
EXTERNAL

The EXTERNAL statement specifies procedures or dummy procedures as external, and allows their symbolic names to be used as actual arguments.

EXTERNAL proc [, proc] ...

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc</td>
<td>Name of external procedure, dummy procedure, or block data routine.</td>
</tr>
</tbody>
</table>

Description

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:

```fortran
EXTERNAL TAN
T = TAN( 45.0 )
...
END
FUNCTION TAN( X )
...
RETURN
END
```

Example 2: Pass a user-defined function name as an argument:

```fortran
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
```
FORMAT

The FORMAT statement specifies the layout of the input or output records.

\textit{label FORMAT (f)}

\begin{table}[h]
\centering
\begin{tabular}{ll}
\textbf{Parameter} & \textbf{Description} \\
\hline
label & Statement number \\
f & Format specification list \\
\hline
\end{tabular}
\end{table}

The items in \( f \) have the form:

\[ [r] \ desc \]

\begin{itemize}
\item \( r \) is an optional repeat factor
\item \( desc \) is an edit descriptor. If \( r \) is present, then \( desc \) must be a repeatable edit descriptor.
\end{itemize}

The repeatable edit descriptors are:

\begin{table}[h]
\centering
\begin{tabular}{c c c c c}
\hline
I & F & E & D & G \\
Iw & Fw & Ew & Dw & Gw \\
Iwd & Fwd & Ewd & Dwed & Gwd \\
O & A & Ewde & Dwde & Gwde \\
Ow & Aw & Ewde & Dwde & Gwde \\
Owd & L & Ewde & Dwde & Gwde \\
Z & Lw & Ewde & Dwde & Gwde \\
zw & L & Ewde & Dwde & Gwde \\
zwd & \\
\hline
\end{tabular}
\end{table}

Here is a 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

The nonrepeatable edit descriptors are:

- 'a1a2 ... an' single quote-delimited character string
- "a1a2 ... an" double quote-delimited character string
- \texttt{nH}a1a2 ... an Hollerith string
- $
The \texttt{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 \texttt{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:

\( (*) , (\$), (\text{'}) , (\text{/}), (:) , B, BN, BZ, H, F, R, Q, S, SU, SP, SS, T, TL, TR, X \)

These descriptors indicate the manner of editing and are \textit{not} repeatable:

- Each \( ai \) is any ASCII character.
- \( n \) is a nonzero, unsigned integer constant.
- \( k \) is an optionally signed integer constant.

See “Formatted I/O” on page 273 for full details of these edit descriptors.
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, 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.

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 > ... )
```

The \( n \) in an \( nH \ldots \) edit descriptor cannot be a variable format expression.

Restrictions

The FORMAT statement label cannot be used in a GO TO, IF-arithmetic, DO, or alternate return.

Warnings

For explicit formats, invalid format strings cause warnings or error messages at compile time.

For formats in variables, invalid format strings cause warnings or error messages at runtime.

For variable format expressions, of the form \(<e>\), invalid format strings cause warnings or error messages at compile time or runtime.

See “Runtime Formats” on page 305 for details.
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 )
```
**FUNCTION (External)**

The FUNCTION statement identifies a program unit as a function subprogram.

\[
\text{[type]} \ \text{FUNCTION} \ fun \ ([ar[, \ ar]...])
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>BYTE *</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>CHARACTER*n (where n must be greater than zero)</td>
</tr>
<tr>
<td></td>
<td>CHARACTER*(*)</td>
</tr>
<tr>
<td></td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td>COMPLEX*8 *</td>
</tr>
<tr>
<td></td>
<td>COMPLEX*16 *</td>
</tr>
<tr>
<td></td>
<td>COMPLEX*32 *</td>
</tr>
<tr>
<td></td>
<td>DOUBLE COMPLEX *</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>INTEGER*2 *</td>
</tr>
<tr>
<td></td>
<td>INTEGER*4 *</td>
</tr>
<tr>
<td></td>
<td>INTEGER*8 *</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*1 *</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*2 *</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*4 *</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*8 *</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td>REAL*4 *</td>
</tr>
<tr>
<td></td>
<td>REAL*8 *</td>
</tr>
<tr>
<td></td>
<td>REAL*16 *</td>
</tr>
<tr>
<td>fun</td>
<td>Symbolic name assigned to function</td>
</tr>
<tr>
<td>ar</td>
<td>Formal argument name</td>
</tr>
</tbody>
</table>

(COMPLEX*32 and REAL*16 are SPARC only.)

An alternate nonstandard syntax for length specifier is as follows: *

\[
\text{[type]} \ \text{FUNCTION} \ fun \ [*m] ([ar[, \ ar]...])
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Unsigned, nonzero integer constant specifying length of the data type.</td>
</tr>
<tr>
<td>ar</td>
<td>Formal argument name</td>
</tr>
</tbody>
</table>
Description

Note the type, value, and formal arguments for a FUNCTION statement.

Type of Function

The function statement involves type, name, and formal arguments.

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.

Note – Compiling with any of the options -dbl, -r8, -i2, or -xtypemap can alter the default data size assumed in the call to or definition of functions unless the data type size is explicitly declared. See Chapter 2 and the Fortran User Guide for details on these options.

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.
£77 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:

```fortran
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:

```fortran
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: *

```fortran
INTEGER FUNCTION FCN*2 ( A, B, C )
```

The above nonstandard form is treated as:

```fortran
INTEGER*2 FUNCTION FCN ( A, B, C )
```
GO TO (Assigned)

The assigned GO TO statement branches to a statement label identified by the assigned label value of a variable.

```
GO TO i {{{,} (s[,s]...)}
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Integer variable name</td>
</tr>
<tr>
<td>s</td>
<td>Statement label of an executable statement</td>
</tr>
</tbody>
</table>

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.

\( i \) must be INTEGER*4 or INTEGER*8, not INTEGER*2.

\( s \) 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 control jumps to must be executable, not DATA, ENTRY, FORMAT, or INCLUDE.

Control 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
```
**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.

```
GO TO (s[,s]...[,e]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Statement label of an <em>executable</em> statement</td>
</tr>
<tr>
<td>e</td>
<td>Expression of type integer or real</td>
</tr>
</tbody>
</table>

**Description**

Execution proceeds as follows:

1. *e* is evaluated first. It is converted to integer, if required.
2. If $1 \leq e \leq n$, 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 control jumps to must be *executable*, not DATA, ENTRY, FORMAT, or INCLUDE.
- Control 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 $N$ equals one, then go to 10.
- If $N$ equals two, then go to 20.
- If $N$ equals three, then go to 30.
- If $N$ equals four, then go to 40.
- If $N$ is less than one or $N$ is greater than four, then fall through to 10.
GO TO (Unconditional)

The *unconditional* GO TO statement transfers control to a specified statement.

```
GO TO s
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Statement label of an <em>executable</em> statement</td>
</tr>
</tbody>
</table>

**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 control jumps to must be executable, not a DATA, ENTRY, FORMAT, or INCLUDE statement.

Control 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
```
**IF (Arithmetic)**

The *arithmetic* IF statement branches to one of three specified statements, depending on the value of an arithmetic expression.

\[
\text{IF} \ (e) \ s1, \ s2, \ s3
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e)</td>
<td>Arithmetic expression: integer, real, double precision, or quadruple precision</td>
</tr>
<tr>
<td>(s1, s2, s3)</td>
<td>Labels of <em>executable</em> statements</td>
</tr>
</tbody>
</table>

**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.

The restrictions are:
- 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 control jumps to must be *executable*, not DATA, ENTRY, FORMAT, or INCLUDE.
- Control cannot jump into a DO, IF, ELSE IF, or ELSE block from outside the block.

**Example**

\[
\begin{align*}
N &= 0 \\
\text{IF} \ (N) &\ 10, \ 20, \ 30
\end{align*}
\]

Since the value of \(N\) is zero, control is transferred to statement label 20.
**IF (Block)**

The *block IF* statement executes one of two or more sequences of statements, depending on the value of a logical expression.

```
IF (e) THEN
...
END IF
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>A logical expression</td>
</tr>
</tbody>
</table>

**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; \( n2 \) 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 that has 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

Control 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:

```fortran
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

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
```
IF (Logical)

The logical IF statement executes one single statement, or does not execute it, depending on the value of a logical expression.

IF (e) st

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Logical expression</td>
</tr>
<tr>
<td>st</td>
<td>Executable statement</td>
</tr>
</tbody>
</table>

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

```fortran
IF ( VALUE .LE. ATAD ) CALL PUNT  ! Note that there is no THEN.
IF ( TALLY .GE. 1000 ) RETURN
```
**IMPLICIT**

The **IMPLICIT** statement confirms or changes the default type of names.

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

**IMPLICIT NONE**

**IMPLICIT UNDEFINED(A-Z)  u**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>BYTE u</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>CHARACTER*16 (where n must be greater than 0)</td>
</tr>
<tr>
<td></td>
<td>CHARACTER* (*)</td>
</tr>
<tr>
<td></td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td>COMPLEX*8  u</td>
</tr>
<tr>
<td></td>
<td>COMPLEX*16 u</td>
</tr>
<tr>
<td></td>
<td>COMPLEX*32 u (SPARC only)</td>
</tr>
<tr>
<td></td>
<td>DOUBLE COMPLEX  u</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>INTEGER*2  u</td>
</tr>
<tr>
<td></td>
<td>INTEGER*4  u</td>
</tr>
<tr>
<td></td>
<td>INTEGER*8  u</td>
</tr>
<tr>
<td></td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*1  u</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*2  u</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*4  u</td>
</tr>
<tr>
<td></td>
<td>LOGICAL*8  u</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td>REAL*4     u</td>
</tr>
<tr>
<td></td>
<td>REAL*8     u</td>
</tr>
<tr>
<td></td>
<td>REAL*16    u (SPARC only)</td>
</tr>
<tr>
<td></td>
<td>AUTOMATIC u</td>
</tr>
<tr>
<td></td>
<td>STATIC u</td>
</tr>
</tbody>
</table>

- 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**

The different uses for implicit typing and no implicit typing are described here.
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.

---

**Note** – Compiling with any of the options `-dbl`, `-i2`, `-r8`, or `-xtypemap` can alter the assumed size of names typed with an `IMPLICIT` statement that does not specify a size: `IMPLICIT REAL (A-Z)`. See Chapter 2 and the *Fortran User’s Guide* for details.

---

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 77 Standard restricts this usage to *only once*. For $£77$, if a letter is used twice, each usage is declared in order. See 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.
**INCLUDE**

The INCLUDE * statement inserts a file into the source program.

```
INCLUDE 'file'
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Name of the file to be inserted</td>
</tr>
</tbody>
</table>

INCLUDE "file"

**Description**

The contents of the named file replace the INCLUDE statement.

**Search Path**

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 that contains the source file with the INCLUDE statement
2. The directories that are named in the `-Iloc` options
3. The current directory in which the `f77` command was issued
4. The directories in the default list. For a standard install, the default list is:
   ```
   /opt/SUNWspro/<release>/include/f77  /usr/include
   ```
   For a non-standard install to a directory `/mydir/`, the default list is:
   ```
   /mydir/SUNWspro/<release>/include/f77  /usr/include
   ```
   The `<release>` 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 *Fortran User’s Guide*. Files included by the preprocessor `#include` directive can contain `#defines` and the like; files included with the compiler `include` statement must contain only FORTRAN statements.

VMS Logical File Names in the `INCLUDE` Statement

`$77` interprets VMS logical file names on the `include` statement if:

- The `-x1[d]` or `-vax=spec` compiler options are set.
- The environment variable `LOGICALNAMEMAPPING` is there to define the mapping between the logical names and the UNIX path name.

`$77` uses the following rules for the interpretation:

- The environment variable should be set to a string with the syntax:

  "iname1=path1; iname2=path2; ...

  where each `iname` 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 `$77` converts file names of the `iname1/file` form to the `path1/file` form.

- For logical names, uppercase and lowercase are 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.

For a standard install, f77 searches these directories:
- `/usr/ftn/projA/ver1`
- `/usr/ftn/ver1`
- `/opt/SUNWspro/<release>/include/f77/ver1`
- `/usr/include`

For a non-standard install to a directory `/mydir`, replace `/opt` with `/mydir`. The
<release> path changes with each compiler release.
INQUIRE

The INQUIRE statement returns information about a unit or file.

INQUIRE([UNIT =] u, slist)

INQUIRE(FILE = fn, slist)

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn</td>
<td>Name of the file being queried</td>
</tr>
<tr>
<td>u</td>
<td>Number of the file being queried</td>
</tr>
</tbody>
</table>
| slist     | The specifiers list 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 |
|           | • IOSTAT = 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).

Inquire either by unit or by file, but not by both in the same 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 following table summarizes the `INQUIRE` specifiers:

**TABLE 4-1  INQUIRE Statement Specifiers**

<table>
<thead>
<tr>
<th>SPECIFIER</th>
<th>Value of Variable</th>
<th>Data Type of Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>'DIRECT'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'SEQUENTIAL'</td>
<td></td>
</tr>
<tr>
<td>BLANK</td>
<td>'NULL'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'ZERO'</td>
<td></td>
</tr>
<tr>
<td>DIRECT *</td>
<td>'YES'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'NO'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'UNKNOWN'</td>
<td></td>
</tr>
<tr>
<td>ERR</td>
<td>Statement number</td>
<td>INTEGER</td>
</tr>
<tr>
<td>EXIST</td>
<td>.TRUE.</td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td></td>
</tr>
<tr>
<td>FORM</td>
<td>'FORMATTED'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'UNFORMATTED'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'BINARY' ✠</td>
<td></td>
</tr>
<tr>
<td>FORMATTED *</td>
<td>'YES'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'NO'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'UNKNOWN'</td>
<td></td>
</tr>
<tr>
<td>IOSTAT</td>
<td>Error number</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NAME †</td>
<td>Name of the file</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>NAMED †</td>
<td>.TRUE.</td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td></td>
</tr>
<tr>
<td>NEXTREC</td>
<td>Next record number</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NUMBER *</td>
<td>Unit number</td>
<td>INTEGER</td>
</tr>
<tr>
<td>OPENED</td>
<td>.TRUE.</td>
<td>LOGICAL</td>
</tr>
<tr>
<td></td>
<td>.FALSE.</td>
<td></td>
</tr>
<tr>
<td>RECL</td>
<td>Record length</td>
<td>INTEGER</td>
</tr>
<tr>
<td>SEQUENTIAL *</td>
<td>'YES'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'NO'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'UNKNOWN'</td>
<td></td>
</tr>
<tr>
<td>UNFORMATTED *</td>
<td>'YES'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'NO'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'UNKNOWN'</td>
<td></td>
</tr>
</tbody>
</table>

* indicates non-standard for inquire-by-unit, but accepted by f77.
† indicates non-standard for inquire-by-file, but accepted by f77.
Also:

- If a file is scratch, then NAMED and NUMBER are not returned.
- If there is no file with the specified name, then these variables are not returned: DIRECT, FORMATTED, NAME, NAMED, SEQUENTIAL, and UNFORMATTED.
- If OPENED=.FALSE., then these variables are not returned: ACCESS, BLANK, FORM, NEXTREC, and RECL.
- If no file is connected to the specified unit, then these variables are not returned: ACCESS, BLANK, DIRECT, FORM, FORMATTED, NAME, NAMED, NEXTREC, NUMBER, RECL, SEQUENTIAL, and UNFORMATTED.
- If ACCESS='SEQUENTIAL', then these variables are not returned: RECL and NEXTREC.
- If FORM='UNFORMATTED', then BLANK is not returned.

INQUIRE Specifier Keywords

The following provides a detailed list of the INQUIRE specifier keywords:

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.

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.

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.

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. If the file is a link, INQUIRE always returns .TRUE., even if the linked file does not exist.

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.

FORM=fm
- fm is a character variable which is assigned the value 'FORMATTED' if the file is connected for formatted I/O, 'UNFORMATTED' if the file is connected for unformatted I/O, 'BINARY' if the file was opened for unstructured binary 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 cannot tell.

IOSTAT=ios
- ios is as in the OPEN statement.

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 77 Standard leaves it undefined.

NAMED=nmd
- nmd is a logical variable that is assigned .TRUE. if the file has a name, .FALSE. otherwise.
NEXTREC = nr
- nr is an integer variable that is assigned one plus the number of the last record read from a file connected for direct access. If the file is not connected, -1 is returned in nr.

NUMBER = num
- num is an integer variable that is set to the number of the unit connected to the file, if any. If no file is connected, num is set to -1.

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.

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 adjust the rcl returned by INQUIRE. The OPEN statement does such an adjustment if the -xl[d] option is set. See “Details of Features That Require -xl[d]” on page 376 for an explanation of -xl[d]. If no file is connected, rcl is set to -1.

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.

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 cannot tell.

UNIT = u
- u is an integer expression or * with the value of the unit. Exactly one of FILE or UNIT must be used.
Examples

Example 1: Inquire by unit:

```fortran
LOGICAL OK
INQUIRE( UNIT=3, OPENED=OK )
IF ( OK ) CALL GETSTD ( 3, STDs )
```

Example 2: Inquire by file:

```fortran
LOGICAL THERE
INQUIRE( FILE='.profile', EXIST=THERE )
IF ( THERE ) CALL GETPROFILE( FC, PROFILE )
```

Example 3: More than one answer, omitting the UNIT=:

```fortran
CHARACTER FN*32
LOGICAL HASNAME, OK
INQUIRE ( 3, OPENED=OK, NAMED=HASNAME, NAME=FN )
IF ( OK .AND. HASNAME ) PRINT *, 'Filename="', FN, '"
```
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.

```
INTEGER [*len [, ] v [*len [/c/ ] ] [, v [*len [/c/ ] ] ] ... 
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v</code></td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td><code>len</code></td>
<td>Either 2, 4, or 8, the length in bytes of the symbolic constant, variable, array element, or function.</td>
</tr>
<tr>
<td><code>c</code></td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

### Description

The declarations can be: `INTEGER`, `INTEGER*2`, `INTEGER*4`, `INTEGER*8`. 
INTEGER

For a declaration such as INTEGER \( H \), the variable \( H \) is usually one INTEGER*4 element in memory, interpreted as a single integer number. Specifying the size is nonstandard.*

If you do not specify the size, a default size is used. The default size, for a declaration such as INTEGER \( H \), can be altered by compiling with any of the options -dbl, -i2, -r8, or -xtypemap. See the discussion in Chapter 2 for details.

INTEGER*2  *

For a declaration such as INTEGER*2 \( H \), the variable \( H \) is always an INTEGER*2 element in memory, interpreted as a single integer number.

INTEGER*4  *

For a declaration such as INTEGER*4 \( H \), the variable \( H \) is always an INTEGER*4 element in memory, interpreted as a single integer number.

INTEGER*8  *

For a declaration such as INTEGER*8 \( H \), the variable \( H \) is always an INTEGER*8 element in memory, interpreted as a single integer number.

Restrictions

Do not use INTEGER*8 variables or 8-byte constants or expressions when indexing arrays, otherwise, only 4 low-order bytes are taken into account. This action can cause unpredictable results in your program if the index value exceeds the range for 4-byte integers.
Examples

Example 1: Each of these integer declarations are equivalent:

```
      INTEGER U, V(9)
      INTEGER*4 U, V(9)
      INTEGER U*4, V(9)*4
```

Example 2: Initialize:

```
      INTEGER U / 1 /, V / 4 /, W*2 / 1 /, X*2 / 4 /
```
INTRINSIC

The INTRINSIC statement lists intrinsic functions that can be passed as actual arguments.

INTRINSIC fun [, fun] ...

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun</td>
<td>Function name</td>
</tr>
</tbody>
</table>

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: 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 77 Standard, a symbolic name can appear only once in an INTRINSIC statement. *
Because they are inline or generic, the following intrinsics cannot be passed as actual arguments:

**TABLE 4-2  Intrinsics That Cannot Be Passed As Actual Arguments**

<table>
<thead>
<tr>
<th>LOC</th>
<th>INT</th>
<th>SNGL</th>
<th>AIMAX0</th>
<th>AMIN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>IINT</td>
<td>SNGLQ</td>
<td>AJMAX0</td>
<td>DMIN1</td>
</tr>
<tr>
<td>IAND</td>
<td>JINT</td>
<td>REAL</td>
<td>IMA0</td>
<td>IMIN1</td>
</tr>
<tr>
<td>IIAND</td>
<td>IQINT</td>
<td>DREAL</td>
<td>JMA0</td>
<td>JMIN1</td>
</tr>
<tr>
<td>JIAND</td>
<td>IIQINT</td>
<td>DBLE</td>
<td>MAX1</td>
<td>QMIN1</td>
</tr>
<tr>
<td>OR</td>
<td>JIQINT</td>
<td>DBLEQ</td>
<td>AMAX1</td>
<td>IMAG</td>
</tr>
<tr>
<td>IOR</td>
<td>IFIX</td>
<td>QEXT</td>
<td>DMAX1</td>
<td>LOG</td>
</tr>
<tr>
<td>IIOR</td>
<td>IIFIX</td>
<td>QEXTD</td>
<td>IMA1</td>
<td>LOG10</td>
</tr>
<tr>
<td>Ieor</td>
<td>JIFIX</td>
<td>QFLOAT</td>
<td>JMA1</td>
<td>QREAL</td>
</tr>
<tr>
<td>IIeor</td>
<td>IIDINT</td>
<td>CMPLX</td>
<td>QMA1</td>
<td>QCMLX</td>
</tr>
<tr>
<td>JIOR</td>
<td>IIDINT</td>
<td>CMPLX</td>
<td>MIN</td>
<td>SIZEOF</td>
</tr>
<tr>
<td>JIEOR</td>
<td>JIDINT</td>
<td>ICHAR</td>
<td>MIN0</td>
<td>EPBASE</td>
</tr>
<tr>
<td>NOT</td>
<td>FLOAT</td>
<td>ICHAR</td>
<td>AMIN0</td>
<td>EPEMAX</td>
</tr>
<tr>
<td>INOT</td>
<td>FLOATI</td>
<td>ACHAR</td>
<td>AIMIN0</td>
<td>EPEMIN</td>
</tr>
<tr>
<td>JNOT</td>
<td>FLOATJ</td>
<td>CHAR</td>
<td>AJMIN0</td>
<td>EPHUGE</td>
</tr>
<tr>
<td>XOR</td>
<td>DFLOAT</td>
<td>MAX</td>
<td>IMIN0</td>
<td>EPMRS</td>
</tr>
<tr>
<td>LSHIFT</td>
<td>DFLOATI</td>
<td>MAX0</td>
<td>JMIN0</td>
<td>EPPREC</td>
</tr>
<tr>
<td>RSHIFT</td>
<td>DFLOATJ</td>
<td>AMAX0</td>
<td>MIN1</td>
<td>EPTINY</td>
</tr>
<tr>
<td>LRSHIFT</td>
<td>I2EXT</td>
<td>JZEXT</td>
<td>ZEXT</td>
<td></td>
</tr>
</tbody>
</table>
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.

LOGICAL [*len [,]] v[*len [/c/]] [, v[*len [/c/]] ...]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Name of a symbolic constant, variable, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>len</td>
<td>Either 1, 2, 4, or 8, the length in bytes of the symbolic constant, variable, array element, or function. 8 is allowed only if -dbl is on.</td>
</tr>
<tr>
<td>c</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

Description

The declarations can be: LOGICAL, LOGICAL*1, LOGICAL*2, LOGICAL*4, LOGICAL*8.
LOGICAL

For a declaration such as LOGICAL H, the variable H is usually one INTEGER*4 element in memory, interpreted as a single logical value. Specifying the size is nonstandard. *

If you do not specify the size, a default size is used. The default size, for a declaration such as LOGICAL Z, can be altered by compiling with any of the options -dbl, -i2, -r8, or -xtypemap. See the discussion in Chapter 2 for details.

LOGICAL*1 *

For a declaration such as LOGICAL*1 H, the variable H is always a BYTE element in memory, interpreted as a single logical value.

LOGICAL*2 *

For a declaration such as LOGICAL*2 H, the variable H is always an INTEGER*2 element in memory, interpreted as a single logical value.

LOGICAL*4 *

For a declaration such as LOGICAL*4 H, the variable H is always an INTEGER*4 element in memory, interpreted as a single logical value.

LOGICAL*8 *

For a declaration such as LOGICAL*8 H, the variable H is always an INTEGER*8 element in memory, interpreted as a single logical value.
Examples

Example 1: Each of these declarations are equivalent:

<table>
<thead>
<tr>
<th>LOGICAL U, V(9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGICAL*4 U, V(9)</td>
</tr>
<tr>
<td>LOGICAL U*4, V(9)*4</td>
</tr>
</tbody>
</table>

Example 2: Initialize:

| LOGICAL U / .false. / , V / 0 / , W*4 / .true. / , X*4 / 'z' / |
**MAP**

The MAP * declaration defines alternate groups of fields in a union.

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

**Description**

Each field declaration can be one of the following:

- Type declaration, which can include initial values
- Substructure—either another structure declaration, or a record that has been previously defined
- Union declaration—see “UNION and MAP” on page 253 for more information.

**Example**

Example: MAP:

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

The NAMELIST * statement defines a list of variables or array names, and associates it with a unique group name.

```
NAMELIST / grname / namelist [\[\] / grname / namelist ] ...
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grname</td>
<td>Symbolic name of the group</td>
</tr>
<tr>
<td>namelist</td>
<td>List of variables and arrays</td>
</tr>
</tbody>
</table>

Description

The NAMELIST statement contains a group name and other items.

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 identifies 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 constructs cannot appear in a NAMELIST statement:

- Constants (parameters)
- Array elements
- Records and record fields
- Character substrings
- Dummy assumed-size arrays

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.
OPEN

The OPEN statement can connect an existing external file to a unit, create a file and connect it to a unit, or change some specifiers of the connection.

OPEN ([UNIT=] u, slist)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT</td>
<td>Unit number</td>
</tr>
<tr>
<td>slist</td>
<td>The specifiers list slist can include one or more of the following</td>
</tr>
<tr>
<td></td>
<td>• FILE = fin or alternatively NAME = fin</td>
</tr>
<tr>
<td></td>
<td>• ACCESS = acc</td>
</tr>
<tr>
<td></td>
<td>• BLANK = blank</td>
</tr>
<tr>
<td></td>
<td>• ERR = s</td>
</tr>
<tr>
<td></td>
<td>• FORM = fm</td>
</tr>
<tr>
<td></td>
<td>• IOSTAT = ios</td>
</tr>
<tr>
<td></td>
<td>• RECL = rl or alternatively RECORDSIZE = rl</td>
</tr>
<tr>
<td></td>
<td>• STATUS = sta or alternatively TYPE = sta</td>
</tr>
<tr>
<td></td>
<td>• FILEOPT = fopt *</td>
</tr>
<tr>
<td></td>
<td>• READONLY *</td>
</tr>
<tr>
<td></td>
<td>• ACTION = act *</td>
</tr>
</tbody>
</table>

Description

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.

Note – For tape I/O, use the TOPEN() routines.
The following table summarizes the OPEN specifiers:

**TABLE 4-3 OPEN Statement Specifiers**

<table>
<thead>
<tr>
<th>SPECIFIER</th>
<th>Value of Variable</th>
<th>Data Type of Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>'APPEND'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'DIRECT'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'SEQUENTIAL'</td>
<td></td>
</tr>
<tr>
<td>ACTION</td>
<td>'READ'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'WRITE'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'READWRITE'</td>
<td></td>
</tr>
<tr>
<td>BLANK</td>
<td>'NULL'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'ZERO'</td>
<td></td>
</tr>
<tr>
<td>ERR</td>
<td>Statement number</td>
<td>INTEGER</td>
</tr>
<tr>
<td>FORM</td>
<td>'FORMATTED'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'UNFORMATTED'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'PRINT'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'BINARY' *</td>
<td></td>
</tr>
<tr>
<td>FILE</td>
<td>Filename</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>FILEOPT</td>
<td>'NOPAD'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'BUFFER=n'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'EOF'</td>
<td></td>
</tr>
<tr>
<td>IOSTAT</td>
<td>Error number</td>
<td>INTEGER</td>
</tr>
<tr>
<td>READONLY</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RECL</td>
<td>Record length</td>
<td>INTEGER</td>
</tr>
<tr>
<td>STATUS</td>
<td>'OLD'</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>'NEW'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'UNKNOWN'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'SCRATCH'</td>
<td></td>
</tr>
</tbody>
</table>

The keywords can be specified in any order.

**OPEN Specifier Keywords**

The following provides a detailed list of the OPEN specifier keywords:

\[
[\text{UNIT=}] \ u
\]

- \( u \) is an integer expression or an asterisk (*) that specifies the unit number. \( u \) is required. If \( u \) is first in the parameter list, then \text{UNIT=} can be omitted.
FILE=fin

- fin is a character expression naming the file to open. An OPEN statement need not specify a file name. If the file name is not specified, a default name is created.

- Reopening files: If you open a unit that is 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.

- Switching Files: If you open a unit that is already open, but you specify a different file name, it is as if you closed with the old file name before the open.

- Switching Units: If you open a file that is already open, but you specify a different unit, that is an error. This error is not detected by the ERR= option, however, and the program does not terminate abnormally.

- Scratch Files: 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, although no error message is issued. This is an extension and can be applied only to disk files. *

- 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 77 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 do not have to be randomly accessible, in the sense that tty, pipes, and tapes can be used. For tapes, we recommend the OPEN () routines because they are more reliable.

- If FORM is not, formatted transfer is assumed.
If `FORM='FORMATTED'`, each record is terminated with a newline (\n) character; that is, each record actually has one extra character.

If `FORM='PRINT'`, the file acts like a `FORM='FORMATTED'` file, except for interpretation of the column-1 characters on the output (blank = single space, 0 = double space, 1 = form feed, and + = no advance).

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=\textit{fm}**

- The `FORM=\textit{fm}` clause is optional. \textit{fm} is a character expression. Possible values are `'FORMATTED'`, `'UNFORMATTED'`, `'BINARY'` or `'PRINT'`. * The default is `'FORMATTED'`.
- This option interacts with `ACCESS`.
- `'PRINT'` makes it a `print` file.
- `'BINARY'` treats the file as a sequential unformatted file with no record marks. * Specifying `ACCESS='DIRECT'` or `RECL=n` with `FORM='BINARY'` generates an error. Each `WRITE` statement writes as many bytes in binary as there are in the data on the output list. A `READ` statement reads as many bytes from the input file as are required by the input list. Since no record marks are recognized, it is not possible to read "outside the record". Other than abnormal system errors, the only input error that can occur is reading the end-of-file. `BACKSPACE` on a `FORM='BINARY'` file is not allowed and generates a runtime error.

**RECL=\textit{rl}**

- The `RECL=\textit{rl}` clause is required if `ACCESS='DIRECT'` and ignored otherwise.
- \textit{rl} is an integer expression for the length in characters of each record of a file. \textit{rl} must be positive.
- If the record length is unknown, you can use `RECL=1`; see “Direct Access I/O” on page 272 for more information.
- If `-x1[d]` is not set, \textit{rl} is number of characters, and record length is \textit{rl}.
- If `-x1[d]` is set, \textit{rl} is number of words, and record length is \textit{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. For tapes, we recommend the `TOPEN()` routines because they are more reliable.
ERR=s

- The ERR=s clause is optional. 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. 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 or IOSTAT=ios.

BLANK=blnk

- The BLANK=blnk clause is optional, and is for formatted input only. The blnk is a character expression that indicates how blanks are treated. Possible values are 'ZERO' and 'NULL'.
- 'ZERO'—Blanks are treated as zeroes.
- 'NULL'—Blanks are ignored during numeric conversion. This is the default.

STATUS=sta

- The STATUS=sta clause is optional. sta is a character expression. Possible values are: 'OLD', 'NEW', 'UNKNOWN', or 'SCRATCH'.
- '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. This is the default.
- 'SCRATCH'—For a file opened with STATUS='SCRATCH', a temporary file with a name of the form tmp.FAAAxnnnnn 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 be deleted. To prevent deletion, CLOSE with STATUS='KEEP'.
- The FORTRAN 77 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 writable directory, then
the program puts scratch files there.

FILEOPT=fopt

The FILEOPT=fopt clause is optional. fopt is a character expression. Possible
values are 'NOPAD', 'BUFFER=n', and 'EOF'.

'NOPAD'—Do not 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'—Sets the buffer size to be used by the I/O unit to n bytes. This
suboption is intended for use with regular files, such as disk files. For good
performance, the buffer size should be a multiple of the page size. Using large
buffers generally improves the performance of sequential I/O. Performance of
direct I/O is usually best if the buffer size equals the record length. One exception
is that if the record length is one, the buffer size should be at least one page.

Normally the BUFFER suboption should not be used with tape files, where the
proper buffer size is determined by the tape hardware, the controllers, and the file
system. Some kinds of tape drives offer limited functionality. These limitations
can cause normal Fortran I/O to be unreliable. The tape I/O routines in the
Fortran library (see t open(3F)) are recommended alternatives for such drives.

'EOF'—Opens a file at end-of-file rather than at the beginning (useful for
appending data to file), for example, FILEOPT='EOF'. Unlike
ACCESS='APPEND', in this case, both READ and BACKSPACE are allowed.

READONLY

The file is opened read-only.

ACTION=act

This specifier denotes file permissions. Possible values are: READ, WRITE, and
READWRITE.

If act is READ, it specifies that the file is opened read-only.

If act is WRITE, it specifies that the file is opened write-only. You cannot execute a
BACKSPACE statement on a write-only file.

If act is READWRITE, it specifies that the file is opened with both read and write
permissions.
Examples

Here are six examples.

Example 1: Open a file and connect it to unit 8—either of the following forms of the `OPEN` statement opens the file, `projectA/data.test`, and connects it to FORTRAN unit 8:

```
OPEN(UNIT=8, FILE='projectA/data.test')
OPEN(8, FILE='projectA/data.test')
```

In the above example, these 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 `OPEN`.

Example 5: Allowing for variable-length records;

```
OPEN(1, ACCESS='DIRECT', recl=1)
```

See “Direct Access I/O” on page 272 for more information about variable-length records.
Example 6: Scratch file:

```
OPEN(1, STATUS='SCRATCH')
```

This statement 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 environment variable is set.
OPTIONS

The OPTIONS * statement overrides compiler command-line options.

OPTIONS /qualifier [/qualifier ...]

Description

The following table shows the OPTIONS statement qualifiers:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>/[NO]G_FLOATING</td>
<td>None (not implemented)</td>
</tr>
<tr>
<td>/[NO]I4</td>
<td>Enables/Disables the -i2 option</td>
</tr>
<tr>
<td>/[NO]F77</td>
<td>None (not implemented)</td>
</tr>
<tr>
<td>/CHECK=ALL</td>
<td>Enables the -c option</td>
</tr>
<tr>
<td>/CHECK=[NO]OVERFLOW</td>
<td>None (not implemented)</td>
</tr>
<tr>
<td>/CHECK=[NO]BOUNDS</td>
<td>Disables/Enables the -c option</td>
</tr>
<tr>
<td>/CHECK=[NO]UNDERFLOW</td>
<td>None (not implemented)</td>
</tr>
<tr>
<td>/CHECK=None</td>
<td>Disables the -c option</td>
</tr>
<tr>
<td>/NOCHECK</td>
<td>Disables the -c option</td>
</tr>
<tr>
<td>/[NO]EXTEND_SOURCE</td>
<td>Disables/enables the -e option</td>
</tr>
</tbody>
</table>

Restrictions

The OPTIONS statement must be the first statement in a program unit; 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 occupy 4 bytes rather than 2, with or without the \texttt{-i2} option on the command line. This rule does not change the size of integer constants, only variables.

```
OPTIONS /I4
PROGRAM FFT
...
END
```

By way of contrast, if you use \texttt{/NOI4}, then all integer variables declared with no explicit size occupy 2 bytes rather than 4, with or without the \texttt{-i2} option on the command line. However, integer constants occupy 2 bytes with \texttt{-i2}, and 4 bytes otherwise.
PARAMETER

The PARAMETER statement assigns a symbolic name to a constant.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>Symbolic name</td>
</tr>
<tr>
<td>e</td>
<td>Constant expression</td>
</tr>
</tbody>
</table>

An alternate syntax is allowed, if the -x1 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 “Constant Expressions” on page 83 for more information.

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.

* The Sun WorkShop FORTRAN 77 compiler extends the PARAMETER statement to accept any expression, including non-constant expressions. The statement will get a warning message to indicate that it is non-standard, and the value will be determined at runtime wherever the symbol is referenced. However, if the symbol defined in a PARAMETER statement with a non-constant expression appears in a statement where a constant is expected (such as a DATA statement) it will get an error.

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:

```fortran
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:

```plaintext
  CHARACTER HEADING*(*)
  PARAMETER (HEADING='I/O Error Number')
```

Example 3: The alternate syntax, if the –xl compilation flag is specified:

```plaintext
  PARAMETER FLAG1 = .TRUE.
```

The above statement is treated as:

```plaintext
  LOGICAL FLAG1
  PARAMETER (FLAG1 = .TRUE.)
```

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 specified.

Example: An ambiguous statement:

```plaintext
  PARAMETER S = .TRUE.
```

With –xl, the above statement is a PARAMETER statement about the variable S.

```plaintext
  PARAMETER S = .TRUE.
```

It is not an assignment statement about the variable PARAMETERS.

```plaintext
  PARAMETERS = .TRUE.
```
PAUSE

The PAUSE statement suspends execution, and waits for you to type: go.

PAUSE [str]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>String of not more than 5 digits or a character constant</td>
</tr>
</tbody>
</table>

Description

The PAUSE statement suspends program execution temporarily, and waits for acknowledgment. On acknowledgment, execution continues.

If the argument string is present, it is displayed on the screen (written to stdout), followed by the following message:

```
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. See this example:

demo% cat p.f
    PRINT *, "Start"
    PAUSE 1
    PRINT *, "Ok"
END
demo% f77 p.f -silent
demo% a.out
Start
PAUSE: 1
To resume execution, type: go
Any other input will terminate the program.
g o Execution resumed after PAUSE.
Ok
demo%
If stdin is not a tty I/O device, PAUSE displays a message like this:

```bash
PAUSE: To resume execution, type: kill -15 pid
```

where `pid` is the process ID.

Example: stdin not a tty I/O device:

```bash
demo% a.out < mydatafile
PAUSE: To resume execution, type: kill -15 20537
demo%
```

For the above example, type the following command line at a shell prompt in some other window. The window displaying the message cannot accept command input.

```bash
demo% kill -15 20537
```
POINTER

The POINTER * statement establishes pairs of variables and pointers.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1, v2</td>
<td>Pointer-based variables, also called pointees</td>
</tr>
<tr>
<td>p1, p2</td>
<td>Corresponding pointers</td>
</tr>
</tbody>
</table>

Description

Each pointer contains the address of its paired variable.

A pointer-based variable, or pointee, 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. (Variable names appearing on POINTER statements are considered VOLATILE by the compiler.)

The use of pointers is described in “Pointers” on page 63.

Examples

Example 1: A simple POINTER statement:

```fortran
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:

```fortran
  * 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. You then assign the address of \( A \) to \( P \), so now any use of \( V \) refers to \( A \) by the pointer \( P \). The program prints 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, 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:

```fortran
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
```

This is a slightly more realistic example. The size might well be some large number, say, 10,000. Once that’s allocated, the subroutines perform their tasks, not knowing that the array was dynamically allocated.
Example 6: One way to use pointers to make a linked list in F77:

demo% cat 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
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
```
Remember:

- Do not optimize programs using pointers like this with -O3, -O4, or -O5.
- The warnings can be ignored.
- This is not the normal usage of pointers described at the start of this section.

```
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%
```
PRINT

The PRINT statement writes from a list to stdout.

PRINT f [, iolist]
PRINT grname

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Format identifier</td>
</tr>
<tr>
<td>iolist</td>
<td>List of variables, substrings, arrays, and records</td>
</tr>
<tr>
<td>grname</td>
<td>Name of the namelist group</td>
</tr>
</tbody>
</table>

Description

The PRINT statement accepts the following arguments.

Format Identifier

f is a format identifier and can be:
- An asterisk (*), which indicates list-directed I/O. See “List-Directed I/O” on page 312 on for more information.
- 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 that specifies the format string. The integer array is nonstandard. *

Output List

iolist can be empty or can contain output items or implied DO lists. The output items must be one of the following:
- Variables
- Substrings
- Arrays
- Array elements
- 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.

Implied DO lists are described on “Implied DO Lists” on page 119.

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. Doing so reduces the relative risk of a program freeze.

FORTRAN I/O from an exception handler amounts to recursive I/O. See the next point.

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 may occur. This risk exists independent of parallelization.

Example: Recursive I/O fails intermittently:

```fortran
PRINT *, x, f(x)   Not allowed because f() does I/O.
END
FUNCTION F(X)
PRINT *, X
RETURN
END
```
Examples

Example 1: Formatted scalars:

```fortran
CHARACTER TEXT*16
PRINT 1, NODE, TEXT
1 FORMAT (I2, A16)
```

Example 2: List-directed array:

```fortran
PRINT *, I, J, (VECTOR(I), I = 1, 5)
```

Example 3: Formatted array:

```fortran
INTEGER VECTOR(10)
PRINT '(12 I2)', I, J, VECTOR
```

Example 4: Namelist:

```fortran
CHARACTER LABEL*16
REAL QUANTITY
INTEGER NODE
NAMELIST /SUMMARY/ LABEL, QUANTITY, NODE
PRINT SUMMARY
```
PROGRAM

The PROGRAM statement identifies the program unit as a main program.

PROGRAM  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgm</td>
<td>Symbolic name of the main program</td>
</tr>
</tbody>
</table>

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 uppercase), or a runtime error results

The name of the program can be the same as a local name in the main program. The FORTRAN 77 Standard does not allow this practice.

Example

Example: A PROGRAM statement:

```fortran
PROGRAM US_ECONOMY
  NVARS = 2
  NEQS = 2
  ...
```
READ

The READ statement reads data from a file or the keyboard to items in the list.

**Note** – Use the `OPEN()` routines to read from tape devices. See the *Fortran Library Reference Manual.*

```fortran
READ f [, iolist]
READ([UNIT=] u, [NML=] grname [, IOSTAT=ios] [, END=s] [, ERR=s])
READ grname
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>Unit identifier of the unit connected to the file</td>
</tr>
<tr>
<td>f</td>
<td>Format identifier</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier</td>
</tr>
<tr>
<td>rn</td>
<td>Record number to be read</td>
</tr>
<tr>
<td>s</td>
<td>Statement label for end of file processing</td>
</tr>
<tr>
<td>iolist</td>
<td>List of variables</td>
</tr>
<tr>
<td>grname</td>
<td>Name of a namelist group</td>
</tr>
</tbody>
</table>

An alternate to the `UNIT=`, `REC=` form is as follows:

```fortran
READ( u 'rn ... ) iolist
```

The options can be specified in any order.

**Description**

The READ statement accepts the following arguments.

**Unit Identifier**

*u* is either an external unit identifier or an internal file identifier.
An external unit identifier must be one of these:

- A nonnegative integer expression
- An 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:

- An asterisk (*), indicating list-directed I/O. See “List-Directed I/O” on page 312 for more information.
- A 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. *

See “Runtime Formats” on page 305 for details on formats evaluated at runtime.

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 or implied DO lists. The input items can be any of the following:

- Variables
- Substrings
- Arrays
- Array elements
- 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.

Implied DO lists are described on “Implied DO Lists” on page 119.

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

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.
   ios 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.

There are two forms of READ:

```fortran
READ f [, iolist]
READ ([NML= ] grname)
```

The above two forms operate the same way as the others, except that reading from the keyboard is implied.

Execution has the following differences:
- 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.
- If an end-of-line, CR, or NL is reached before the input list is satisfied, input continues from the next line.
- 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 equivalent to opening the file with the options in the following example:

```fortran
OPEN (u, FILE='FORT.u', STATUS='OLD', ACCESS='SEQUENTIAL',
&              FORM=fmt)
```

Note also:
- The value of fmt is 'FORMATTED' or 'UNFORMATTED' accordingly, as the read is formatted or unformatted.
■ A 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, 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:

```fortran
DIMENSION V(5)
READ( 3, '(5F4.1)') V
```

Example 6: Namelist-directed read:

```fortran
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)
```
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.

`REAL [len [, ]] v[len [/c/]] [, v[len [/c/]]] ...`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Name of a variable, symbolic constant, array, array declarator, function, or dummy function</td>
</tr>
<tr>
<td>len</td>
<td>Either 4, 8, or 16 (SPARC only), the length in bytes of the symbolic constant, variable, array element, or function</td>
</tr>
<tr>
<td>c</td>
<td>List of constants for the immediately preceding name</td>
</tr>
</tbody>
</table>

Description

Following are descriptions for REAL, REAL*4, REAL*8, and REAL*16.
REAL

For a declaration such as REAL \( W \), the variable \( W \) is usually a REAL*4 element in memory, interpreted as a real number. Specifying the size is nonstandard. *

The default size, for a declaration such as REAL \( H \), can be altered by compiling with any of the options \(-\text{dbl}, -r8, \) or \(-\text{xtypemap}\). See the discussion in Chapter 2 for details.

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.

Examples

Example 1: Simple real variables—these declarations are all equivalent:

\[
\begin{align*}
\text{REAL} & \ U, \ V(9) \\
\text{REAL*4} & \ U, \ V(9) \\
\text{REAL} & \ U*4, \ V(9) *4
\end{align*}
\]

Example 2: Initialize variables (REAL*16 is SPARC only):

\[
\begin{align*}
\text{REAL} & \ U/1.0/, \ V/4.3/, \ D*8/1.0/, \ Q*16/4.5/
\end{align*}
\]
Example 3: Specify dimensions for some real arrays:

```fortran
REAL A(10,100), V(10)
REAL X*4(10), Y(10)*4
```

Example 4: Initialize some arrays:

```fortran
REAL A(10,100) / 1000 * 0.0 /, B(2,2) / 1.0, 2.0, 3.0, 4.0/
```

Example 5: Double and quadruple precision (REAL*16 is SPARC only):

```fortran
REAL*8  R
REAL*16 Q
DOUBLE PRECISION D
```

In the above example, D and R are both double precision; Q is quadruple precision.
RECORD

The RECORD statement defines variables to have a specified structure, or arrays to be arrays of variables with such structures.

```plaintext
RECORD /struct-name/ record-list [, /struct-name/ record-list]...
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>struct_name</code></td>
<td>Name of a previously declared structure</td>
</tr>
<tr>
<td><code>record_list</code></td>
<td>List of variables, arrays, or array declarators</td>
</tr>
</tbody>
</table>

### 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.

Structures, fields, and records are discussed in “Structures” on page 56.

### 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.
- Record fields are not allowed in COMMON statements.
Records and record fields are not allowed in DATA, EQUIVALENCE, NAMELIST, PARAMETER, AUTOMATIC, STATIC, or SAVE statements. To initialize records and record fields, use the STRUCTURE statement. See “STRUCTURE” on page 244 for more information.

Example

Example 1: Declare some items to be records of a specified structure:

```fortran
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:

```fortran
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(1X I5/1X A16/1X F8.2/1X A8)
END
```
The above program produces the following output:

```
82
CacheBoard
1000.00
96K
```
RETURN

A RETURN statement returns control to the calling program unit.

RETURN \[ e \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e )</td>
<td>Expression of type INTEGER or REAL</td>
</tr>
</tbody>
</table>

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) \( \ast \) 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 that contains 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
```
RE W I N D

The *REWIND* statement positions the file associated with the specified unit to its initial point.

**Note** – Use the **TOPEN()** routines to rewind tape devices. See the *Fortran Library Reference* for details.

**REWIND**

```fortran
REWIND  u
REWIND ([UNIT=] u [, IOSTAT=ios] [, ERR= s])
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>u</strong></td>
<td>Unit identifier of an external unit connected to the file. <strong>u</strong> must be connected for <em>sequential access</em>, or <em>append access</em>.</td>
</tr>
<tr>
<td><strong>ios</strong></td>
<td>I/O specifier, an integer variable or an integer array element</td>
</tr>
<tr>
<td><strong>s</strong></td>
<td>Error specifier: <strong>s</strong> must be the label of an executable statement in the same program in which this <em>REWIND</em> statement occurs. The program control is transferred to this label in case of an error during the execution of the <em>REWIND</em> statement.</td>
</tr>
</tbody>
</table>

**Description**

The options can be specified in any order.

Rewinding a unit not associated with any file has no effect. Likewise, *REWIND* in a terminal file has no effect either.

Using a *REWIND* statement on a direct-access file is not defined in the FORTRAN 77 Standard, and is unpredictable.
Examples

Example 1: Simple form of unit specifier:

```fortran
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:

```fortran
INTEGER CODE
...
REWIND (UNIT = 3)
REWIND (UNIT = 3, IOSTAT = CODE, ERR = 100)
...
100 WRITE (*,*) 'error in rewinding'
STOP
```
SAVE

The SAVE statement preserves items in a subprogram after the RETURN or END statements are executed, preventing them from becoming undefined.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Name of an array, variable, or common block (enclosed in slashes), occurring in a subprogram</td>
</tr>
</tbody>
</table>

Description

`SAVE` variables are placed in an internal static area. All common blocks are already preserved because they have been allocated to a static area. Therefore, common block names specified in `SAVE` statements are allowed but ignored.

A `SAVE` statement is optional in the main program and has no effect.

A `SAVE` with no list preserves all local variables and arrays in the routine.

Local variables and arrays are already static by default, predisposing the need for `SAVE`. However, using `SAVE` can ensure portability, especially with routines that leave a subprogram by some way other than a `RETURN`.

Restrictions

The following constructs 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: A SAVE statement:

```
SUBROUTINE FFA(N)
DIMENSION A(1000,1000), V(1000)
SAVE A
...
RETURN
END
```
Statement Function

A statement function statement is a function-like declaration, made in a single statement.

fun ([d1, d2]...) = e

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun</td>
<td>Name of statement function being defined</td>
</tr>
<tr>
<td>d</td>
<td>Statement function dummy argument</td>
</tr>
<tr>
<td>e</td>
<td>Expression. e can be any of the types arithmetic, logical, or character.</td>
</tr>
</tbody>
</table>

Description

If a statement function is referenced, the defined calculations are inserted.

Example: The following statement is a statement function:

\[
\text{ROOT}( A, B, C ) = (-B + \sqrt{B^2-4.0*A*C})/(2.0*A)
\]

The statement function argument list indicates the order, number, and type of arguments for the statement function.

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.
Restrictions

Note these 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 \textit{fun} and \textit{e} are the same as those for the assignment statement. The type of \textit{fun} and \textit{e} can be different, in which case \textit{e} is converted to the type of \textit{fun}.
- The actual arguments must agree in order, number, and type with corresponding dummy arguments.
- If a dummy argument is defined as a structure, the corresponding actual argument must be similarly defined as the same structure.
- A dummy argument cannot be an array or function name, or have the same name as the function.
- The same argument cannot be specified more than once in the argument list.
- The statement function may 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 \texttt{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 a dummy argument of the statement function, not the local variable of the same name.
- The length specification of a character statement function or its dummy argument of type \texttt{CHARACTER} must be an integer constant expression.
- A statement function cannot be invoked recursively.
Examples

Example 1: Arithmetic statement function:

```plaintext
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:

```plaintext
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:

```plaintext
CHARACTER FIRST*1, STR*16, S*1
FIRST(S) = S(1:1)
READ( *, * ) STR
IF ( FIRST(STR) .LT. " " ) CALL CONTROL ( S, A )
...
```
The `STATIC` statement ensures that the specified items are stored in static memory.

```
STATIC list
```

### Description

All local variables and arrays are classified static by default: 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.

However, you can still use `STATIC` to ensure portability, especially with routines that leave a subprogram by some way other than a `RETURN`.

Also note that:

- 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` does not declare the variable `X` to be both `STATIC` and `REAL`; it declares the variable `REALX` to be `STATIC`.

### Example

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

The STOP statement terminates execution of the program.

\[ \text{STOP } [str] \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>String of no more than 5 digits or a character constant</td>
</tr>
</tbody>
</table>

Description

The argument \( str \) is displayed when the program stops.
If \( str \) is not specified, no message is displayed.

Examples

Example 1: Integer:

\[
\text{stop 9}
\]

The above statement displays:

\[
\text{STOP: 9}
\]

Example 2: Character:

\[
\text{stop 'error'}
\]

The above statement displays:

\[
\text{STOP: error}
\]
STRUCTURE

The STRUCTURE * statement organizes data into structures.

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

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, which can include initial values

Description

A STRUCTURE statement 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.

Structures, fields, and records are described in “Structures” on page 56.

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 \texttt{f77} types are allowed, subject to the following rules and restrictions:

- Any dimensioning needed must be in the type statement. The \texttt{DIMENSION} statement has no effect on field names.

- You can specify the pseudo-name \texttt{%FILL} for a field name. The \texttt{%FILL} is provided for compatibility with other versions of FORTRAN. It is not needed in \texttt{f77} because the alignment problems are taken care of for you. It is a useful feature if you want to make one or more fields not referenceable in some particular subroutine. The only thing that \texttt{%FILL} does is provide a field of the specified size and type, and preclude referencing it.

- You must explicitly type all field names. The \texttt{IMPLICIT} statement does not apply to statements in a STRUCTURE declaration, nor do the implicit \texttt{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.

In a structure declaration, the offset of field \texttt{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:

```fortran
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:

```fortran
STRUCTURE /VARLENSTR/
    INTEGER*4 NBYTES
    CHARACTER A*25
END STRUCTURE
RECORD /VARLENSTR/ VLS
VLS.NBYTES = 0
```
SUBROUTINE

The SUBROUTINE statement identifies a named program unit as a subroutine, and specifies arguments for it.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>Name of subroutine subprogram</td>
</tr>
<tr>
<td>d</td>
<td>Variable name, array name, record name, or dummy procedure name, an asterisk, or an ampersand</td>
</tr>
</tbody>
</table>

**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
```

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
```

In this example, the RETURN 1 statement refers to the first alternate return label (first *). The RETURN 2 statement refers to the second alternate return label (second *) specified in the SUBROUTINE statement.
**TYPE**

The **TYPE** statement writes to stdout.

**TYPE** *f* [, *iolist*]

**TYPE** *grname*

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>f</em></td>
<td>Format identifier</td>
</tr>
<tr>
<td><em>iolist</em></td>
<td>List of variables, substrings, arrays, and records</td>
</tr>
<tr>
<td><em>grname</em></td>
<td>Name of the namelist group</td>
</tr>
</tbody>
</table>

---

**Description**

The **TYPE** statement is provided for compatibility and is equivalent to:

- **PRINT** *f* [, *iolist*]
- **WRITE**(*, *grname*)
- **WRITE**(*, *iolist*)
- **WRITE**(*, *grname*)

**Example**

Example: Formatted and namelist output:

```
INTEGER V(5)
REAL X(9), Y
NAMELIST /GNAM/ X, Y
...
TYPE 1, V
1 FORMAT( 5 I3 )
...
TYPE GNAM
...```
The Type Statement

The type statement specifies the data type of items in the list, optionally specifies array dimensions, and initializes with values.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type</code></td>
<td>One of the following: BYTE * CHARACTER CHARACTER<em>n (where n is greater than 0) CHARACTER</em>(<em>) COMPLEX COMPLEX</em>8 * COMPLEX<em>16 * COMPLEX</em>32 * (SPARC only) DOUBLE COMPLEX * INTEGER INTEGER<em>2 * INTEGER</em>4 * INTEGER<em>8 * LOGICAL LOGICAL</em>1 * LOGICAL<em>2 * LOGICAL</em>4 * LOGICAL<em>8 * REAL REAL</em>4 * REAL<em>8 * REAL</em>16 * (SPARC only) DOUBLE PRECISION</td>
</tr>
<tr>
<td><code>v</code></td>
<td>Variable name, array name, array declarator, symbolic name of a constant, statement function or function subprogram name</td>
</tr>
<tr>
<td><code>clist</code></td>
<td>List of constants. There are more details about <code>clist</code> in the section on the DATA statement.</td>
</tr>
</tbody>
</table>

`type` can be preceded by either AUTOMATIC or STATIC.

Description

A `type` statement can be used to:
• Confirm or to override the type established by default or by the `IMPLICIT` statement
• Specify dimension information for an array, or confirm the type of an intrinsic function
• Override the length by one of the acceptable lengths for that data type

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:

```plaintext
type VariableName / constant / ...
```
or

```plaintext
type ArrayName / constant, ... /
```
or

```plaintext
type ArrayName / r*constant /
```

Example: Various type statements:

```plaintext
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 / 
```

When you initialize a data type, remember the following restrictions:

• For a simple variable, there must be exactly one constant.
• If any element of an array is initialized, all must be initialized.
• 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. For example:

```plaintext
INTEGER Z / 4 / 
POINTER ( x, Z )  Warning issued, not initialized
```

In this case, the compiler issues a warning message, and Z is not initialized.

If a variable or array is not initialized, its values are undefined.
If such initialization statements involve variables in COMMON, and the –ansi compiler flag is set, then a warning is issued.

**Note** – Compiling with any of the options –dbl, –r8, –i2, or –xtypemap can alter the default size of names typed without an explicit size. See the discussion in Chapter 2.

**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:

```fortran
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
- TITLE is initialized to 'Heading'
UNION and MAP

The UNION * statement defines groups of fields that share memory at runtime.

The syntax of a UNION declaration is as follows:

```
UNION
  MAP
    field-declaration
    field-declaration
    ...
  ENDMAP
  MAP
    field-declaration
    field-declaration
    ...
  ENDMAP
END UNION
```

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. Also:

- 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.

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

- Structure declaration
- Record
- Union declaration
- Declaration 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; PERSON.CREDITS references a field from the second map.

- If the variables of the first 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.
VIRTUAL

The VIRTUAL * statement is treated the same as the DIMENSION statement.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Name of an array</td>
</tr>
<tr>
<td>a(d)</td>
<td>Specifies the dimension of the array. It is a list of 1 to 7 declarators separated by commas</td>
</tr>
</tbody>
</table>

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
```
**VOLATILE**

The `VOLATILE` statement prevents optimization on the specified items.

```
VOLATILE nlist
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nlist</code></td>
<td>List of variables, arrays, or common blocks</td>
</tr>
</tbody>
</table>

**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.
WRITE

The WRITE statement writes data from the list to a file.

Note – For tape I/O, use the TOPEN() routines.

WRITE([ UNIT=] u, [NML=] grname [, IOSTAT=ios] [, ERR=s])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>Unit identifier of the unit connected to the file</td>
</tr>
<tr>
<td>f</td>
<td>Format identifier</td>
</tr>
<tr>
<td>ios</td>
<td>I/O status specifier</td>
</tr>
<tr>
<td>rn</td>
<td>Record number</td>
</tr>
<tr>
<td>s</td>
<td>Error specifier (statement label)</td>
</tr>
<tr>
<td>iolist</td>
<td>List of variables</td>
</tr>
<tr>
<td>grname</td>
<td>Name of the namelist group</td>
</tr>
</tbody>
</table>

The options can be specified in any order.

An alternate for the REC=rn form is allowed, as follows:

```
WRITE( u ' rn ... ) iolist *
```

See Example 3, later on in this section.

Description

Unit Identifier

u is either an external unit identifier or an internal file identifier.

An external unit identifier must be one of the following:

- A nonnegative integer expression
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 “List-Directed I/O” on page 312 for more information.
- 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 that specifies the format string. This is called a runtime format or a variable format. The integer array is nonstandard. *

See “Runtime Formats” on page 305 for details on formats evaluated at runtime.

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 it can contain output items or implied DO lists. The output items must be one of the following:

- Variables
- Substrings
- Arrays
- Array elements
- 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.

Implied DO lists are described in “Implied DO Lists” on page 119.

If the output item is a character expression that employs the concatenation operator, the length specifiers of its operands can be an asterisk (*). This rule is nonstandard.*

If a function appears in the output list, that function must not cause an input/output statement to be executed.

Namelist-Directed 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

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

Note these 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. Doing so reduces the relative risk of a system freeze. FORTRAN I/O from an exception handler amounts to recursive I/O. See the 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 results. This risk exists independent of using parallelization.

Example: Recursive I/O fails intermittently:

```
WRITE(*,*) x, f(x) Not allowed because 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:

```fortran
WRITE( *, * ) A, V
or
PRINT *, A, V
```

Example 5: Formatted write to an internal file:

```fortran
CHARACTER CA*16, L*8 /'abcdefgh'/, R*8 /'ijklmnop'/
WRITE( CA, 1 ) L, R
1 FORMAT( 2 A8 )
```

Example 6: Write an entire array:

```fortran
DIMENSION V(5)
WRITE( 3, '(5F4.1)') V
```

Example 7: Namelist-directed write:

```fortran
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 )
```
CHAPTER 5

Input and Output

This chapter describes the general concepts of FORTRAN input and output, and provides details on the different kinds of I/O. See also the Input/Output chapter in the Fortran Programming Guide.

Essential FORTRAN I/O Concepts

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 Solaris releases, this limit is 256. A FORTRAN program can increase this limit beyond 64 by calling the setrlim() function. See the setrlim(2) man page. 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 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 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 causes 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=I to branch on end-of-file. File position and the value of I/O list items are undefined following an error. END= catches both EOF and error conditions; ERR= catches only error conditions.

If your 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 is 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 causes 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 kinds of I/O supported by **f77** are: formatted, unformatted, binary*, 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 files and internal files. 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 a field of a structured record.

Combinations of I/O

I/O combinations on external files are:

<table>
<thead>
<tr>
<th>Allowed</th>
<th>Not Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential unformatted</td>
<td>Direct-access, list-directed I/O</td>
</tr>
<tr>
<td>Sequential formatted</td>
<td>Direct-access, NAMELIST I/O</td>
</tr>
<tr>
<td>Sequential list-directed</td>
<td>NAMELIST I/O on internal files</td>
</tr>
<tr>
<td>Sequential NAMELIST</td>
<td>Unformatted, internal I/O</td>
</tr>
<tr>
<td>Direct unformatted</td>
<td>Binary, direct-access*</td>
</tr>
<tr>
<td>Direct formatted</td>
<td>Binary, formatted*</td>
</tr>
<tr>
<td>Binary sequential unformatted*</td>
<td></td>
</tr>
</tbody>
</table>

The following table shows combinations of I/O form, access mode, and physical file types.

**TABLE 5-1** Summary of **f77** Input and Output

<table>
<thead>
<tr>
<th>Kind of I/O</th>
<th>Access Mode:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Sequential</td>
</tr>
<tr>
<td>Formatted</td>
<td>Internal</td>
</tr>
<tr>
<td>External</td>
<td>Only formatted records of same or variable length.</td>
</tr>
</tbody>
</table>
Avoid list-directed internal writes. The number of lines and items per line varies with the values of items.

## Printing 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.
  - If column 1 is +, it is replaced by a control sequence that causes a return to the beginning of the previous line.
- 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. FORM='PRINT' is for one file per call.

If you open a file with FORM='PRINT', then that file has the same content as if it was opened with FORM='FORMATTED', and filtered with the output filter, asa.

If you compile with the -oldldo option (old list-directed output), then all the 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 to switch between FORM='PRINT' and FORM='FORMATTED' 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:

```fortran
OPEN( UNIT=6, FORM='PRINT')
```

**Scratch Files**

Scratch files are temporary files that normally disappears after execution is completed.

Example: Create a scratch file:

```fortran
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 access later:

```fortran
CLOSE( UNIT=7, STATUS='KEEP' )
```

Remember to get the real name of the scratch file. Use 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. This routine:

- Recognizes carriage control for all formatted files.
- Ignores trailing and embedded blanks in input files.
- Positions files at the beginning or end upon opening.
- Preattaches file names of a specified pattern with logical units.
Example: IOINIT and logical unit preattachment:

```
CALL IOINIT ( .TRUE., .FALSE., .FALSE., 'FORT', .FALSE.)
```

For the above call, the FORTRAN runtime system looks in the environment for names of the form FORTnn, and then 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' )
```

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) and the Sun Fortran Library Reference.

Example: Attach external files ini1.inp and ini1.out to units 1 and 2:

In sh:

```
demo$ TST01=ini1.inp
demo$ TST02=ini1.out
demo$ export TST01 TST02
```

In csh:

```
demo% setenv TST01 ini1.inp
demo% setenv TST02 ini1.out
```
Example: Attach the files, `ini1.inp` and `ini1.out`, to units 1 and 2:

```fortran
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
```

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 from:

```
/opt/SUNWspro/<release>/src/ioinit.f
```

The `<release>` path changes with each release of the compiler.

---

**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.

In direct 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.
■ In using direct unformatted I/O, 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 code 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 code 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.
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; similarly for a write.

Example: Sequential, formatted reads:

```fortran
CHARACTER X*80
READ( 5, '(A)' ) X
READ( X, '(I3,I4)' ) N1, N2
```

The above code 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 the third record of the internal file, LINE:

```plaintext
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%
```

**Formatted I/O**

In formatted I/O:
- 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 sequential access reads or writes one or more logical records.

**Input Actions**

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 according to the instructions in the associated format.
- Puts converted data into internal storage for each list item of the list.

Example: Formatted read:

```plaintext
READ( 6, 10 ) A, B
10 FORMAT( F8.3, F6.2 )
```
Output Actions

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 according to the instructions in the associated format.
- Transfers the items to the external record or to an internal file.
- Terminates formatted output records 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, the logical record length is determined by the format statement that interacts 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>
<thead>
<tr>
<th>Purpose</th>
<th>FORTRAN 77</th>
<th>77 Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank control</td>
<td>BN, BZ</td>
<td>B</td>
</tr>
<tr>
<td>Carriage control</td>
<td>/, space, 0, 1</td>
<td>$</td>
</tr>
<tr>
<td>Character edit</td>
<td>nh, nh, 'aaa'</td>
<td>&quot;aaa&quot;, A</td>
</tr>
<tr>
<td>Floating-point edit</td>
<td>Dw.dEe, Ew.dEe, D, dEe, Gw.dEe</td>
<td></td>
</tr>
<tr>
<td>Hexadecimal edit</td>
<td>Zw.m</td>
<td></td>
</tr>
<tr>
<td>Integer edit</td>
<td>Tw.m</td>
<td></td>
</tr>
<tr>
<td>Logical edit</td>
<td>Lw</td>
<td></td>
</tr>
<tr>
<td>Octal edit</td>
<td>Ow.m</td>
<td></td>
</tr>
<tr>
<td>Position control</td>
<td>nX, Tn, TLn, TRn</td>
<td>nT, T, X</td>
</tr>
<tr>
<td>Radix control</td>
<td>nR, R</td>
<td></td>
</tr>
<tr>
<td>Remaining characters</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>Scale control</td>
<td>nP</td>
<td></td>
</tr>
<tr>
<td>Sign control</td>
<td>S, SP, SS</td>
<td></td>
</tr>
<tr>
<td>Terminate a format</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>Variable format expression</td>
<td>&lt; e &gt;</td>
<td></td>
</tr>
</tbody>
</table>

Specifiers can be uppercase as well as lowercase characters in format statements and in all the alphabetic arguments to the I/O library routines.

\( w, m, d, e \) Parameters (As In Gw.dEe)

The definitions for the parameters, \( w, m, d, \) and \( e \) are:

- \( w \) and \( e \) are non-zero, unsigned integer constants.
- \( d \) and \( m \) are unsigned integer constants.
- \( w \) 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 \( w \), \( d \), and \( e \)

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 are 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:

\[ A_w, I_w, L_w, O_w, Z_w, D_w.d, E_w.d, G_w.d, E_w.d_e, G_w.d_E \]

Example: With the default \( w=7 \) for \( \text{INTEGER}^*2 \), and since 161 decimal = A1 hex:

```
INTEGER*2 M
M = 161
WRITE ( *, 8 ) M
8 FORMAT ( Z )
END
```

This example produces the following output:

```
demo% f77 def1.f
def1.f:
 MAIN:
demo% a.out
ΔΔΔΔΔΔΔΔ
demo%
```

\( \Delta \) represents a blank character position. The defaults for \( w, d, \) and \( e \) are summarized in the following table.

<table>
<thead>
<tr>
<th>Field Descriptor</th>
<th>List Element</th>
<th>( w )</th>
<th>( d )</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I, O, Z</td>
<td>BYTE</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I, O, Z</td>
<td>( \text{INTEGER}^*2, \text{LOGICAL}^*2 )</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>I, O, Z</td>
<td>( \text{INTEGER}^*4, \text{LOGICAL}^*4 )</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O, Z</td>
<td>REAL*4</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O, Z</td>
<td>REAL*8</td>
<td>23</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O, Z</td>
<td>( \text{REAL}^*16, \text{COMPLEX}^*32 )</td>
<td>44</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>L</td>
<td>LOGICAL</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>F, E, D, G</td>
<td>REAL, COMPLEX*8</td>
<td>15</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
For complex items, the value for $w$ is for each real component. The default for the A descriptor with character data is the declared length of the corresponding I/O list element. REAL*16 and COMPLEX*32 are **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 of the edit specifier itself, including blanks. 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):

```fortran
WRITE(*,1)
1 FORMAT( 'This is an apostrophe ''.'')
WRITE(*,2)
2 FORMAT( "This is an apostrophe ".")
END
```

The above program writes this message twice: **This is an apostrophe '.**
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 the 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 \(BN\):

```fortran
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
```

Note these rules 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)

Use edit descriptor $, and space, 0, or 1 for carriage control.

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 descriptor 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:

```fortran
* doll.f The $ edit descriptor with space
  WRITE ( *, 2 )
  2 FORMAT (' Enter the node number: ', $ )
  READ ( *, * ) NODENUM
  END
```

The above code 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, and +

The following first-character slew controls and actions are provided:

<table>
<thead>
<tr>
<th>Character</th>
<th>Vertical spacing before printing</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>One line</td>
</tr>
<tr>
<td>0</td>
<td>Two lines</td>
</tr>
<tr>
<td>1</td>
<td>To first line of next page</td>
</tr>
<tr>
<td>+</td>
<td>No advance (stdout only, not files)</td>
</tr>
</tbody>
</table>

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: if the character in the first column is +, it is replaced by a control sequence that causes printing to return to the first column of the previous line, where the rest of the input line is printed.

Space, 0, 1, and + work for stdout if piped through asa.

Example: First-character formatting, standard output piped through asa:

```
demo% cat slewl.f
    WRITE( *, '("abcd")')  The blank single spaces
    WRITE( *, '( "efg")')  The "0" double spaces
    WRITE( *, '( "0hij")')  The "1" starts this on a new page
    WRITE( *, '("1klm")')  The "+" starts this at col 1 of latest line
    END

demo% f77 -silent slewl.f

demo% a.out | asa | lpr

demo% cat slewl.out
    bcd
    efg
    hij
    klmnop  This starts on a new page. The + of +nop is obeyed.

demo% cat slewl.out
    bcd
    efg
    hij
    nop  This starts on a new page. The + of +nop is obeyed.

demo%
```

The program, slewl.f produces file, slewl.out, as printed by lpr:

The results are different on a screen; the tabbing puts in spaces:

```
bcd
efg

hij

klmnop  This starts on a new page. The + of +nop is obeyed.
```

See asa(1).

The space, 0, and 1, and + 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, '("1kim")')
WRITE ( 1, '("+", T5, "nop")')
CLOSE ( 1, STATUS='KEEP')
END
demo% f77 -silent slew2.f
demo% a.out
```

The program, `slew2.f`, produces the file, `slew2.out`, that is equal to the file, `slew1.out`, in the example above.

Slew control codes '0', '1', and '+' in column one are in the output file as '\n', '\f', and '\r', 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 behavior.

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
```
The various values of \( n \), in \texttt{CHARACTER C*} are:

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Size } n & 9 & 7 & 4 & 1 \\
\hline
\text{Data} & \text{NodeAId} & \text{NodeAId} & \text{NodeAId} & \text{NodeAId} \\
\text{Format} & \text{A7} & \text{A7} & \text{A7} & \text{A7} \\
\text{Memory} & \text{NodeAIdA} & \text{NodeAId} & \text{eAId} & \text{d} \\
\hline
\end{array}
\]

\( \Delta \) indicates a blank space.

Example: Output strings of 3, 5, and 7 characters, each in a 5 character field:

```fortran
PRINT 1, 'The', 'whole', 'shebang'
1 FORMAT( A5 / A5 / A5 )
END
```

The above program displays:

```
The
whole
sheba
```
The maximum characters in noncharacter types are summarized in the following table.

**TABLE 5-5** Maximum Characters in Noncharacter Type Hollerith (nHaaa)

<table>
<thead>
<tr>
<th>Type of List Item</th>
<th>Maximum Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL*1</td>
<td>1</td>
</tr>
<tr>
<td>LOGICAL*2</td>
<td>2</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL*8</td>
<td>8</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>4</td>
</tr>
<tr>
<td>INTEGER*8</td>
<td>8</td>
</tr>
<tr>
<td>REAL</td>
<td>4</td>
</tr>
<tr>
<td>REAL*4</td>
<td>4</td>
</tr>
<tr>
<td>REAL*8</td>
<td>8</td>
</tr>
<tr>
<td>REAL*16 (SPARC only)</td>
<td>16</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>8</td>
</tr>
<tr>
<td>COMPLEX*8</td>
<td>8</td>
</tr>
<tr>
<td>COMPLEX*16</td>
<td>16</td>
</tr>
<tr>
<td>COMPLEX*32 (SPARC only)</td>
<td>32</td>
</tr>
<tr>
<td>DOUBLE COMPLEX</td>
<td>16</td>
</tr>
</tbody>
</table>

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 does not have these old Hollerith (nH) notations, although the FORTRAN Standard recommends implementing the Hollerith feature to improve compatibility with old programs. But such constants cannot be used as input data elements in list-directed or NAMELIST input.

For example, these two formats 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 )
```
Reading Into Hollerith Edit Descriptors

For compatibility with older programs, f77 also allows reads into Hollerith edit descriptors.

Example: Read into hollerith edit descriptor—no list in the READ statement:

```
demo% cat holl.f
  WRITE( *, 1 )
  1  FORMAT( 6Holder )
  READ( *, 1 )
  WRITE( *, 1 )
END

demo% f77 holl.f
holl.f:
  MAIN
demo% a.out
older
newer
newer
demo%
```

In the above code, if the format is a runtime format (variable format), then the reading into the actual format does not work, and the format remains unchanged. Hence, the following program fails:

```
CHARACTER F*18 / '(A8)' /
READ(*,F) ! ← Does not work.
...`

Obviously, there are better ways to read into the actual format.

Integer Editing (I)

The I specifier is used for decimal integer data items. The general form is:

```
I[w[m]]
```

The Iw and Iw.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 becomes defined with an integer datum. On output, the specified list item must be defined as an integer datum.
On input, an \( IW \) edit specifier is treated identically to an \( IW \) edit specifier.

The output field for the \( IW \) 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 \( IW.m \) edit specifier is the same as for the \( IW \) 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:

```plaintext
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:

```plaintext
N = 1234
PRINT 1, N, N, N, N
1 FORMAT( I6 / I4 / I2 / I6.5 )
END
```

The above program displays:

```
1234
1234
**
01234
```
Logical Editing (L)

The **L** specifier is used for logical data items. The general form is:

\[
\text{L}w
\]

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 becomes 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:

```fortran
PRINT '( L1 / L2 / L4 )', A, B, C
END
```

The program above displays:

```
T
ΔT
ΔΔΔF
```

Example: log2.f, logical input:

```fortran
LOGICAL*4 A
1 READ '(L8)', A
PRINT *, A
GO TO 1
END
```

The program above accepts any of the following as valid input data:

```
t true T TRUE .t .t .T .T .TRUE. TooTrue
f false F FALSE .f .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:

\[
\text{O}w \ [ .m ] \\
\text{Z}w \ [ .m ]
\]

where \( w \) is the number of characters in the external field. 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. \( m \) 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:

\[
654321 
\]

The first digit in the example appears in input column 1.

The program that does the input is:

\[
\begin{align*}
\text{READ} & \ ( \ * , \ 2 \ ) \ M \\
& \text{FORMAT} \ ( \ 06 \ )
\end{align*}
\]
The above data and program result in the octal value 654321 being loaded into the variable \( M \). Further examples are included in the following table.

**TABLE 5-6**  Sample Octal/Hex Input Values

<table>
<thead>
<tr>
<th>Format</th>
<th>External Field</th>
<th>Internal (Octal or Hex) Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>O4</td>
<td>1234(\Delta)</td>
<td>1234</td>
</tr>
<tr>
<td>O4</td>
<td>16234</td>
<td>1623</td>
</tr>
<tr>
<td>O3</td>
<td>97(\Delta)(\Delta)</td>
<td>Error: “9” not allowed</td>
</tr>
<tr>
<td>Z5</td>
<td>A23DE(\Delta)</td>
<td>A23DE</td>
</tr>
<tr>
<td>Z5</td>
<td>A23DEF</td>
<td>A23DE</td>
</tr>
<tr>
<td>Z4</td>
<td>95.AF2</td>
<td>Error: “.” not allowed</td>
</tr>
</tbody>
</table>

The general rules for octal and hex input are:
- 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:

```fortran
M = 161
WRITE ( *, 8 ) M
8 FORMAT ( Z3 )
END
```

The program above displays A1 (161 decimal = A1 hex):

```fortran
\( \Delta A1 \)
```

The letter A appears in output column 2.
Further examples are included in the following table.

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal (Decimal) Value</th>
<th>External (Octal/Hex) Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>32767</td>
<td>Δ77777</td>
</tr>
<tr>
<td>02</td>
<td>14251</td>
<td>**</td>
</tr>
<tr>
<td>04.3</td>
<td>27</td>
<td>Δ033</td>
</tr>
<tr>
<td>04.4</td>
<td>27</td>
<td>0033</td>
</tr>
<tr>
<td>06</td>
<td>-32767</td>
<td>*****</td>
</tr>
<tr>
<td>24</td>
<td>32767</td>
<td>7FFFF</td>
</tr>
<tr>
<td>23.3</td>
<td>2708</td>
<td>A94</td>
</tr>
<tr>
<td>26.4</td>
<td>2708</td>
<td>ΔΔ0A94</td>
</tr>
<tr>
<td>25</td>
<td>-32767</td>
<td>*****</td>
</tr>
</tbody>
</table>

The general rules for octal and hex output are:

- 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 \( m \).

**Positional Editing**

For horizontal positioning along the print line, '77 supports the forms:

\( Tn, \ TLn, \ TRn, \ nT, \ nx \)

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.

\( Tn — \text{Absolute Columns} \)

This tab reads from the \( n \)th column or writes to the \( n \)th column. If \( n \) is missing it is treated as \( T1 \).

\( TLn — \text{Relative Columns} \)

This tab reads from the \( n \)th column to the \( left \) or writes to the \( n \)th column to the \( left \). If \( n \) is missing it is treated as \( TL0 \).
**TRn—Relative Columns**

This tab reads from the \( n \)th column to the right or writes to the \( n \)th column to the right. If \( n \) is missing it is treated as TR0.

**nT—Relative Tab Stop**

This tab tabs to the next tab stop for both read and write. If \( n \) is omitted, this tab uses \( n = 1 \) and tabs to the next tab stop. (This edit specifier is not standard FORTRAN 77)

The rules and Restrictions for tabbing are:

- 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.

**nX—Positions**

The \( nX \) 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 \( nX \) 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 \( nX \) specifier writes \( n \) blanks.

The \( n \) defaults to 1.
Example: Input, TN (absolute tabs):

```fortran
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
```

The two-line data file is:

```bash
demo% cat mytab.data
defguvwx
12345678
demo%
```

The run and the output are:

```bash
demo% a.out
uvde
5612

demo%
```

The above example first reads columns 5 and 6, then columns 1 and 2.

Example: Output TN (absolute tabs); this program writes an output file:

```fortran
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
```

```bash
demo%
```
The output file is:

```
demo% cat mytab.rep
123456789:123456789:
demo%
```

The above example writes 20 characters, then changes columns 10 and 20.

Example: Input, \( TR_j \) and \( TL_n \) (relative tabs)—the program reads:

```
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 two-line data file is:

```
demo% cat mytab.data
  defguvwx
  12345678

demo%
```

The run and the output are:

```
demo% a.out
dwig
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:

```fortran
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
```

The above program writes 20 characters, tabs left 11 to column 10, then tabs 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 of the edit specifier itself, including blanks. 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):

```fortran
WRITE( *, 1 )
1 FORMAT( 'This is a quote "." )
WRITE( *, 2 )
2 FORMAT( "This is a quote "." )
END
```

This program writes this message twice: This is a quote ".".
Radix Control (R)

The format specifier is R or nR, where 2 ≤ n ≤ 36. If n is omitted, the default decimal radix is restored.

You can 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:

```
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
`0000006E`
demo%

SU is described in “Sign Editing (SU, SP, SS, S)” on page 303.

Editing REAL Data (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 Dw and Dw.d edit specifiers indicate that the field to be edited occupies w positions. 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 becomes 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 $Dw.d$ edit specifier has the 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.

$w$ 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 $D$ editing in the program, $Dinp.f$:

```plaintext
```

The above program displays:

```
0.123D+05 0.2357D+02 0.345678D+00
```

In the above example, the first input data item has no decimal point, so $D8.3$ 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 in the program $Dout.f$:

```plaintext
R = 1234.678
PRINT 1, R, R, R
1 FORMAT( D9.3 / D8.4 / D13.4 )
END
```

The above program 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:

\[ E \{ w \{ .d \} \{ e \} \} \]

`w` indicates that the field to be edited occupies `w` positions.

`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.

`e` indicates the number of digits in the exponent field. The default is 2.

The specified input/output list item must be of type real. On input, the specified list item becomes defined with a real datum. On output, the specified list item must be defined as a real datum.

The output field for the `Ew.d` edit specifier has the 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, a decimal point, the magnitude of the value of the list item rounded to `d` decimal digits, and an exponent.

For the form `Ew.d`:

- If \(|\text{exponent}| \leq 99\), it has the form `Enn` or `0nn`.
- If \(99 \leq |\text{exponent}| \leq 999\), it has the form `nnn`.

For the form `Ew.dEe`, if \(|\text{exponent}| \leq (10^e) - 1\), then the exponent has the form `nnn`.

For the form `Dw.d`:

- If \(|\text{exponent}| \leq 99\), it has the form `Dnn` or `Enn` or `0nn`.
- If \(99 \leq |\text{exponent}| \leq 999\), it has the form `nnn`.

`n` is any digit.

The sign in the exponent is required.

`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 \geq d+6`; if `e` is present, then `w \geq d+e+4`. For negative numbers, `w \geq d+7`; if `e` is present, then `w \geq d+e+5`.  

\[n\]
Example: Real input with $E$ editing in the program, Einp.f:

```
CHARACTER L*40//'123456789E2 1234.67E-3 12.45670 '/
PRINT '( E15.6, E15.6, E15.7 )', R, S, T
END
```

The above program displays:

```
0.123457E+06
0.123467E+01
0.1245670E+02
```

In the above example, the first input data item has no decimal point, so $E9.3$ 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 in the program Eout.f:

```
R = 1234.678
PRINT 1, R, R, R
1 FORMAT( E9.3 / E8.4 / E13.4 )
END
```

The above program displays:

```
0.123E+04
********
0.1235E+04
```

In the above example, $E8.4$ 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 $Ewde$ editing in the program EwdEe.f:

```
REAL X / 0.000789 /
WRITE(*,'( E13.3)') X
WRITE(*,'( E13.3E4)') X
WRITE(*,'( E13.3E5)') X
END
```
The above program displays:

```
ΔΔΔΔ0.789E-03
ΔΔ0.789E-0003
ΔΔ0.789E-00003
```

**F Editing**

The F specifier is for decimal real data items. The general form is:

```
F [ w[.d] ]
```

The Fw and Fw.d edit specifiers indicate that the field to be edited occupies w positions.

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 becomes defined with a real datum. On output, the specified list item must be defined as a real datum.

The output field for the Fw.d edit specifier has the 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.

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 in the program Finp.f:

```
CHARACTER LINE*24 / '12345678 23.5678 .345678' /
READ( LINE, '( F8.3, F8.3, F8.3 )') R, S, T
END
```

The program displays:

```
12345.678DD23.5678D0.345678
```
In the above example, the first input data item has no decimal point, so $F8.3$ 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 in the program $Fout.f$:

```plaintext
R = 1234.678
PRINT 1, R, R, R
1 FORMAT( F9.3 / F8.4 / F13.4 )
END
```

The above program displays:

```
Δ1234.678
*******
ΔΔΔΔ1234.6780
```

In the above example, $F8.4$ does not allow for the sign; $F13.4$ results in four leading blanks and one trailing zero.

**G Editing**

The $G$ specifier is for decimal real data items. The general forms are:

```
G[w[.d]]  or  Gw.dEe
```

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.

<table>
<thead>
<tr>
<th>Range</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.1 \leq N &lt; 1.0$</td>
<td>$F(w-4).d$</td>
</tr>
<tr>
<td>$1.0 \leq N &lt; 10.0$</td>
<td>$F(w-4).(d-1)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$10^{(d-2)} \leq N &lt; 10^{(d-1)}$</td>
<td>...</td>
</tr>
<tr>
<td>$10^{(d-1)} \leq N &lt; 10^d$</td>
<td>$F(w-4).1$</td>
</tr>
<tr>
<td>$10^d \leq N &lt; 10^{d+1}$</td>
<td>$F(w-4).0$</td>
</tr>
</tbody>
</table>
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 following record 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 the \(I_w, Fw.d, Ew.d[Ee]\), and \(Gw.d\) input fields, the field ends when \(w\) characters have been scanned, or a comma has been scanned, whichever occurs first. If it is 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 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 program 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 \( n \)th edit descriptor matches with \( L \) as the \( n \)th 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.
Several restrictions on the \( \mathbf{Q} \) edit descriptor apply:

- The list element \( \mathbf{Q} \) corresponds to must be of \texttt{INTEGER} or \texttt{LOGICAL} data type.
- \( \mathbf{Q} \) does strictly a character count. It gets the number of characters remaining in the input record, and does not get the number of integers or reals or anything else.
- The \( \mathbf{Q} \) edit descriptor cannot be applied for pipe files, as \( \mathbf{Q} \) edit requires that the file be rereadable.
- This descriptor operates on files and \texttt{stdin} (terminal) input.
- This descriptor is prohibited for output and will generate a runtime error.

**Scale Factor (\( \mathbf{P} \))**

The \( \mathbf{P} \) edit descriptor scales 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:

\[ \mathbf{[k]}\mathbf{P} \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
<td>Integer constant, with an optional sign</td>
</tr>
</tbody>
</table>

\( k \) is called the scale factor, and the default value is zero.

Example: \( \text{I/O statements with scale factors:} \)

```fortran
READ ( 1, '( 3P E8.2 )' ) X
WRITE ( 1, '( 1P E8.2 )' ) X
```

\( \mathbf{P} \) by itself is equivalent to \( 0 \mathbf{P} \). It resets the scale factor to the default value \( 0 \mathbf{P} \). Just \( \mathbf{P} \) 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 \( \mathbf{D}, \mathbf{E}, \mathbf{P}, \) and \( \mathbf{G} \) edit descriptors.
Input

On input, any external datum that does not have an exponent field is divided by $10^k$ before it is stored internally.

Input examples: Showing data, scale factors, and resulting value stored:

<table>
<thead>
<tr>
<th>Data</th>
<th>18.63</th>
<th>18.63</th>
<th>18.63E2</th>
<th>18.63</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>E8.2</td>
<td>3P E8.2</td>
<td>3P E8.2</td>
<td>-3P E8.2</td>
</tr>
<tr>
<td>Memory</td>
<td>18.63</td>
<td>.01863</td>
<td>18.63E2</td>
<td>18630</td>
</tr>
</tbody>
</table>

Output

On output, with D, and E descriptors, and with G descriptors if the E editing is required, the internal item gets its basic real constant part multiplied by $10^k$, and the exponent is 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 gets its basic real constant part multiplied by $10^k$ before it is written out.

Output Examples: Showing value stored, scale factors, and resulting output:

<table>
<thead>
<tr>
<th>Memory</th>
<th>290.0</th>
<th>290.0</th>
<th>290.0</th>
<th>290.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>2P E9.3</td>
<td>1P E9.3</td>
<td>-1P E9.3</td>
<td>F9.3</td>
</tr>
<tr>
<td>Display</td>
<td>29.00E+01</td>
<td>2.900E+02</td>
<td>0.029E+04</td>
<td>0.290E+03</td>
</tr>
</tbody>
</table>

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; 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, the 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 )
```

The rules and restrictions for sign control are:

- 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**

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.

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**

Each slash increases the record number by one, and the file is positioned at the start of the record with that record number.

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:

```fortran
* coll.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 like 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. Doing so improves flexibility. There is some increase in execution time because this kind of format specifier is 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:

```fortran
demo% cat runtim.f
CHARACTER CS*8
    CHARACTER CA(1:7)*1 /'(1,X,1,12)'/
    CHARACTER S(1:7)*6
    INTEGER*4 IA(2)
STRUCTURE / STR /
    CHARACTER*4 A
    INTEGER*4 K
END STRUCTURE
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 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:

```plaintext
1 FORMAT( 3F6.1 )
```

can be replaced by the variable N, as in:

```plaintext
1 FORMAT( 3F<N>.1 )
```

or by the slightly more complicated expression 2*N+M, as in:

```plaintext
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 nH... edit descriptor.

The rules and restrictions for variable format expressions are:

- 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 nH... edit descriptor cannot be a variable expression.

## 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.

Here are some 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—create 3 records with 2 integers each:

```fortran
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
```

```bash
demo% f77 -silent Direct1.f
demo% a.out
demo%
```

Example: Direct access—read the 3 records:

```fortran
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
```

```bash
demo% f77 -silent Direct2.f
demo% a.out
  4  5  6  7  8  9
demo%
```

Here we knew beforehand the size of the records on the file. In this case we can read the file just as it was written.

However, if we only know the size of each item but not the size of the records on a file we can use `recl=1` on the OPEN statement to have the I/O list itself determine how many items to read:
Example: Direct-access read, variable-length records, \( \text{recl}=1 \):

```
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%
```

In the above example, after reading 3 integers (12 bytes), you start the next read at record 13.

---

**Binary I/O**

Opening a file for binary I/O allows the program to read and write data as a stream of binary data without record boundaries. (This feature is not standard Fortran 77).

The `FORM='BINARY'` option on an `OPEN` statement declares that unit to be a sequential unformatted file without record marks:

```
OPEN(1, FORM='BINARY')
```

A binary file cannot also be declared direct-access or formatted.

On a `WRITE` statement, binary data is written to the file as a stream of bytes, as many as there are in the output list. On a `READ` statement, as many data bytes are read as demanded by the variables on the input list. Since there are no record marks (end-of-record) in the file, there will be no possibility of reading past a record mark. Other
than abnormal system errors, the only situation that the program can detect is reading in the end-of-file. Each `READ` statement just reads the next sequence of bytes in the file, as shown in the following example:

```fortran
program bin
  character *25 string
  character *5 word
  open(1, FORM='BINARY')
  string = 'alphabetagammaepsilon'
  write(1) string
  rewind 1
  do 1 i=1,6
    word = '     '
    read(1) word
1       print*, word
  end
```

An `INQUIRE` on a binary file returns ‘BINARY’ for the `FORM=` parameter, ‘SEQUENTIAL’ for `ACCESS=`, ‘YES’ for `UNFORMATTED=`, ‘YES’ for `SEQUENTIAL`. `BACKSPACE` on a binary file is not allowed and causes a runtime error message.
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
```

Note these 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 ("), pairs of apostrophes (’), or unquoted strings (see “Unquoted Strings” on page 314), but *not* hollerith (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:

```
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. Any text that follows 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.

Note these rules for list-directed output:
In general, each record starts with a blank space. For a print file, that blank is not printed. See “Printing Files” on page 266 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%
```

In the above example, if you need accuracy, specify the format.

Also note:

- 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 (\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

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>Two blanks followed by the number</td>
</tr>
<tr>
<td>CHARACTER*8</td>
<td>\n \n{\text{n = length of character expression}}</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>'AA(', 1PE14.5E2, ',', 1PE14.5E2, ')'</td>
</tr>
<tr>
<td>COMPLEX*16</td>
<td>'AA(', 1PE22.13.E2, ',', 1PE22.13.E2, ')'</td>
</tr>
<tr>
<td>COMPLEX*32</td>
<td>'AA(', 1PE44.34E3, ',', 1PE44.34E3, ')'</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td>Two blanks followed by the number</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>Two blanks followed by the number</td>
</tr>
<tr>
<td>INTEGER*8</td>
<td>Two blanks followed by the number</td>
</tr>
<tr>
<td>LOGICAL*1</td>
<td>Two blanks followed by the number</td>
</tr>
<tr>
<td>LOGICAL*2</td>
<td>L3</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>L3</td>
</tr>
<tr>
<td>LOGICAL*8</td>
<td>L3</td>
</tr>
<tr>
<td>REAL</td>
<td>1PE14.5E2</td>
</tr>
<tr>
<td>REAL*8</td>
<td>1PE22.13.E2</td>
</tr>
<tr>
<td>REAL*16</td>
<td>1PE44.34E4</td>
</tr>
</tbody>
</table>

COMPLEX*32 and REAL*16 are SPARC only.

### 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:

```fortran
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
 231label 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.

**NAMELIST I/O**

`NAMELIST` I/O produces 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 lists the variables and arrays of that group.
Syntax Rules

The syntax of the NAMELIST statement is:

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group-name</td>
<td>Name of group</td>
</tr>
<tr>
<td>namelist</td>
<td>List of variables or arrays, separated by commas</td>
</tr>
</tbody>
</table>

See “NAMELIST” on page 195 for details.

Example: NAMELIST statement:

```
CHARACTER*18 SAMPLE
LOGICAL*4 NEW
REAL*4 DELTA
NAMELIST /CASE/ SAMPLE, NEW, DELTA
```

A variable or array can be listed in more than one NAMELIST group.

The input data can include array elements and strings. It can include substrings in the sense that the input constant data string can be shorter than the declared size of the variable.

Restrictions

`group name` can appear in only the NAMELIST, READ, or WRITE statements, and must be unique for the program.

`list` cannot include constants, array elements, dummy assumed-size arrays, structures, substrings, records, record fields, pointers, or pointer-based variables.

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, \texttt{DELTA} is in the group \texttt{CASE} and in the group \texttt{GRID}.

**Output Actions**

\texttt{NAMELIST} output uses a special form of \texttt{WRITE} statement, which makes a report that shows the group name. 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 writes the report so that \texttt{NAMELIST} input can read it.

The syntax of \texttt{NAMELIST WRITE} is:

\begin{verbatim}
WRITE ( extu, namelist-specifier [, iostat] [, err])
\end{verbatim}

where \texttt{namelist-specifier} has the form:

\begin{verbatim}
[NML=]group-name
\end{verbatim}

and \texttt{group-name} has been previously defined in a \texttt{NAMELIST} statement.

The \texttt{NAMELIST WRITE} statement writes values of all variables in the group, in the same order as in the \texttt{NAMELIST} statement.

**Example: NAMELIST output:**

\begin{verbatim}
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
naml.f:
   MAIN:
demo% a.out
Δ&case sample= Demo, new= T, delta= 0.100000
Δ&end
demo%
\end{verbatim}
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.

The WRITE can have the form of the following example:

```
WRITE ( UNIT=6, NML=CASE )
```

### Input Actions

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.

If the $group-name is not found, the input records are read until end of file.

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 variables, SAMPLE, NEW, DELTA, and MAT. If you do omit the keyword NML, then you must also omit the keyword UNIT. The unit parameter must be first, namelist-specifier must be second, and there must not be a format specifier.
The READ can have the form of the following example:

```
READ ( UNIT=1, NML=CASE )
```

**Data Syntax**

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:

```
$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. 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 the next section.

The following is sample data to be read by the program segment above:

```
$case delta=0.05, mat( 2, 2 ) = 2.2, sample='Demo' $
```
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:

```fortran
$case
$delta=0.05
$mat( 2, 2 ) = 2.2
$sample='Demo'
$
```

Syntax Rules

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 starts in or after column two. It may start in column 1, but this is non-standard.
- 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.

Note an 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. 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:

```
\[\text{sample='use "$" in 2 (Read as: use $ in 2)}\]
\[\text{sample='don''t' (Read as: don't)}\]
\[\text{sample="don''t" (Read as: don't)}\]
\[\text{sample="don't" (Read as: don't)}\]
```

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, and so on.

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:

```
* 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:

```
$GRID ARRAY = 9,9,9,9,,,,,8,8,8,8 $
```

This code loads 9s into row 1, skips 4 elements, and loads 8s into row 3 of ARRAY.

**Arrays Only**

The forms r* and r* can be used only with an array.
The form \( r \times c \) 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:

```fortran
* nam3.f Name list "r*c" and "r*"
REAL PSI(10)
NAMELIST /GRID/ PSI
WRITE ( *, * ) 'Input?'
READ ( *, GRID )
WRITE ( *, GRID )
END
```

The input for nam3.f is:

```
$GRID PSI = 5*980$
```

The program, nam3.f, reads the above input and loads 980.0 into the first 5 elements of the array PSI.

- The form \( r \times \) 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.

The other input is:

```
$GRID PSI = 3* 5*980$
```

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.

**Name Requests**

If your program is doing NAMELIST input from the terminal, you can request the group name and NAMELIST names that it accepts.

To do so, enter a question mark (?) in column two and press Return. The group name and variable names are then displayed. The program then waits again for input.
Example: Requesting names:

```fortran
! 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% `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?
  $case
  $sample
  $new
  $delta
  $end
  $case sample="Test 2", delta=0.03 $  <-- Keyboard Input
demo%
CHAPTER 6

Intrinsic Functions

This chapter tabulates and explains the set of intrinsic functions that are part of Sun WorkShop FORTRAN 77. (For information about Fortran library routines, see the Fortran Library Reference.)

Intrinsic functions that are Sun extensions of the ANSI FORTRAN 77 standard are marked with ✠.

Intrinsic functions have generic and specific names when they accept arguments of more than one data type. In general, the generic name returns a value with the same data type as its argument. However, there are exceptions such as the type conversion functions (TABLE 6-2) and the inquiry functions (TABLE 6-7). The function may also be called by one of its specific names to handle a specific argument data type.

With functions that work on more than one data item (e.g. sign(a1,a2)), all the data arguments must be the same type.

In the following tables, the FORTRAN 77 intrinsic functions are listed by:

- Intrinsic Function – description of what the function does
- Definition – a mathematical definition
- No. of Args. – number of arguments the function accepts
- Generic Name – the function’s generic name
- Specific Names – the function’s specific names
- Argument Type – data type associated with each specific name
- Function Type – data type returned for specific argument data type

Note – Compiler options -dbl, -i2, -r8, and -xtypemap change the default sizes of variables and have an effect on intrinsic references. See “Remarks” on page 338, and the discussion of default sizes and alignment in “Size and Alignment of Data Types” on page 31.
Arithmetic and Mathematical Functions

This section details arithmetic, type conversion, trigonometric, and other functions. "a" stands for a function’s single argument, "a1" and "a2" for the first and second arguments of a two argument function, and "ar" and "ai" for the real and imaginary parts of a function’s complex argument.

Note that REAL*16 and COMPLEX*32 are SPARC only.
## Arithmetic

### TABLE 6-1 Arithmetic Functions

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>No. of Args</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute value</td>
<td>$</td>
<td>a</td>
<td>= (a_r^2 + a_i^2)^{1/2}$</td>
<td>1</td>
<td>ABS</td>
<td>IABS, ABS, DABS, CABS, QABS, ZABS, CDABS, CQABS</td>
</tr>
<tr>
<td>Truncation</td>
<td>int(a)</td>
<td>1</td>
<td>AINT</td>
<td>AINT, DINT, QINT</td>
<td>REAL, DOUBLE</td>
<td>REAL, DOUBLE, REAL*16</td>
</tr>
<tr>
<td>Nearest whole number</td>
<td>int(a+.5) if $a \geq 0$ int(a-.5) if $a &lt; 0$</td>
<td>1</td>
<td>ANINT</td>
<td>ANINT, DNINT, QNINT</td>
<td>REAL, DOUBLE</td>
<td>REAL, DOUBLE, REAL*16</td>
</tr>
<tr>
<td>Nearest integer</td>
<td>int(a+.5) if $a \geq 0$ int(a-.5) if $a &lt; 0$</td>
<td>1</td>
<td>NINT</td>
<td>NINT, IDNINT, IQNINT</td>
<td>REAL, DOUBLE</td>
<td>INTEGER, DOUBLE, REAL*16</td>
</tr>
<tr>
<td>Remainder</td>
<td>$a_1 - \text{int}(a_1/a_2)\times a_2$</td>
<td>2</td>
<td>MOD</td>
<td>MOD, AMOD, DMOD, QMOD</td>
<td>INTEGER, REAL, DOUBLE</td>
<td>INTEGER, REAL, DOUBLE, REAL*16</td>
</tr>
<tr>
<td>Transfer of sign</td>
<td>$</td>
<td>a_1</td>
<td>$ if $a_2 \geq 0$ $-</td>
<td>a_1</td>
<td>$ if $a_2 &lt; 0$</td>
<td>2</td>
</tr>
<tr>
<td>Positive difference</td>
<td>$a_1 - a_2$ if $a_1 &gt; a_2$ 0 if $a_1 \leq a_2$</td>
<td>2</td>
<td>DIM</td>
<td>IDIM, DIM, DDIM, QDIM</td>
<td>INTEGER, REAL, DOUBLE</td>
<td>INTEGER, REAL, DOUBLE, REAL*16</td>
</tr>
<tr>
<td>Double and quad products</td>
<td>$a_1 \times a_2$</td>
<td>2</td>
<td>-</td>
<td>DPROD, QPROD</td>
<td>REAL, DOUBLE</td>
<td>DOUBLE, REAL*16</td>
</tr>
<tr>
<td>Choosing largest value</td>
<td>$\max(a_1, a_2, \ldots)$</td>
<td>≥2</td>
<td>MAX</td>
<td>MAX0, AMAX1, DMAX1, QMAX1</td>
<td>INTEGER, REAL, DOUBLE</td>
<td>INTEGER, REAL, DOUBLE, REAL*16</td>
</tr>
</tbody>
</table>
### Table 6-1: Arithmetic Functions (Continued)

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>No. of Args.</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAX0</td>
<td>AMAX0</td>
<td>INTEGRER</td>
<td>REAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX1</td>
<td>MAX1</td>
<td>REAL</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN</td>
<td>MIN0</td>
<td>INTEGER</td>
<td>REAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMIN1</td>
<td>AMIN0</td>
<td>INTEGER</td>
<td>REAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMIN1</td>
<td>DMIN1</td>
<td>DOUBLE</td>
<td>REAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QMIN1</td>
<td>QMIN1</td>
<td>REAL*16</td>
<td>REAL*16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Choosing smallest value \( \min(a_1, a_2, \ldots) \) \( \geq 2 \)


## Type Conversion

**TABLE 6-2** Type Conversion Functions

<table>
<thead>
<tr>
<th>Conversion to</th>
<th>No. of Args</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>1</td>
<td>INT</td>
<td>-</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INT</td>
<td>REAL</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IFIX</td>
<td>REAL</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IDINT</td>
<td>DOUBLE</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>COMPLEX</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>COMPLEX*16</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>COMPLEX*32</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IQINT</td>
<td>REAL*16</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

| REAL          | 1           | REAL         | REAL           | INTEGER       | REAL          |
|               |             | REAL         | FLOAT          | INTEGER       | REAL          |
|               |             | -            | REAL           | REAL          | REAL          |
|               |             | SNGL         | DOUBLE         | REAL          | REAL          |
|               |             | SNGLQ        | REAL*16        | REAL          | REAL          |
|               |             | -            | COMPLEX        | REAL          | REAL          |
|               |             | -            | COMPLEX*16     | REAL          | REAL          |
|               |             | -            | COMPLEX*32     | REAL          | REAL          |

| DOUBLE        | 1           | DBLE         | DBLE           | INTEGER       | DOUBLE PRECISION |
|               |             | DREAL        | REAL           | DOUBLE        | DOUBLE PRECISION |
|               |             | -            | DOUBLE         | DOUBLE        | DOUBLE PRECISION |
|               |             | -            | COMPLEX        | DOUBLE        | DOUBLE PRECISION |
|               |             | -            | COMPLEX*16     | DOUBLE        | DOUBLE PRECISION |
|               |             | -            | REAL*16        | DOUBLE        | DOUBLE PRECISION |
|               |             | -            | COMPLEX*32     | DOUBLE        | DOUBLE PRECISION |
|               |             | DBLEQ        | REAL*16        | DOUBLE        | DOUBLE PRECISION |
|               |             | -            | COMPLEX*32     | DOUBLE        | DOUBLE PRECISION |

| REAL*16       | 1           | QREAL        | QREAL          | INTEGER       | REAL*16       |
|               |             | QREAL        | QREAL          | INTEGER       | REAL*16       |
|               |             | -            | REAL           | REAL*16       | REAL*16       |
|               |             | QEXT         | INTEGER        | REAL*16       | REAL*16       |
|               |             | -            | REAL           | REAL*16       | REAL*16       |
|               |             | QEXTD        | DOUBLE         | REAL*16       | REAL*16       |
|               |             | -            | REAL*16        | REAL*16       | REAL*16       |
|               |             | -            | COMPLEX        | REAL*16       | REAL*16       |
|               |             | -            | COMPLEX*16     | REAL*16       | REAL*16       |
|               |             | -            | COMPLEX*32     | REAL*16       | REAL*16       |
### TABLE 6-2 Type Conversion Functions (Continued)

<table>
<thead>
<tr>
<th>Conversion to</th>
<th>No. of Args</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX</td>
<td>1 or 2</td>
<td>CMPLX</td>
<td></td>
<td>INTEGER</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX*16</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX*32</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>1 or 2</td>
<td>DCMPLX*</td>
<td></td>
<td>INTEGER</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td>COMPLEX</td>
<td></td>
<td></td>
<td></td>
<td>REAL</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX*16</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX*32</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td>COMPLEX*32</td>
<td>1 or 2</td>
<td>QCMPLX*</td>
<td></td>
<td>INTEGER</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX*16</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td>INTEGER</td>
<td>1</td>
<td>–</td>
<td>ICHAR</td>
<td>CHARACTER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
<td>IACHAR *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td>1</td>
<td>–</td>
<td>CHAR</td>
<td>INTEGER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>–</td>
<td>ACHAR *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On an ASCII platform, including Sun systems:
- ACHAR is a nonstandard synonym for CHAR
- IACHAR is a nonstandard synonym for ICHAR

On a non-ASCII platform, ACHAR and IACHAR were intended to provide a way to deal directly with ASCII.
## Trigonometric Functions

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>Args</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sine</strong>&lt;br&gt;See Note (7).</td>
<td>\sin(a)</td>
<td>1</td>
<td>SIN</td>
<td>SIN</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DSIN</td>
<td>DSIN</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QSIN</td>
<td>QSIN</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CSIN</td>
<td>CSIN</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ZSIN</td>
<td>ZSIN</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CDSIN</td>
<td>CDSIN</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CQSIN</td>
<td>CQSIN</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(clazz)</td>
<td>(clazz)</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td><strong>Sine</strong>&lt;br&gt;(degrees)&lt;br&gt;See Note (7).</td>
<td>\sin(a)</td>
<td>1</td>
<td>SIND</td>
<td>SIND</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DSIND</td>
<td>DSIND</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QSIND</td>
<td>QSIND</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td><strong>Cosine</strong>&lt;br&gt;See Note (7).</td>
<td>\cos(a)</td>
<td>1</td>
<td>COS</td>
<td>COS</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DCOS</td>
<td>DCOS</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QCOS</td>
<td>QCOS</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CCOS</td>
<td>CCOS</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ZCOS</td>
<td>ZCOS</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CDCOS</td>
<td>CDCOS</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CQCOS</td>
<td>CQCOS</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(clazz)</td>
<td>(clazz)</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td><strong>Cosine</strong>&lt;br&gt;(degrees)&lt;br&gt;See Note (7).</td>
<td>\cos(a)</td>
<td>1</td>
<td>COSD</td>
<td>COSD</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DCOSD</td>
<td>DCOSD</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QCOSD</td>
<td>QCOSD</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td><strong>Tangent</strong>&lt;br&gt;See Note (7).</td>
<td>\tan(a)</td>
<td>1</td>
<td>TAN</td>
<td>TAN</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DTAN</td>
<td>DTAN</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QTAN</td>
<td>QTAN</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td><strong>Tangent</strong>&lt;br&gt;(degrees)&lt;br&gt;See Note (7).</td>
<td>\tan(a)</td>
<td>1</td>
<td>TAND</td>
<td>TAND</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DTAND</td>
<td>DTAND</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QTAND</td>
<td>QTAND</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td><strong>Arcsine</strong>&lt;br&gt;See Note (7).</td>
<td>\arcsin(a)</td>
<td>1</td>
<td>ASIN</td>
<td>ASIN</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DASIN</td>
<td>DASIN</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QASIN</td>
<td>QASIN</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td><strong>Arcsine</strong>&lt;br&gt;(degrees)&lt;br&gt;See Note (7).</td>
<td>\arcsin(a)</td>
<td>1</td>
<td>ASIND</td>
<td>ASIND</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DASIND</td>
<td>DASIND</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QASIND</td>
<td>QASIND</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td><strong>Arccosine</strong>&lt;br&gt;See Note (7).</td>
<td>\arccos(a)</td>
<td>1</td>
<td>ACOS</td>
<td>ACOS</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DACOS</td>
<td>DACOS</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QACOS</td>
<td>QACOS</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Intrinsic Function</td>
<td>Definition</td>
<td>Args</td>
<td>Generic Name</td>
<td>Specific Names</td>
<td>Argument Type</td>
<td>Function Type</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------</td>
<td>------</td>
<td>--------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Arccosine (degrees)</td>
<td>$\text{arccos}(a)$</td>
<td>1</td>
<td>ACOSD *</td>
<td>DACOSD *</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QACOSD *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Arctangent (degrees)</td>
<td>$\text{arctan}(a)$</td>
<td>1</td>
<td>ATAN</td>
<td>DATAN</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QATAN *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Arctangent (a1/a2)</td>
<td>$\text{arctan}(a1/a2)$</td>
<td>2</td>
<td>ATAN2</td>
<td>DATAN2</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QATAN2 *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Arctangent (degrees)</td>
<td>$\text{arctan}(a)$</td>
<td>1</td>
<td>ATAND *</td>
<td>DATAND *</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QATAND *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Arctangent (a1/a2)</td>
<td>$\text{arctan2d}(a1/a2)$</td>
<td>2</td>
<td>ATAN2D *</td>
<td>DATAN2D *</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QATAN2D *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Hyperbolic Sine</td>
<td>$\text{sinh}(a)$</td>
<td>1</td>
<td>SINH</td>
<td>DSINH</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QSINH *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Hyperbolic Cosine</td>
<td>$\text{cosh}(a)$</td>
<td>1</td>
<td>COSH</td>
<td>DCOSH</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QCOSH *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Hyperbolic Tangent</td>
<td>$\text{tanh}(a)$</td>
<td>1</td>
<td>TANH</td>
<td>DTANH</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QTANH *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
</tbody>
</table>
### Other Mathematical Functions

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>No. of Args.</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imaginary part of a complex number</td>
<td>$ai$</td>
<td>1</td>
<td>IMAG</td>
<td>AIMAG</td>
<td>COMPLEX</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DIMAG</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QIMAG</td>
<td>COMPLEX*32</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Conjugate of a complex number</td>
<td>$(ar, -ai)$</td>
<td>1</td>
<td>CONJG</td>
<td>CONJG</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DCONJG</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QCONJG</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td>Square root</td>
<td>$a**(1/2)$</td>
<td>1</td>
<td>SQRT</td>
<td>SQRT</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DSQRT</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QSQRT</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CSQRT</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ZSQRT</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CDSQRT</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CQSQRT</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td>Cube root</td>
<td>$a**(1/3)$</td>
<td>1</td>
<td>CBRT</td>
<td>CBRT</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DCBRT</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QCBRT</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CCBRT</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ZCBRT</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CDCBRT</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CQCBRT</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td>Exponential</td>
<td>$e^{a}$</td>
<td>1</td>
<td>EXP</td>
<td>EXP</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DEXP</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QEXP</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CEXP</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ZEXP</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CDEXP</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CQEXP</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
<tr>
<td>Natural logarithm</td>
<td>$\log(a)$</td>
<td>1</td>
<td>LOG</td>
<td>ALOG</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DLOG</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QLOG</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CLOG</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ZLOG</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CDLOG</td>
<td>DOUBLE COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CQLOG</td>
<td>COMPLEX*32</td>
<td>COMPLEX*32</td>
</tr>
</tbody>
</table>
The error function: \( \frac{2}{\sqrt{\pi}} \times \text{integral from 0 to a of } e^{-t^2} \, dt \)

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>No. of Args.</th>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common logarithm</td>
<td>( \log_{10}(a) )</td>
<td>1</td>
<td>LOG10</td>
<td>ALOG10</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DLOG10</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>QLOG10 *</td>
<td>REAL*16</td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Error function</td>
<td>( \text{erf}(a) )</td>
<td>1</td>
<td>ERF</td>
<td>ERF *</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>(See note below)</td>
<td></td>
<td></td>
<td>DERF *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>Error function</td>
<td>( 1.0 - \text{erf}(a) )</td>
<td>1</td>
<td>ERFC</td>
<td>ERFC *</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DERFC *</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Table 6-4 Other Mathematical Functions (Continued)
Character Functions

Table 6-5 Character Functions

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>No. of Args</th>
<th>Specific Names</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion</td>
<td>Conversion to character</td>
<td>1</td>
<td>CHAR</td>
<td>INTEGER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>Conversion to integer</td>
<td>1</td>
<td>ACHAR *</td>
<td>INTEGER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>Conversion to integer</td>
<td>1</td>
<td>ICHAR *</td>
<td>INTEGER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>See also: TABLE 6-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index of a substring</td>
<td>Location of substring a2 in string a1</td>
<td>2</td>
<td>INDEX</td>
<td>CHARACTER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>See Note (10).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td>Length of character entity</td>
<td>1</td>
<td>LEN</td>
<td>CHARACTER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>See Note (11).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lexically greater</td>
<td>a1 ≥ a2</td>
<td>2</td>
<td>LGE</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>than or equal</td>
<td>See Note (12).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lexically greater</td>
<td>a1 &gt; a2</td>
<td>2</td>
<td>LGT</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>than</td>
<td>See Note (12).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lexically less than</td>
<td>a1 ≤ a2</td>
<td>2</td>
<td>LLE</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>or equal</td>
<td>See Note (12).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lexically less than</td>
<td>a1 &lt; a2</td>
<td>2</td>
<td>LLT</td>
<td>CHARACTER</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>than</td>
<td>See Note (12).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On an ASCII platform (including Sun systems):

- ACHAR is a nonstandard synonym for CHAR
- ICHAR is a nonstandard synonym for ICHAR

On a non-ASCII platform, ACHAR and ICHAR were intended to provide a way to deal directly with ASCII.
Miscellaneous Functions

Other miscellaneous functions include bitwise functions, environmental inquiry functions, and memory allocation and deallocation functions.

Bit Manipulation *

None of these functions are part of the FORTRAN 77 Standard.

<table>
<thead>
<tr>
<th>Bitwise Operations</th>
<th>No. of Args.</th>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complement</td>
<td>1</td>
<td>NOT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>And</td>
<td>2</td>
<td>AND</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>IAND</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Inclusive or</td>
<td>2</td>
<td>OR</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>IOR</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Exclusive or</td>
<td>2</td>
<td>XOR</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>IEOR</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Shift</td>
<td>2</td>
<td>ISHFT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Left shift</td>
<td>2</td>
<td>LSHIFT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Right shift</td>
<td>2</td>
<td>RSHIFT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Logical right shift</td>
<td>2</td>
<td>LRSHFT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Circular shift</td>
<td>3</td>
<td>ISHFTC</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Bit extraction</td>
<td>3</td>
<td>IBITS</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Bit set</td>
<td>2</td>
<td>IBSET</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Bit test</td>
<td>2</td>
<td>BTEST</td>
<td>INTEGER</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>Bit clear</td>
<td>2</td>
<td>IBCLR</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
The above functions are available as intrinsic or extrinsic functions. See also the discussion of the library bit manipulation routines in the Fortran Library Reference manual.

Environmental Inquiry Functions *

None of these functions are part of the FORTRAN 77 Standard.

**TABLE 6-7** Environmental Inquiry Functions

<table>
<thead>
<tr>
<th>Definition</th>
<th>No. of Args.</th>
<th>Generic Name</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base of Number System</td>
<td>1</td>
<td>EPBASE</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Number of Significant Bits</td>
<td>1</td>
<td>EPPREC</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Minimum Exponent</td>
<td>1</td>
<td>EPEMIN</td>
<td>REAL</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Maximum Exponent</td>
<td>1</td>
<td>EPEMAX</td>
<td>REAL</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Least Nonzero Number</td>
<td>1</td>
<td>EPTINY</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Largest Number Representable</td>
<td>1</td>
<td>EPHUGE</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>Epsilon</td>
<td>1</td>
<td>EPMRSP</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
</tbody>
</table>
Memory *

None of these functions are part of the FORTRAN 77 Standard.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Definition</th>
<th>No. of Args</th>
<th>Specific Name</th>
<th>Argument Type</th>
<th>Function Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Address of See Note (17).</td>
<td>1</td>
<td>LOC</td>
<td>Any</td>
<td>INTEGER<em>4, INTEGER</em>8</td>
</tr>
<tr>
<td>Allocate</td>
<td>Allocate memory and return address. See Note (17).</td>
<td>1</td>
<td>MALLOC, MALLOC64</td>
<td>INTEGER<em>4, INTEGER</em>8</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Deallocate</td>
<td>Deallocate memory allocated by MALLOC. See Note (17).</td>
<td>1</td>
<td>FREE</td>
<td>Any</td>
<td>-</td>
</tr>
<tr>
<td>Size</td>
<td>Return the size of the argument in bytes. See Note (18).</td>
<td>1</td>
<td>SIZEOF</td>
<td>Any expression</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Although malloc(3F) and free(3F) are not, strictly speaking, intrinsics, they are listed here and in the Fortran Library Reference. Additional non-standard intrinsics, such as isetjmp(3F), longjmp(3F), and date_and_time(3F), are also described in the Fortran Library Reference and by their man pages.

Remarks

The following remarks apply to all of the intrinsic function tables in this chapter.

- The abbreviation DOUBLE stands for DOUBLE PRECISION.
- An intrinsic that takes INTEGER arguments accepts INTEGER*2, INTEGER*4, or INTEGER*8.
- INTEGER intrinsics that take INTEGER arguments return values of INTEGER type determined as follows – note that options -i2, -dbl, and -xtypemap may alter the default sizes of actual arguments:
  - mod sign dim max min and iand or ior xor ieor — size of the value returned is the largest of the sizes of the arguments.
  - abs ishft lshift rshift lrshft ibset btest ivclr ishftc ibits — size of the value returned is the size of the first argument.
Chapter 6 \ Intrinsic Functions

■ int ebase eprec — size of the value returned is the size of default INTEGER.
■ ephuge — size of the value returned is the size of the default INTEGER, or the size of the argument, whichever is largest.

Options that change the default data sizes (see “Size and Alignment of Data Types” on page 31) also change the way some intrinsics are used. For example, with -dbl in effect, a call to ZCOS with a DOUBLE COMPLEX argument will automatically become a call to CQCOS because the argument has been promoted to COMPLEX*32. The following functions have this capability:

aimag alog amod cabs ccbtrt ccos cdabs cdcbtrt cdcos cdexp cdlog cdsin cdsqrt cexp clog csin cssqrt dabs dacos dacosd dasin dasind datan datand dcbtrt dconjg dcos dcosd dcosh ddim derf derfc dexp dimag dint dlog dmod dnint dprod dsign dsin dsind dsinh dsqrt dtan dtand dtanh idnnt iidnnt jidnnt zabs zcbrt zcos zexp zlog zsin zsqrt

■ The following functions permit arguments of an integer or logical type of any size:

and iand ieor iiand iieor iior inot ior jiand jieor jior jnot lrshft lshift not or rshift xor

■ (SPARC only) An intrinsic that is shown to return a default REAL, DOUBLE PRECISION, COMPLEX, or DOUBLE COMPLEX value will return the prevailing type depending on certain compilation options. (See “Size and Alignment of Data Types” on page 31.) For example, if compiled with -xtypemap=real:64, double:64:

■ A call to a REAL function returns REAL*8
■ A call to a DOUBLE PRECISION function returns REAL*8
■ A call to a COMPLEX function returns COMPLEX*16
■ A call to a DOUBLE COMPLEX function returns COMPLEX*16

Other options that alter the data sizes of default data types are -r8 and -dbl, which also promote DOUBLE to QUAD. The -xtypemap= option provides more flexibility than these older compiler options and is preferred.

■ 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, the absolute value of a complex argument, and others. 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 FORTRAN extensions added.

(1) INT

If $\lambda$ is type integer, then $\text{INT}(\lambda)$ is $\lambda$.  
If $\lambda$ is type real or double precision, then:

if $|\lambda| < 1$, then $\text{INT}(\lambda)$ is 0
if $|\lambda| \geq 1$, then $\text{INT}(\lambda)$ is the greatest integer that does not exceed the magnitude of $\lambda$, and whose sign is the same as the sign of $\lambda$. (Such a mathematical integer value may be too large to fit in the computer integer type.)

If $\lambda$ is type complex or double complex, then apply the above rule to the real part of $\lambda$.

If $\lambda$ is type real, then $\text{IFIX}(\lambda)$ is the same as $\text{INT}(\lambda)$.

(2) REAL

If $\lambda$ is type real, then $\text{REAL}(\lambda)$ is $\lambda$.

If $\lambda$ is type integer or double precision, then $\text{REAL}(\lambda)$ is as much precision of the significant part of $\lambda$ as a real datum can contain.

If $\lambda$ is type complex, then $\text{REAL}(\lambda)$ is the real part of $\lambda$.

If $\lambda$ is type double complex, then $\text{REAL}(\lambda)$ is as much precision of the significant part of the real part of $\lambda$ as a real datum can contain.

(3) DBLE

If $\lambda$ is type double precision, then $\text{DBLE}(\lambda)$ is $\lambda$.

If $\lambda$ is type integer or real, then $\text{DBLE}(\lambda)$ is as much precision of the significant part of $\lambda$ as a double precision datum can contain.

If $\lambda$ is type complex, then $\text{DBLE}(\lambda)$ is as much precision of the significant part of the real part of $\lambda$ as a double precision datum can contain.

If $\lambda$ is type COMPLEX*16, then $\text{DBLE}(\lambda)$ is the real part of $\lambda$.

(3') QREAL

If $\lambda$ is type REAL*16, then $\text{QREAL}(\lambda)$ is $\lambda$. 
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 or COMPLEX*32, 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, COMPLEX*16, or COMPLEX*32.

(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, COMPLEX*16, or COMPLEX*32.

(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
■ \text{CHAR}(\text{ICHAR}(C)) = C, for any character \( C \) capable of representation in the processor

(6) \text{COMPLEX}

A \text{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) \text{COMPLEX Function}

The result of a function of type \text{COMPLEX} is the principal value.

(8') \text{CBRT}

If \( a \) is of \text{COMPLEX} type, \text{CBRT} results in \text{COMPLEX} \( RT1=(A, B) \), where: \( A \geq 0.0 \), and \(-60 \text{ degrees} \leq \arctan (B/A) < +60 \text{ degrees}\).

Other two possible results can be evaluated as follows:

\begin{itemize}
  \item \( RT2 = RT1 \times (-0.5, \text{square\_root}\ (0.75)) \)
  \item \( RT3 = RT1 \times (-0.5, \text{square\_root}\ (0.75)) \)
\end{itemize}

(9) Argument types

All arguments in an intrinsic function reference must be of the same type.

(10) \text{INDEX}

\text{INDEX}(X, Y) \text{ is the place in } X \text{ where } Y \text{ starts. That is, it is the starting position within character string } X \text{ of the first occurrence of character string } Y.

If \( Y \) does not occur in \( X \), then \text{INDEX}(X, Y) \text{ is 0.}

If \( \text{LEN}(X) < \text{LEN}(Y) \), then \text{INDEX}(X, Y) \text{ is 0.}

\text{INDEX} \text{ returns default INTEGRER*4 data. If compiling for a 64-bit environment, the compiler will issue a warning if the result overflows the INTEGRER*4 data range. To use \text{INDEX} in a 64-bit environment with character strings larger than the INTEGRER*4 limit (2 Gbytes), the \text{INDEX} function and the variables receiving the result must be declared INTEGRER*8.}

(11) \text{LEN}

\text{LEN} \text{ returns the declared length of the CHARACTER argument variable. The actual value of the argument is of no importance.}
LEN returns default INTEGER*4 data. If compiling for a 64-bit environment, the compiler will issue a warning if the result overflows the INTEGER*4 data range. To use LEN in a 64-bit environment with character variables larger than the INTEGER*4 limit (2 Gbytes), the LEN function and the variables receiving the result must be declared INTEGER*8.

(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 “VMS Language Extensions” on page 371 for details on other bitwise operations.

(14) Shift

LSHIFT shifts a1 logically left by a2 bits (inline code).

LRSHIFT shifts a1 logically right by a2 bits (inline code).

RSHIFT shifts a1 arithmetically right by a2 bits.

ISHIFT shifts a1 logically left if a2 > 0 and right if a2 < 0.

The LSHIFT and RSHIFT functions are the FORTRAN analogs of the C << and >> operators. As in C, the semantics depend on the hardware.

The behavior of the shift functions with an out of range shift count is hardware dependent and generally unpredictable. In this release, shift counts larger than 31 result in hardware dependent behavior.

(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, MALLOC, and FREE

The LOC function returns the 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 address of that block.

LOC returns default INTEGER*4 in 32-bit environments, INTEGER*8 in 64-bit environments.

MALLOC is a library function and not an intrinsic. It too returns default INTEGER*4 in 32-bit environments, INTEGER*8 in 64-bit environments. However, MALLOC must be explicitly declared INTEGER*8 when compiling for 64-bit environments.

The value returned by LOC or MALLOC should be stored in variables typed POINTER, INTEGER*4, or INTEGER*8 in 64-bit environments. The argument to FREE must be the value returned by a previous call to MALLOC and hence should have data type POINTER, INTEGER*4, or INTEGER*8.

MALLOC64 always takes an INTEGER*8 argument (size of memory request in bytes) and always returns an INTEGER*8 value. Use this routine rather than MALLOC when compiling programs that must run in both 32-bit and 64-bit environments. The receiving variable must be declared either POINTER or INTEGER*8.

(18) SIZEOF

The SIZEOF intrinsic cannot be applied to arrays of an assumed size, characters of a length that is passed, or subroutine calls or names. SIZEOF returns default INTEGER*4 data. If compiling for a 64-bit environment, the compiler will issue a warning if the result overflows the INTEGER*4 data range. To use SIZEOF in a 64-bit environment with arrays larger than the INTEGER*4 limit (2 Gbytes), the SIZEOF function and the variables receiving the result must be declared INTEGER*8.

* VMS Intrinsic Functions

This section lists VMS FORTRAN intrinsic routines recognized by $F77$. They are, of course, nonstandard. *
VMS Double-Precision Complex

### TABLE 6-9  VMS Double-Precision Complex Functions

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDABS</td>
<td></td>
<td>Absolute value</td>
<td>COMPLEX*16</td>
<td>REAL*8</td>
</tr>
<tr>
<td>CDEXP</td>
<td></td>
<td>Exponential, e**a</td>
<td>COMPLEX*16</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>CDLOG</td>
<td></td>
<td>Natural log</td>
<td>COMPLEX*16</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>CDSQRT</td>
<td></td>
<td>Square root</td>
<td>COMPLEX*16</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>CDSIN</td>
<td></td>
<td>Sine</td>
<td>COMPLEX*16</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>CDCOS</td>
<td></td>
<td>Cosine</td>
<td>COMPLEX*16</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>DCMPLX</td>
<td></td>
<td>Convert to DOUBLE COMPLEX</td>
<td>Any numeric</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>DCONJG</td>
<td></td>
<td>Complex conjugate</td>
<td>COMPLEX*16</td>
<td>COMPLEX*16</td>
</tr>
<tr>
<td>DIMAG</td>
<td></td>
<td>Imaginary part of complex</td>
<td>COMPLEX*16</td>
<td>REAL*8</td>
</tr>
<tr>
<td>DREAL</td>
<td></td>
<td>Real part of complex</td>
<td>COMPLEX*16</td>
<td>REAL*8</td>
</tr>
</tbody>
</table>

VMS Degree-Based Trigonometric

### TABLE 6-10  VMS Degree-Based Trigonometric Functions

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIND</td>
<td></td>
<td>Sine</td>
<td>REAL*4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>SIND</td>
<td></td>
<td></td>
<td>REAL*8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>QSIND</td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>COSD</td>
<td></td>
<td>Cosine</td>
<td>REAL*4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>COSD</td>
<td></td>
<td></td>
<td>REAL*8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>QCOSD</td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>TAND</td>
<td></td>
<td>Tangent</td>
<td>REAL*4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>TAND</td>
<td></td>
<td></td>
<td>REAL*8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>QTAND</td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
<tr>
<td>ASIND</td>
<td></td>
<td>Arc sine</td>
<td>REAL*4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>ASIND</td>
<td></td>
<td></td>
<td>REAL*8</td>
<td>REAL*8</td>
</tr>
<tr>
<td>QASIND</td>
<td></td>
<td></td>
<td>REAL*16</td>
<td>REAL*16</td>
</tr>
</tbody>
</table>
TABLE 6-11  VMS Bit-Manipulation Functions

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBITS</td>
<td>IIBITS</td>
<td>From a1, initial bit a2, extract a3 bits</td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td>JIBITS</td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>ISHFT</td>
<td>IISHFT</td>
<td>Shift a1 logically by a2 bits; if a2 positive shift left, if a2 negative shift right</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JISHFT</td>
<td></td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>ISHFTC</td>
<td>IISHFTC</td>
<td>In a1, circular shift by a2 places, of right a3 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JISHFTC</td>
<td></td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IAND</td>
<td>IIAND</td>
<td>Bitwise AND of a1, a2</td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td>JIAND</td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IOR</td>
<td>IIOR</td>
<td>Bitwise OR of a1, a2</td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td>JIOR</td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
</tbody>
</table>
### TABLE 6-11 VMS Bit-Manipulation Functions (Continued)

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIEOR</td>
<td>IIEOR</td>
<td>Bitwise exclusive OR of a1, a2</td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td>JIEOR</td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>NOT</td>
<td>INOT</td>
<td>Bitwise complement</td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td>JNOT</td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IBSET</td>
<td>IIBSET</td>
<td>In a1, set bit a2 to 1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>JIBSET</td>
<td>In a1, set bit a2 to 1; return new a1</td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In a1, set bit a2 to 1; return new a1</td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>BTEST</td>
<td>BITEST</td>
<td>If bit a2 of a1 is 1, return .TRUE.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>BJTEST</td>
<td></td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IBCLR</td>
<td>IIBCLR</td>
<td>In a1, set bit a2 to 0; return new a1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>JIBCLR</td>
<td></td>
<td>INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
</tbody>
</table>
VMS Multiple Integer Types

The possibility of multiple integer types is not addressed by the FORTRAN Standard. $\texttt{f77}$ copes with their existence by treating a specific \texttt{INTEGER}-to-\texttt{INTEGER} function name (\texttt{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>
<thead>
<tr>
<th>TABLE 6-12</th>
<th>VMS Integer Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific Names</td>
<td>Function</td>
</tr>
<tr>
<td>$\texttt{IIABS}$</td>
<td>Absolute value</td>
</tr>
<tr>
<td>$\texttt{JIABS}$</td>
<td>Absolute value</td>
</tr>
<tr>
<td>$\texttt{IMAX0}$</td>
<td>Maximum $^1$</td>
</tr>
<tr>
<td>$\texttt{JMAX0}$</td>
<td>Maximum $^1$</td>
</tr>
<tr>
<td>$\texttt{IMIN0}$</td>
<td>Minimum $^1$</td>
</tr>
<tr>
<td>$\texttt{JMIN0}$</td>
<td>Minimum $^1$</td>
</tr>
<tr>
<td>$\texttt{IIDIM}$</td>
<td>Positive difference $^2$</td>
</tr>
<tr>
<td>$\texttt{JIDIM}$</td>
<td>Positive difference $^2$</td>
</tr>
<tr>
<td>$\texttt{IMOD}$</td>
<td>Remainder of $a1/a2$</td>
</tr>
<tr>
<td>$\texttt{JMOD}$</td>
<td>Remainder of $a1/a2$</td>
</tr>
<tr>
<td>$\texttt{IISIGN}$</td>
<td>Transfer of sign, $</td>
</tr>
<tr>
<td>$\texttt{JISIGN}$</td>
<td>Transfer of sign, $</td>
</tr>
</tbody>
</table>

1. There must be at least two arguments.
2. The positive difference is: $a1-\min(a1,a2)$
Functions Coerced to a Particular Type

Some VMS FORTRAN functions coerce to a particular INTEGER type.

<table>
<thead>
<tr>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IINT</td>
<td>Truncation toward zero</td>
<td>REAL*4</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JINT</td>
<td>Truncation toward zero</td>
<td>REAL*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>LINT</td>
<td>Truncation toward zero</td>
<td>REAL*4</td>
<td>INTEGER*8</td>
</tr>
<tr>
<td>IIDINT</td>
<td>Truncation toward zero</td>
<td>REAL*8</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JIDINT</td>
<td>Truncation toward zero</td>
<td>REAL*8</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IIQINT</td>
<td>Truncation toward zero</td>
<td>REAL*16</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JIQINT</td>
<td>Truncation toward zero</td>
<td>REAL*16</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>ININT</td>
<td>Nearest integer, INT(a+.5*sign(a))</td>
<td>REAL*4</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JNINT</td>
<td>Nearest integer, INT(a+.5*sign(a))</td>
<td>REAL*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IIDNNT</td>
<td>Nearest integer, INT(a+.5*sign(a))</td>
<td>REAL*8</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JIDNNT</td>
<td>Nearest integer, INT(a+.5*sign(a))</td>
<td>REAL*8</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IIQNNT</td>
<td>Nearest integer, INT(a+.5*sign(a))</td>
<td>REAL*16</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JIQNNT</td>
<td>Nearest integer, INT(a+.5*sign(a))</td>
<td>REAL*16</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IIFIX</td>
<td>Fix</td>
<td>REAL*4</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JIFIX</td>
<td>Fix</td>
<td>REAL*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IMAX1(a,a2,...)</td>
<td>Maximum of two or more arguments</td>
<td>REAL*4</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JMAX1(a,a2,...)</td>
<td>Maximum of two or more arguments</td>
<td>REAL*4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>IMIN1(a,a2,...)</td>
<td>Minimum of two or more arguments</td>
<td>READ*4</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JMIN1(a,a2,...)</td>
<td>Minimum of two or more arguments</td>
<td>READ*4</td>
<td>INTEGER*4</td>
</tr>
</tbody>
</table>
Functions Translated to a Generic Name

In some cases, each VMS-specific name is translated into an f77 generic name.

### TABLE 6-14  VMS Functions That Are Translated into f77 Generic Names

<table>
<thead>
<tr>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOATI</td>
<td>Convert to REAL*4</td>
<td>INTEGER*2</td>
<td>REAL*4</td>
</tr>
<tr>
<td>FLOATJ</td>
<td>Convert to REAL*4</td>
<td>INTEGER*4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>DFLOTI</td>
<td>Convert to REAL*8</td>
<td>INTEGER*2</td>
<td>REAL*8</td>
</tr>
<tr>
<td>DFLOTJ</td>
<td>Convert to REAL*8</td>
<td>INTEGER*4</td>
<td>REAL*8</td>
</tr>
<tr>
<td>AIMAX0</td>
<td>Maximum</td>
<td>INTEGER*2</td>
<td>REAL*4</td>
</tr>
<tr>
<td>AJMAX0</td>
<td>Maximum</td>
<td>INTEGER*4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>AIMIN0</td>
<td>Minimum</td>
<td>INTEGER*2</td>
<td>REAL*4</td>
</tr>
<tr>
<td>AJMIN0</td>
<td>Minimum</td>
<td>INTEGER*4</td>
<td>REAL*4</td>
</tr>
</tbody>
</table>

### 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 6-15  Zero-Extend Functions

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>Specific Names</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEXT</td>
<td>ZEXT</td>
<td>Zero-extend</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>IZEXT</td>
<td>IZEXT</td>
<td>Zero-extend</td>
<td>BYTE LOGICAL<em>1 LOGICAL</em>2 INTEGER*2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>JZEXT</td>
<td>JZEXT</td>
<td>Zero-extend</td>
<td>BYTE LOGICAL<em>1 LOGICAL</em>2 LOGICAL<em>4 INTEGER INTEGER</em>2 INTEGER*4</td>
<td>INTEGER*4</td>
</tr>
</tbody>
</table>
APPENDIX A

ASCII Character Set
This appendix contains two tables: ASCII character sets and control characters.

### TABLE A-1  ASCII Character Set

<table>
<thead>
<tr>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Name</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Name</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Name</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>NUL</td>
<td>32</td>
<td>040</td>
<td>20</td>
<td>SP</td>
<td>64</td>
<td>100</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>140</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>01</td>
<td>SOH</td>
<td>33</td>
<td>041</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>101</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>141</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>02</td>
<td>STX</td>
<td>34</td>
<td>042</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>102</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>142</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>03</td>
<td>ETX</td>
<td>35</td>
<td>043</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>103</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>143</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>04</td>
<td>EOT</td>
<td>36</td>
<td>044</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>104</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>144</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>05</td>
<td>ENQ</td>
<td>37</td>
<td>045</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>105</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>145</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>06</td>
<td>ACK</td>
<td>38</td>
<td>046</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>106</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>146</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>07</td>
<td>BEL</td>
<td>39</td>
<td>047</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>107</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>147</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>08</td>
<td>BS</td>
<td>40</td>
<td>050</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>110</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>150</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>09</td>
<td>HT</td>
<td>41</td>
<td>051</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>111</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>151</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>0A</td>
<td>LF</td>
<td>42</td>
<td>052</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>112</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>152</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>0B</td>
<td>VT</td>
<td>43</td>
<td>053</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>113</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>153</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>0C</td>
<td>FF</td>
<td>44</td>
<td>054</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>114</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>154</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>0D</td>
<td>CR</td>
<td>45</td>
<td>055</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>115</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>155</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>0E</td>
<td>SO</td>
<td>46</td>
<td>056</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>116</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>156</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>0F</td>
<td>SI</td>
<td>47</td>
<td>057</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>117</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>157</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>10</td>
<td>DLE</td>
<td>48</td>
<td>060</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>120</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>160</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>11</td>
<td>DC1</td>
<td>49</td>
<td>061</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>121</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>161</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>12</td>
<td>DC2</td>
<td>50</td>
<td>062</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>122</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>162</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>13</td>
<td>DC3</td>
<td>51</td>
<td>063</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>123</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>163</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>14</td>
<td>DC4</td>
<td>52</td>
<td>064</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>124</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>164</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>15</td>
<td>NAK</td>
<td>53</td>
<td>065</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>125</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>165</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>16</td>
<td>SYN</td>
<td>54</td>
<td>066</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>126</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>166</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>17</td>
<td>ETB</td>
<td>55</td>
<td>067</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>127</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>167</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>18</td>
<td>CAN</td>
<td>56</td>
<td>070</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>130</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>170</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>19</td>
<td>EM</td>
<td>57</td>
<td>071</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>131</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>171</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>1A</td>
<td>SUB</td>
<td>58</td>
<td>072</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>132</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>172</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>1B</td>
<td>ESC</td>
<td>59</td>
<td>073</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>133</td>
<td>5B</td>
<td>{</td>
<td>123</td>
<td>173</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>1C</td>
<td>FS</td>
<td>60</td>
<td>074</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>134</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>174</td>
<td>7C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>1D</td>
<td>GS</td>
<td>61</td>
<td>075</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>135</td>
<td>5D</td>
<td>}</td>
<td>125</td>
<td>175</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>1E</td>
<td>RS</td>
<td>62</td>
<td>076</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>136</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>176</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>1F</td>
<td>US</td>
<td>63</td>
<td>077</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>137</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>177</td>
<td>7F</td>
<td>DEL</td>
</tr>
<tr>
<td>Dec</td>
<td>Oct</td>
<td>Hex</td>
<td>Name</td>
<td>Keys</td>
<td>Meaning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>NUL</td>
<td>^P</td>
<td>Null or time fill character</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
<td>SOH</td>
<td>^A</td>
<td>Start of heading</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>02</td>
<td>STX</td>
<td>^B</td>
<td>Start of text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>03</td>
<td>ETX</td>
<td>^C</td>
<td>End of text (EOM)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>04</td>
<td>EOT</td>
<td>^D</td>
<td>End of transmission</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>05</td>
<td>ENQ</td>
<td>^E</td>
<td>Enquiry (WRU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>06</td>
<td>ACK</td>
<td>^F</td>
<td>Acknowledge (RU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>07</td>
<td>BEL</td>
<td>^G</td>
<td>Bell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>08</td>
<td>BS</td>
<td>^H</td>
<td>Backspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>09</td>
<td>HT</td>
<td>^I</td>
<td>Horizontal tab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>012</td>
<td>0A</td>
<td>LF</td>
<td>^J</td>
<td>Line feed (newline)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>013</td>
<td>0B</td>
<td>VT</td>
<td>^K</td>
<td>Vertical tab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>014</td>
<td>0C</td>
<td>FF</td>
<td>^L</td>
<td>Form Feed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>015</td>
<td>0D</td>
<td>CR</td>
<td>^M</td>
<td>Carriage Return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>016</td>
<td>0E</td>
<td>SO</td>
<td>^N</td>
<td>Shift Out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>017</td>
<td>0F</td>
<td>SI</td>
<td>^O</td>
<td>Shift In</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>020</td>
<td>10</td>
<td>DLE</td>
<td>^P</td>
<td>Data link escape</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>021</td>
<td>11</td>
<td>DC1</td>
<td>^Q</td>
<td>Device control 1 (X-ON)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>022</td>
<td>12</td>
<td>DC2</td>
<td>^R</td>
<td>Device control 2 (TAPE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>023</td>
<td>13</td>
<td>DC3</td>
<td>^S</td>
<td>Device control 3 (X-OFF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>024</td>
<td>14</td>
<td>DC4</td>
<td>^T</td>
<td>Device control 4 (TAPE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>025</td>
<td>15</td>
<td>NAK</td>
<td>^U</td>
<td>Negative acknowledge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>026</td>
<td>16</td>
<td>SYN</td>
<td>^V</td>
<td>Synchronous idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>027</td>
<td>17</td>
<td>ETB</td>
<td>^W</td>
<td>End of transmission blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>030</td>
<td>18</td>
<td>CAN</td>
<td>^X</td>
<td>Cancel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>031</td>
<td>19</td>
<td>EM</td>
<td>^Y</td>
<td>End Of medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>032</td>
<td>1A</td>
<td>SS</td>
<td>^Z</td>
<td>Special sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>033</td>
<td>1B</td>
<td>ESC</td>
<td>s^K</td>
<td>Escape (^ [ )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>034</td>
<td>1C</td>
<td>FS</td>
<td>s^L</td>
<td>File separator (^ \ )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>035</td>
<td>1D</td>
<td>GS</td>
<td>s^M</td>
<td>Group separator (^ )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>036</td>
<td>1E</td>
<td>RS</td>
<td>s^N</td>
<td>Record separator (^ ` )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>037</td>
<td>1F</td>
<td>US</td>
<td>s^O</td>
<td>Unit separator (^ / )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>177</td>
<td>7F</td>
<td>DEL</td>
<td>s^0</td>
<td>Delete or rubout (^ _ )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sample Statements

This appendix shows a table that contains selected samples of the $f77$ statement types. The purpose is to provide a quick reference for syntax details of the more common variations of each statement type.

Nonstandard features are tagged with a small black cross (✠).

<table>
<thead>
<tr>
<th>TABLE B-1</th>
<th>FORTRAN Statement Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>ACCEPT *, A, I</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>ASSGN 9 TO I</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>C = 'abc'</td>
</tr>
<tr>
<td></td>
<td>C = &quot;abc&quot;</td>
</tr>
<tr>
<td></td>
<td>C = S // 'abc'</td>
</tr>
<tr>
<td></td>
<td>C = S(I:M)</td>
</tr>
<tr>
<td></td>
<td>L = L1 .OR. L2</td>
</tr>
<tr>
<td></td>
<td>L = I .LE. 80</td>
</tr>
<tr>
<td></td>
<td>N = N+1</td>
</tr>
<tr>
<td></td>
<td>X = '7FF00000'x</td>
</tr>
<tr>
<td></td>
<td>CURR = NEXT</td>
</tr>
<tr>
<td></td>
<td>NEXT.ID = 82</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>AUTOMATIC A, B, C</td>
</tr>
<tr>
<td></td>
<td>AUTOMATIC REAL P, D, Q</td>
</tr>
<tr>
<td></td>
<td>IMPLICIT AUTOMATIC REAL (X-Z)</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>BACKSPACE U</td>
</tr>
<tr>
<td></td>
<td>BACKSPACE(UNIT=U,IOSTAT=I, ERR=9)</td>
</tr>
<tr>
<td>BLOCK DATA</td>
<td>BLOCK DATA</td>
</tr>
<tr>
<td></td>
<td>BLOCK DATA COEFFS</td>
</tr>
</tbody>
</table>
### TABLE B-1  FORTRAN Statement Samples  (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BYTE</strong></td>
<td>BYTE A, B, C</td>
<td>Initialize A and B</td>
</tr>
<tr>
<td></td>
<td>BYTE A, B, C(10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BYTE A /'x'/, B /255/, C(10)</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>CALL P( A, B )</td>
<td>Alternate return</td>
</tr>
<tr>
<td></td>
<td>CALL P( A, B, *9 )</td>
<td>Alternate return</td>
</tr>
<tr>
<td></td>
<td>CALL P( A, B, 49 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL P</td>
<td></td>
</tr>
<tr>
<td><strong>CHARACTER</strong></td>
<td>CHARACTER C<em>80, D</em>1(4)</td>
<td>Initialize B and C*</td>
</tr>
<tr>
<td></td>
<td>CHARACTER*18 A, B, C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHARACTER A, B*3 /'xyz'/, C /'z'/</td>
<td></td>
</tr>
<tr>
<td><strong>CLOSE</strong></td>
<td>CLOSE ( UNIT=I )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLOSE( UNIT=U, ERR=90, IOSTAT=I )</td>
<td></td>
</tr>
<tr>
<td><strong>COMMON</strong></td>
<td>COMMON / DELTAS / H, P, T</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMON X, Y, Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMON P, D, Q(10,100)</td>
<td></td>
</tr>
<tr>
<td><strong>COMPLEX</strong></td>
<td>COMPLEX U, V, U(3,6)</td>
<td>Double complex</td>
</tr>
<tr>
<td></td>
<td>COMPLEX U*16</td>
<td>Quad complex</td>
</tr>
<tr>
<td></td>
<td>COMPLEX U*32 <strong>SPARC only</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPLEX U/(1.0,1.0)/,V/(1.0,10.0)/</td>
<td>Initialize U and V</td>
</tr>
<tr>
<td><strong>CONTINUE</strong></td>
<td>100 CONTINUE</td>
<td></td>
</tr>
<tr>
<td><strong>DATA</strong></td>
<td>DATA A, C / 4.01, 'z' /</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA (V(I),I=1,3) / .7, .8, .9/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA ARRAY(4,4) / 1.0/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA B,O,X,Y /B'00111111', O'37', X'1f', Z'1f' /</td>
<td></td>
</tr>
<tr>
<td><strong>DECODE</strong></td>
<td>DECODE ( 4, 1, S ) V</td>
<td></td>
</tr>
<tr>
<td><strong>DIMENSION</strong></td>
<td>DIMENSION ARRAY(4, 4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIMENSION V(1000), W(3)</td>
<td></td>
</tr>
<tr>
<td><strong>DO</strong></td>
<td>DO 100 I = INIT, LAST, INCR</td>
<td>Unlabeled DO</td>
</tr>
<tr>
<td></td>
<td>... 100 CONTINUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DO I = INIT, LAST</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... END DO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DO WHILE ( DIFF .LE. DELTA )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... END DO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DO 100 WHILE ( DIFF .LE. DELTA )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... 100 CONTINUE</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Examples</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>DOUBLE COMPLEX*</td>
<td>DOUBLE COMPLEX U, V</td>
<td>COMPLEX*16 *</td>
</tr>
<tr>
<td></td>
<td>DOUBLE COMPLEX U, V</td>
<td>COMPLEX *</td>
</tr>
<tr>
<td></td>
<td>COMPLEX U/(1.0,1.0D0)/,</td>
<td>Initialize U and V *</td>
</tr>
<tr>
<td></td>
<td>V/(1.0,1.0D0)/</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION A, D, Y(2)</td>
<td>REAL*8 *</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION A,D/1.2D3/,Y(2)</td>
<td>Initialize D *</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
<td>Compare to IF (Block)</td>
</tr>
<tr>
<td>ELSE IF</td>
<td>ELSE IF</td>
<td></td>
</tr>
<tr>
<td>ENCODE *</td>
<td>ENCODE( 4, I, T ) A, B, C</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>END</td>
<td></td>
</tr>
<tr>
<td>END DO *</td>
<td>END DO</td>
<td>Compare to DO</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>ENDFILE ( UNIT=I )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDFILE I</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDFILE( UNIT=U, IOSTAT=I, ERR=9 )</td>
<td></td>
</tr>
<tr>
<td>END IF</td>
<td>END IF</td>
<td></td>
</tr>
<tr>
<td>END MAP *</td>
<td>END MAP</td>
<td>Compare to MAP</td>
</tr>
<tr>
<td>END STRUCTURE</td>
<td>END STRUCTURE</td>
<td>Compare to STRUCTURE</td>
</tr>
<tr>
<td>END UNION *</td>
<td>END UNION</td>
<td>Compare to UNION</td>
</tr>
<tr>
<td>ENTRY</td>
<td>ENTRY SCHLEP( X, Y )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENTRY SCHLEP( A1, A2, *4 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENTRY SCHLEP</td>
<td></td>
</tr>
<tr>
<td>EQUIVALENCE</td>
<td>EQUIVALENCE ( V(1), A(1,1) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EQUIVALENCE ( V, A )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EQUIVALENCE (X,V(10)), (P,D,Q)</td>
<td></td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>EXTERNAL RNGKTA, FIT</td>
<td></td>
</tr>
</tbody>
</table>
TABLE B-1  FORTRAN Statement Samples  (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT</td>
<td>10 FORMAT(/\n2X,213,3F6.1,4E12.2,2A6,3L2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(/\n2D6.1, 3G12.2 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(2I3.3,3G6.1E3,4E12.2E3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 FORMAT('a quoted string',</td>
<td>Strings *</td>
</tr>
<tr>
<td></td>
<td>&quot;another&quot;, I2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(18Hhollerith string, I2)</td>
<td>Hollerith</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT( 1X, T10, A1, T20, A1 )</td>
<td>Tabs</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(5X,TR10,A1,TR10,A1,TL5,A1)</td>
<td>Tab right, left</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(&quot; Init=&quot;&quot;, I2, :, 3X,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Last=&quot;&quot;, I2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(1X,&quot;Enter path name &quot;, $)</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT( F4.2, Q, 80 A1</td>
<td>Q *</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT(' Octal ',O6,' , Hex ',Z6 )</td>
<td>Octal, hex *</td>
</tr>
<tr>
<td></td>
<td>10 FORMAT( 3F&lt;N&gt;.2 )</td>
<td>Variable expression *</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>FUNCTION Z( A, B )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FUNCTION W( P,D, *9 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHARACTER FUNCTION R*4(P,D,*9 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER*2 FUNCTION M( I, J )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Short integer *</td>
</tr>
<tr>
<td>GO TO</td>
<td>GO TO 99</td>
<td>Unconditional</td>
</tr>
<tr>
<td></td>
<td>GO TO I, ( 10, 50, 99 )</td>
<td>Assigned</td>
</tr>
<tr>
<td></td>
<td>GO TO I</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GO TO ( 10, 50, 99), I</td>
<td>Computed</td>
</tr>
</tbody>
</table>
### Table B-1  FORTRAN Statement Samples (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF ( I -K ) 10, 50, 90</td>
<td>Arithmetic IF</td>
</tr>
<tr>
<td></td>
<td>IF ( L ) RETURN</td>
<td>LOGICAL IF</td>
</tr>
<tr>
<td></td>
<td>IF ( L ) THEN</td>
<td>BLOCK IF</td>
</tr>
<tr>
<td></td>
<td>N=N+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL CALC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>K=K+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL DISP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENDIF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF ( C .EQ. 'a' ) THEN</td>
<td>BLOCK IF</td>
</tr>
<tr>
<td></td>
<td>NA=NA+1</td>
<td>With ELSE IF</td>
</tr>
<tr>
<td></td>
<td>CALL APPEND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ELSE IF ( C .EQ. 'b' ) THEN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NB=NB+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL BEFORE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ELSE IF ( C .EQ. 'c' ) THEN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NC=NC+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL CENTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>END IF</td>
<td></td>
</tr>
<tr>
<td>IMPLICIT</td>
<td>IMPLICIT COMPLEX (U-W,Z)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IMPLICIT UNDEFINED (A-Z)</td>
<td></td>
</tr>
<tr>
<td>INCLUDE</td>
<td>INCLUDE 'project02/header'</td>
<td></td>
</tr>
<tr>
<td>INQUIRE</td>
<td>INQUIRE( UNIT=3, OPENED=OK )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INQUIRE( FILE='mydata', EXIST=OK )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INQUIRE( UNIT=3, OPENED=OK, IOSTAT=ERRNO )</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER C, D(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER C*2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER*4 A, B, C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER A/ 100 /, B, C / 9 /</td>
<td>Initialize A and C *</td>
</tr>
<tr>
<td>INTRINSIC</td>
<td>INTRINSIC SQRT, EXP</td>
<td></td>
</tr>
<tr>
<td>LOGICAL</td>
<td>LOGICAL C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOGICAL B<em>1, C</em>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOGICAL*1 B, C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOGICAL*4 A, B, C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOGICAL B / .FALSE. /, C</td>
<td>Initialize B *</td>
</tr>
</tbody>
</table>
### TABLE B-1 FORTRAN Statement Samples (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map *</td>
<td>MAP</td>
<td>Compare to STRUCTURE and UNION</td>
</tr>
<tr>
<td></td>
<td>CHARACTER *18 MAJOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>END MAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER*2 CREDITS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHARACTER*8 GRAD_DATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>END MAP</td>
<td></td>
</tr>
<tr>
<td>NAMELIST *</td>
<td>NAMELIST /CASE/ S, N, D</td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td>OPEN( UNIT=3, FILE=&quot;data.test&quot; )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OPEN( UNIT=3, IOSTAT=ERRNO )</td>
<td></td>
</tr>
<tr>
<td>OPTIONS *</td>
<td>OPTIONS /CHECK /EXTEND_SOURCE</td>
<td></td>
</tr>
<tr>
<td>PARAMETER</td>
<td>PARAMETER (A=“xyz”), (PI=3.14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PARAMETER (A=“z”, PI=3.14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PARAMETER X=11, Y=X/3</td>
<td></td>
</tr>
<tr>
<td>PAUSE</td>
<td>PAUSE</td>
<td></td>
</tr>
<tr>
<td>POINTER *</td>
<td>POINTER ( P, V ), ( I, X )</td>
<td></td>
</tr>
<tr>
<td>PRAGMA *</td>
<td>EXTERNAL RNG !$PRAGMA C(RNG) C() directive</td>
<td></td>
</tr>
<tr>
<td>PROGRAM</td>
<td>PROGRAM FIDDLE</td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td>PRINT *, A, I</td>
<td>List-directed</td>
</tr>
<tr>
<td></td>
<td>PRINT 10, A, I</td>
<td>Formatted</td>
</tr>
<tr>
<td></td>
<td>PRINT 10, M</td>
<td>Array M</td>
</tr>
<tr>
<td></td>
<td>PRINT 10, (M(I),I=J,K)</td>
<td>Implied-DO</td>
</tr>
<tr>
<td></td>
<td>PRINT 10, C(I:K)</td>
<td>Substring</td>
</tr>
<tr>
<td></td>
<td>PRINT *(A6,I3)’, A, I</td>
<td>Character constant format</td>
</tr>
<tr>
<td></td>
<td>PRINT FMT=(A6,I3)’, A, I</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRINT S, I</td>
<td>Switch variable has format number</td>
</tr>
<tr>
<td></td>
<td>PRINT FMT=S, I</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRINT G</td>
<td>Namelist *</td>
</tr>
<tr>
<td>READ</td>
<td>READ *, A, I</td>
<td>List-directed</td>
</tr>
<tr>
<td></td>
<td>READ 1, A, I</td>
<td>Formatted</td>
</tr>
<tr>
<td></td>
<td>READ 10, M</td>
<td>Array M</td>
</tr>
<tr>
<td></td>
<td>READ 10, (M(I),I=J,K)</td>
<td>Implied-DO</td>
</tr>
<tr>
<td></td>
<td>READ 10, C(I:K)</td>
<td>Substring</td>
</tr>
<tr>
<td></td>
<td>READ *(A6,I3)’, A, I</td>
<td>Character constant</td>
</tr>
<tr>
<td>Name</td>
<td>Examples</td>
<td>Comments</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>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</td>
<td>Formatted read from a file</td>
<td></td>
</tr>
<tr>
<td>READ ( *, 2 ) X, Y</td>
<td>Formatted read from standard input</td>
<td></td>
</tr>
<tr>
<td>READ ( *, 10 ) M</td>
<td>Array M</td>
<td></td>
</tr>
<tr>
<td>READ ( *, 10 ) (M(I),I=J,K)</td>
<td>Implied-DO</td>
<td></td>
</tr>
<tr>
<td>READ ( *, 10 ) C(I:K)</td>
<td>Substring</td>
<td></td>
</tr>
<tr>
<td>READ ( 1, * ) X, Y READ ( *, * ) X, Y</td>
<td>List-directed from file—from standard input</td>
<td></td>
</tr>
<tr>
<td>READ ( 1, *(A6,I3)') X, Y READ ( 1, FMT='(A6,I3)') X, Y</td>
<td>Character constant format</td>
<td></td>
</tr>
<tr>
<td>READ ( 1, C ) X, Y READ ( 1, FMT=C ) X, Y</td>
<td>Switch variable has format number</td>
<td></td>
</tr>
<tr>
<td>READ ( 1, S ) X, Y READ ( 1, FMT=S ) X, Y</td>
<td>Namelist read * Namelist read from a file *</td>
<td></td>
</tr>
<tr>
<td>READ ( *, G ) READ ( 1, G )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ ( 1, END=8, ERR=9 ) X, Y</td>
<td>Unformatted direct access</td>
<td></td>
</tr>
<tr>
<td>READ ( 1, REC=3 ) V</td>
<td>Unformatted direct access</td>
<td></td>
</tr>
<tr>
<td>READ ( 1, 2, REC=3 ) V</td>
<td>Formatted direct access</td>
<td></td>
</tr>
<tr>
<td>READ ( CA, 1, END=8, ERR=9 ) X, Y</td>
<td>Internal formatted sequential</td>
<td></td>
</tr>
<tr>
<td>READ ( CA, *, END=8, ERR=9 ) X, Y</td>
<td>Internal list-directed sequential access *</td>
<td></td>
</tr>
<tr>
<td>READ (CA,REC=4,END=8,ERR=9) X,Y</td>
<td>Internal direct access *</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>REAL R, M(4) REAL R<em>4 REAL</em>8 A, B, C REAL*16 A, B, C SPARC only</td>
<td>Double precision * Quad precision *</td>
</tr>
<tr>
<td>REAL A / 3.14 /, B, C / 100.0 /</td>
<td>Initialize A and C *</td>
<td></td>
</tr>
<tr>
<td>RECORD</td>
<td>RECORD /PROD/ CURR,PRIOR,NEXT</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>RETURN RETURN 2</td>
<td>Standard return Alternate return</td>
</tr>
<tr>
<td>Name</td>
<td>Examples</td>
<td>Comments</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>REWIND</td>
<td>REWIND 1&lt;br&gt;REWIND I&lt;br&gt;REWIND ( UNIT=U, IOSTAT=I, ERR=9 )</td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>SAVE A, /B/, C</td>
<td></td>
</tr>
<tr>
<td>STATIC</td>
<td>STATIC A, B, C&lt;br&gt;STATIC REAL P, D, Q&lt;br&gt;IMPLICIT STATIC REAL (X-Z)</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td>STOP&lt;br&gt;STOP &quot;all done&quot;</td>
<td></td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>STRUCTURE /PROD/&lt;br&gt;INT INTEGER<em>4 ID / 99 /&lt;br&gt;CHARACTER</em>18 NAME&lt;br&gt;CHARACTER<em>8 MODEL / 'XL' /&lt;br&gt;REAL</em>4 COST&lt;br&gt;REAL*4 PRICE</td>
<td>END STRUCTURE&lt;br&gt;Alternate return *</td>
</tr>
<tr>
<td>SUBROUTINE</td>
<td>SUBROUTINE SHR( A, B, *9 )&lt;br&gt;SUBROUTINE SHR( A, B, &amp;9 )&lt;br&gt;SUBROUTINE SHR( A, B )&lt;br&gt;SUBROUTINE SHR</td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td>TYPE *, A, I</td>
<td>Compare to PRINT</td>
</tr>
<tr>
<td>UNION</td>
<td>UNION&lt;br&gt;MAP&lt;br&gt;CHARACTER<em>18 MAJOR&lt;br&gt;END MAP&lt;br&gt;MAP&lt;br&gt;INTEGER</em>2 CREDITS&lt;br&gt;CHARACTER*8 GRAD_DATE&lt;br&gt;END MAP&lt;br&gt;END UNION</td>
<td>Compare to STRUCTURE</td>
</tr>
<tr>
<td>VIRTUAL</td>
<td>VIRTUAL M(10,10), Y(100)</td>
<td></td>
</tr>
<tr>
<td>VOLATILE</td>
<td>VOLATILE V, Z, MAT, /INI/</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE B-1 FORTRAN Statement Samples (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE</td>
<td>WRITE( 1, 2 ) X, Y</td>
<td>Formatted write to a file</td>
</tr>
<tr>
<td></td>
<td>WRITE( UNIT=1, FMT=2 ) X, Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, 2, ERR=8, END=9 ) X, Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(UNIT=1,FMT=2,ERR=8,END=9) X,Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE( *, 2 ) X, Y</td>
<td>Formatted write to stdout (Array M)</td>
</tr>
<tr>
<td></td>
<td>WRITE( *, 10 ) M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE( *, 10 ) (M(I),I=J,K)</td>
<td>Implied-DO</td>
</tr>
<tr>
<td></td>
<td>WRITE( *, 10 ) C(I:K)</td>
<td>Substring</td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, * ) X, Y</td>
<td>List-directed write to a file</td>
</tr>
<tr>
<td></td>
<td>WRITE( *, * ) X, Y</td>
<td>List-directed write to standard output</td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, 'A6,I3') X, Y</td>
<td>Character constant format</td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, FMT='A6,I3') X, Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, C ) X, Y</td>
<td>Character variable format</td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, FMT=C ) X, Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, S ) X, Y</td>
<td>Switch variable has format number</td>
</tr>
<tr>
<td></td>
<td>WRITE( 1, FMT=S ) X, Y</td>
<td></td>
</tr>
</tbody>
</table>
| | WRITE( *, CASE ) | Namelist write *
| | WRITE( 1, CASE ) | Namelist write to a file *
| | WRITE( 1, END=8, ERR=9 ) X, Y | Unformatted sequential access |
| | WRITE( 1, REC=3 ) V | Unformatted direct access |
| | WRITE( 1 '3 ) V | |
| | 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 * |
Data Representations

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.

This appendix is a brief introduction to data representation. For more in-depth explanations, see the Sun WorkShop Fortran Programming Guide and Numerical Computation Guide.

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. Quad is *SPARC only*.

$$(-1)^{\text{sign}} \times 2^{\text{exponent-bias}} \times 1.f$$

<table>
<thead>
<tr>
<th>TABLE C-1</th>
<th>Floating-point Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
</tr>
<tr>
<td>Sign</td>
<td>Bit 31</td>
</tr>
<tr>
<td>Exponent</td>
<td>Bits 30–23</td>
</tr>
<tr>
<td>Bias 127</td>
<td>Bits 1023</td>
</tr>
<tr>
<td>Fraction</td>
<td>Bits 22–0</td>
</tr>
<tr>
<td>Range approx.</td>
<td>$3.402823e+38$</td>
</tr>
<tr>
<td></td>
<td>$1.797693e+38$</td>
</tr>
</tbody>
</table>
Extreme Exponents

The representations of extreme exponents are as follows.

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:

\((-1)^{\text{sign}} \times 2^{\text{1-bias}} \times 0.f\)

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.
IEEE Representation of Selected Numbers

The values here are as shown by `dbx`, in hexadecimal.

<table>
<thead>
<tr>
<th>Value</th>
<th>Single-Precision</th>
<th>Double-Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>00000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>-0</td>
<td>80000000</td>
<td>8000000000000000</td>
</tr>
<tr>
<td>+1.0</td>
<td>3F800000</td>
<td>3FF0000000000000</td>
</tr>
<tr>
<td>-1.0</td>
<td>BF800000</td>
<td>BFF0000000000000</td>
</tr>
<tr>
<td>+2.0</td>
<td>40000000</td>
<td>4000000000000000</td>
</tr>
<tr>
<td>+3.0</td>
<td>40400000</td>
<td>4008000000000000</td>
</tr>
<tr>
<td>+Infinity</td>
<td>7F800000</td>
<td>7FF0000000000000</td>
</tr>
<tr>
<td>-Infinity</td>
<td>FF800000</td>
<td>FFF0000000000000</td>
</tr>
<tr>
<td>NaN</td>
<td>7Fxxxxxx</td>
<td>7FFxxxxxxxxxxxxx</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub</td>
<td>Subnormal number</td>
</tr>
<tr>
<td>Num</td>
<td>Normalized number</td>
</tr>
</tbody>
</table>
### TABLE C-3  Extreme Value Abbreviations (Continued)

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inf</td>
<td>Infinity (positive or negative)</td>
</tr>
<tr>
<td>NaN</td>
<td>Not a Number</td>
</tr>
<tr>
<td>Uno</td>
<td>Unordered</td>
</tr>
</tbody>
</table>

### TABLE C-4  Extreme Values: Addition and Subtraction

<table>
<thead>
<tr>
<th>Left Operand</th>
<th>Right Operand</th>
<th>0</th>
<th>Sub</th>
<th>Num</th>
<th>Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Sub</td>
<td>Num</td>
<td>Inf</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>Sub</td>
<td>Sub</td>
<td>Num</td>
<td>Inf</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Num</td>
<td>Inf</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Inf</td>
<td>Inf</td>
<td>Inf</td>
<td>Inf</td>
<td>Note</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>

Note: Inf ± Inf and Inf + Inf = Inf; Inf - Inf = NaN.

### TABLE C-5  Extreme Values: Multiplication

<table>
<thead>
<tr>
<th>Left Operand</th>
<th>Right Operand</th>
<th>0</th>
<th>Sub</th>
<th>Num</th>
<th>Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>0</td>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num</td>
<td>0</td>
<td>NS</td>
<td>Num</td>
<td>Inf</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Inf</td>
<td>NaN</td>
<td>Inf</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>
In the above table, NS means either Num or Sub result possible.

### TABLE C-6  Extreme Values: Division

<table>
<thead>
<tr>
<th>Left Operand</th>
<th>Right Operand</th>
<th>0</th>
<th>Sub</th>
<th>Num</th>
<th>Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>Inf</td>
<td>Num</td>
<td>Num</td>
<td>0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Num</td>
<td>Inf</td>
<td>Num</td>
<td>Num</td>
<td>0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>Inf</td>
<td>Inf</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>

### TABLE C-7  Extreme Values: Comparison

<table>
<thead>
<tr>
<th>Left Operand</th>
<th>Right Operand</th>
<th>0</th>
<th>Sub</th>
<th>Num</th>
<th>Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>=</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>Uno</td>
</tr>
<tr>
<td>Sub</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>Uno</td>
</tr>
<tr>
<td>Num</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>Uno</td>
</tr>
<tr>
<td>Inf</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>Uno</td>
</tr>
<tr>
<td>NaN</td>
<td>Uno</td>
<td>Uno</td>
<td>Uno</td>
<td>Uno</td>
<td>Uno</td>
<td>Uno</td>
</tr>
</tbody>
</table>

Notes:
- 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.

**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 in the following tables.

**TABLE C-8  **Bits and Bytes for Intel and VAX Computers

<table>
<thead>
<tr>
<th>Byte ( n+3 )</th>
<th>Byte ( n+2 )</th>
<th>Byte ( n+1 )</th>
<th>Byte ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24</td>
<td>23 22 21 20 19 18 17 16</td>
<td>15 14 13 12 11 10 09 08</td>
<td>07 06 05 04 03 02 01 00</td>
</tr>
</tbody>
</table>

Most Significant Least significant

**TABLE C-9  **Bits and Bytes for 680x0 and SPARC Computers

<table>
<thead>
<tr>
<th>Byte ( n )</th>
<th>Byte ( n+1 )</th>
<th>Byte ( n+2 )</th>
<th>Byte ( n+3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24</td>
<td>23 22 21 20 19 18 17 16</td>
<td>15 14 13 12 11 10 09 08</td>
<td>07 06 05 04 03 02 01 00</td>
</tr>
</tbody>
</table>

Most Significant Least significant

The bits are numbered the same on these systems, even though the bytes are numbered differently.

Following are some possible problem areas:

- 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 SPARC system routines, you must convert them.
- If you convert character-to-integer or integer-to-character between architectures, you should use XDR.
- If you read binary data created on an architecture with a different byte order, then you must filter it to correct the byte order.

See also the \( \times \text{dr}(3N) \) man page.
VMS Language Extensions

This chapter describes the VMS language extensions that Sun WorkShop Fortran 77 supports. These extensions are all, of course, nonstandard.

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 dbx as well as f77.

VMS Language Features in Sun Fortran

This list is a summary of the VMS features that are included in f77. 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 (O)
  - Carriage Control ($)
  - Octal ($)
- 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:

  ```fortran
  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 -. Here are two consecutive operators:

\[
X = A \; \times \times \; -B
\]

The above statement is treated as follows:

\[
X = A \; \times \times \; (-B)
\]

■ Illegal real expressions

When the compiler finds a REAL expression where it expects an integer expression, it truncates and makes a type conversion to INTEGER.

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):

\[
\text{DATA N1 /B'0011111'/, N2/O'37'/, N3/X'1f'/, N4/Z'1f'}/
\]

■ 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:

```fortran
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:

```fortran
CALL SUBX ( I, *100, Z ) ! Standard
CALL SUBX ( I, &100, Z ) ! Nonstandard alternate syntax
```

- 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 accepted.
  Example: A nonstandard form for record specifier:

```fortran
READ ( K ' N ) LIST
```

The above is treated as:

```fortran
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 accepted
  - %LOC is treated as LOC
Appendix D  VMS Language Extensions

- %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 nh... 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 )
```

6 can be replaced by the variable N, as in:

```
1 FORMAT( 3F<N>.1 )
```

VMS Features Requiring -x1 or -vax=spec

You get most VMS features automatically without any special options. For a few of them, however, you must add the -x1 option on the f77 command line.

In general, you need this -x1 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 -x1 option forces the compiler to interpret it as VMS FORTRAN.

Note also the -vax=spec option, which allows specification of these VMS extensions individually. See the Fortran User's Guide for details.

Summary of Features That Require -x1 [d]

You must use -x1[d] to access the following features:

- Unformatted record size in words rather than bytes (-x1)
- VMS-style logical file names (-x1)
- Quote (”) character introducing octal constants (-x1)
- Backslash (\) as ordinary character within character constants (-x1)
- Nonstandard form of the PARAMETER statement (-x1)
Details of Features That Require \texttt{-x1[d]}

Here are the details:

- **Unformatted record size in words rather than bytes**
  
  In \texttt{f77}, direct-access, unformatted files are always opened with the logical record size in \texttt{bytes}.

  If the \texttt{-x1[d]} option is \textit{not} set, then the argument \texttt{n} in the \texttt{OPEN} option \texttt{RECL=n} is assumed to be the number of bytes to use for the record size.

  If the \texttt{-x1[d]} option is set, then the argument \texttt{n} in the \texttt{OPEN} option \texttt{RECL=n} is assumed to be the number of \textit{words}, so the compiler uses \texttt{n*4} as the number of bytes for the record size.

  If the \texttt{-x1[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 result could happen if the information is passed in variable character strings.

  The record size returned by an \texttt{INQUIRE} statement is \textit{not} adjusted by the compiler; that is, \texttt{INQUIRE} always returns the number of \texttt{bytes}.

  These record sizes apply to direct-access, unformatted files only.

- **VMS-style logical file names**

  If the \texttt{-x1[d]} option is set, then the compiler interprets VMS logical file names on the \texttt{INCLUDE} statement if it finds the environment variable, \texttt{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:

  \begin{verbatim}
  "lname1=path1; lname2=path2; ...  
  \end{verbatim}

  Remember these rules for VMS style logical file names:

  - Each \textit{lname} is a logical name and each \textit{path1}, \textit{path2}, and so forth, is the path name of a directory (without a trailing \texttt{/}).
  - It ignores all blanks when parsing this string.
  - It strips any trailing \texttt{/[no]list} from the file name in the \texttt{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 the `path1/file` form.
- For logical names, uppercase and lowercase are significant. If a logical name is encountered on the `INCLUDE` statement which is not specified in the `LOGICALNAMEMAPPING`, the file name is used, unchanged.
- 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:

  ```fortran
  JCOUNT = ICOUNT + "703
  ```

  The above statement is treated as:

  ```fortran
  JCOUNT = ICOUNT + 451
  ```

  If the `-xl[d]` option is *not* on, then the "703 is an error.

  With `-xl[d]`, the VMS FORTRAN notation "703 signals $f77$ to convert from the integer octal constant to its integer decimal equivalent, 451 in this case. In VMS FORTRAN, "703 cannot be the start of a character constant, because VMS FORTRAN character constants are delimited by apostrophes, not quotes.
- 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.
- 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:

  ```fortran
  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 treated as 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 code prints I and X. Without `-xld`, it does not print them.

- Align structures as in VMS FORTRAN

Use this feature if your program has some detailed knowledge of how VMS structures are implemented. If you need to share structures with C, you should use the default: no `-xld`

**Unsupported VMS FORTRAN**

Most VMS FORTRAN extensions are incorporated into the `f77` compiler. The compiler writes messages to standard error for any unsupported statements in the source file. The following is a list of the few VMS statements that are *not* supported.

- `DEFINE FILE` statement
- `DELETE` statement
- `UNLOCK` statement
- `FIND` statement
- `REWRITE` statement
- `KEYID` and key specifiers in `READ` statements
- Nonstandard `INQUIRE` specifiers
  - `CARRIAGECONTROL`
  - `DEFAULTFILE`
  - `KEYED`
  - `ORGANIZATION`
  - `RECORDTYPE`
- Nonstandard `OPEN` specifiers
  - `ASSOCIATEVARIABLE`
  - `BLOCKSIZE`
  - `BUFFERCOUNT`
  - `CARRIAGECONTROL`
  - `DEFAULTFILE`
The intrinsic function, %DESCR

The following parameters on the OPTIONS statement:

- [NO]G_FLOATING
- [NO]F77
- CHECK=[NO]OVERFLOW
- CHECK=[NO]UNDERFLOW

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 you are inconsistent about capitalization, distinctions are made where none are intended.

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 feature works if the long integer fits in 16 bits, because the VAX addresses an integer by its low-order byte. This feature does not work on SPARC systems.

Those VMS system calls that are directly tied to that operating system

Initializing a common block in more than one subprogram

Alphabetizing common blocks so you can rely or depend on the order in which blocks are loaded. You can specify the older with the -M mapfile option to ld.

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.
Index

SYMBOLS
!, 19
$, 14
      edit descriptor, 279
      NAMELIST delimiter, 319
%DESCR, 374
%FILL, 58, 245
%LOC, 374
%REF, 374
%VAL, 374
$, 103, 105, 319, 374
', 374
', 34
*, 107, 109, 374
      alternate return, 103, 105
      comments, 19
+, 280
., 59
Δ, blank character, 276
/, 304, 312
// concatenate string, 76
;:
      array bounds, 49
      character constants, 36
      edit descriptor, 304
      substring operator, 54
<>:, 13, 159, 160
=, 91
?, 322
*, 12
\, 13
_\, 14

NUMERICS
0, 1, + vertical format control, 279

A
A format specifier, 281
ACCEPT, 88, 374
access
      append option in open, 199
      modes, 265
      options in OPEN, 199
      SEQUENTIAL in OPEN file, 199
      two modes for accessing a file, 265
ACHAR, 330
address
      assignment, pointers, 65
      malloc, 65
adjustable array bounds, 50
alignment
      structures, as in VMS, 376, 378
      summary of data types, 31
allowed I/O combinations, 265
alternate
      octal notation, 39
      return, 233, 374
ampersand, alternate return, 103, 105, 374
C

comments, 19
directive, 19
CALL, 103
carriage control, 266, 279
$ , 279
all files, 268
blank, 0, 1, 279
first character, 279
space, 0, 1, 279
carriage return, $ edit descriptor, 279
case, 14, 15
CHAR, 95, 330
CHARACTER
data type, 26
statement, 107
character
array, 49
assignment, 77, 78, 79, 95
boundary, 31
concatenate, 76
constant
delimiter, 319
NAMELIST, 320
constants, 34
declared length, 109
declaring the length, 108
dummy argument, 108
editing, 281
expression, 76
format specifier, 372
functions, 335
join, 76
null constants, 34
operator, 76
packing, 107
set, 12
strings, 108
substring, 54
valid characters in names, 14
characters, special, 13
CLOSE, 110
CMPLX, 330
colon (:) 
array bounds, 49
edit descriptor, 304

substring operator, 54
column one formatting, 266
combinations of I/O, 265
commands, 1
commas in formatted input, 300
comments, 19
!, 19
*, 19
blank-line, 19
C, 19
embedded, 372
end-of-line, 19, 372
COMMON, 14, 112, 372
complex
array, 116
constant in NAMELIST, 321
constants, 36
data type, 26
statement, 114
COMPLEX*16, 27, 37
COMPLEX*32, 27, 37
COMPLEX*8, 27
computed GO TO, 166
concatenation
of strings, 76
operator, 76
conditional termination control, 304
consecutive
commas, NAMELIST, 321
operators, 373
constant
expression, 83
names (symbolic constants), 14
null character constants, 34
octal, 373
radix-50, 372
signed, 34
typeless numeric, 373
unsigned, 34
values in NAMELIST, 320
constants, 33
binary, 43
characters, 34
complex, 36
COMPLEX*16, 37
COMPLEX*32, 37
double complex, 37
double-precision real, 41
fortran 95-style, 46
hex, 43
integer, 38
logical, 39
octal, 43
quad complex, 37
quad real, 42
real, 39
REAL*16, 42
REAL*4, 40
REAL*8, 41
typeless, 43
continuation lines, 17, 18
CONTINUE, 117
control characters, 14, 44, 77
in assignment, 79, 95
meanings, 353

D

d comments, 19
d format specifier, 294
DATA, 118
data
namelist syntax, 319, 322
representation
double precision, 365
real number, 365
signed infinity, 366
type
BYTE, 25
CHARACTER, 26
COMPLEX, 26
COMPLEX*16, 27
COMPLEX*32, 27
COMPLEX*8, 27
DOUBLE COMPLEX, 27, 27
DOUBLE PRECISION, 27
INTEGER, 28
INTEGER*4, 28
LOGICAL, 29
LOGICAL*1, 25, 29
LOGICAL*2, 29
LOGICAL*4, 29, 30
of an expression, 73
properties, 25
quad real, 30
REAL, 30
REAL*16, 30
REAL*4, 30
REAL*8, 30
short integer, 28
data types and data structures, 23
DBLE, 329
DBLEQ, 329
DCMPLX, 330
deallocate memory by free, 66
debug statement, 377
decimal point in octal or hex input, 288
declaration
field, 56, 194, 244
initialize in, 372
map, 62, 253
record, 58, 230
structure, 56
union, 62
DECODE, 121
default
inquire options, 182
delimiter
ccharacter constant, 319
NAMELIST: $ or &, 319
DFLOAT, 329
DIMENSION, 123
dimension arrays, 49
direct
I/O, 270
I/O record specifier, 222, 271, 374
DIRECT option for ACCESS in OPEN file, 199
directives
general, 19
parallelization, 20
DISPOSE option for CLOSE, 374
DO, 126
DO WHILE, 131
DOALL directive, 21
dollar sign
edit descriptor, 279
in names, 14
NAMELIST delimiter, 319
DOSERIAL directive, 21
DOUBLE COMPLEX, 27, 134
DOUBLE PRECISION, 27, 136
double quote, 375, 377
character constants, 34
preceding octal constants, 39
double spacing print, 267
double-complex
arrays, 135
costants, 37
data type, 27
double-precision complex, 27
complex function, 344
data representation, 365
editing, 294
real constants, 41
DREAL, 329
dummy arguments and NAMELIST, 316

E
-e, 18
E format specifier, 296
edit descriptor
/, 304
;, 304
A, 281
D, 294
E, 296
F, 298
G, 299
I, 284
L, 286
P, 302
positional, 289
Q, 300
S, 303
SP, 303
SS, 303
SU, 303
T, 289
X, 289
ELSE, 137
ELSE IF, 139
embedded comments, 372
empty spaces in structures, 58, 245
ENCODE, 121, 141
END, 142
END DO, 143
END FILE, 144
END IF, 146
END MAP, 147
end of text, 77
END STRUCTURE, 148
END UNION, 149
end-of-line comments, 19, 372
ENTRY, 150
environmental inquiry functions, 337
epbase, 337
ephuge, 337
epmax, 337
epmin, 337
epmrsp, 337
epprec, 337
eptiny, 337
equals statement, 91
EQUIVALENCE, 153
ERR
INQUIRE, 182
OPEN specifier, 201
READ, 223
WRITE, 258
error
I/O, 264
escape sequences, 36
evaluation of expressions, 85
executable statements, 16
exponential editing, 296
exponents in octal or hex input, 288
expression
arithmetic, 70, 71
character, 76
constant, 83
evaluation, 85
logical, 80
variable format, 159
extended source lines, 18
extensions, 12
EXTERNAL, 155
extract substring, 54
extreme
  exponent data representation, 366
  values for arithmetic operations, 367

F
F format specifier, 298
field, 56
  argument that is a field, 58, 230
  COMMON with a field, 58, 230
  declaration, 56, 194, 244
  DIMENSION with a field, 58, 230
  dimensioning in type statements, 57, 245
  EQUIVALENCE, not allowed in, 58, 231
  list, 57
  list of a structure, 56, 245
  map with a field, 62, 253
  name, %FILL, 58, 245
  NAMELIST, not allowed in, 58, 231
  offset, 58, 245
  reference, 59
  SAVE, not allowed in, 58, 231
  type, 58, 245
file
  append on open, 199
  carriage control on all files, 268
  INQUIRE, 180
  internal, 272
  limit on number open, 263
  names, VMS logical, 375, 376
  preattached, 269
  properties, 180
  query, 180
  See also scratch files
  specifier for opening, 199
  two access modes for, 265
FILE= specifier, 373
filling with asterisks or spaces, hex and octal
  output, 289
first character carriage control, 279
FLOAT, 329
form feed character, 36
FORM specifier in OPEN, 200
FORM='BINARY', 310
FORM='PRINT', 266
FORMAT, 157
form
  $, 279
  /, 304
  ;, 304
  A, 281
  B, 278
  BN, 278
  BZ, 278
  D, 294
defaults for field descriptors, 276
  E, 296
  F, 298
  G, 299
  I, 284
  L, 286
  O, 287
  of source line, 17
  P, 302
  Q, 300
  R, 294
  read into hollerith edit descriptor, 284
  S, 303
  SP, 303
  specifier, 372
  SS, 303
  standard fixed, 17
  SU, 303
  tab, 18
  variable expressions, 159, 160
  vertical control, 279
  X, 289
  Z, 287
form specifier ",", 293
formats, 305
  runtime, 217, 222, 258, 284, 305
  variable format expressions, 307
formatted
  I/O, 273
  output, 267
formatted I/O, 273
forms of I/O, 265
FORTRAN
  list of statements, 17
  Fortran 95-style constants, 46
free, 66, 338
FUNCTION, 161
function
  bit-manipulation, 346
  double-precision complex, 344
  integer, 348
  intrinsic, 327
  length specifier, 373
  malloc, 65
  memory allocation and deallocation, 338
  names, 14
  type coercing, 349
  types, 24
  zero-extend, 350

G
  G format specifier, 299
  general real editing, 299
  GO TO, 164, 168
  GO TO assigned, 164
  GO TO unconditional, 168
  GO TO, computed, 166

H
  hex and octal
    format, 287
    format samples, 288
    input, 287
    input rules, 288
    output, 288, 289
  hexadecimal
    constants, 43
    initialization, 43
  hollerith, 94, 284
  horizontal positioning, 289

I
  I format specifier, 284
  I/O, 265
    allowable combinations, 265
    binary, 310
    direct, 270
    errors, 264
    random, 270

summary, 265
-1z, 28
IACHAR, 330
ICCHAR, 330
IDINT, 329
IEEE
  representation of selected numbers, 367
IF, 169, 170, 173
IFIX, 329
illegal REAL expressions, 373
IMPLICIT, 174
implicit
  none data typing, 372
  typing, 23
implicit data typing, 23
INCLUDE, 177, 376
initial line, 17
initialize
  in BLOCK DATA, 372
  in COMMON, 372
  in declaration, 372
input commas, 300
INQUIRE, 180, 182
inquire
  by file, 185
  by unit, 180, 185
  specifiers summary, 181
inquire option
  ACCESS, 182
  BLANK, 182
  defaults, 182
  DIRECT, 182
  ERR, 182
  EXIST, 183
  FILE, 183
  FORM, 183
  FORMATTED, 183
  IOSTAT, 183
  NAME, 183
  NAMED, 183
  NEXTREC, 184
  none for permissions, 180
  NUMBER, 184
  OPENED, 184
  RECL, 184
  SEQUENTIAL, 184
UNFORMATTED, 184
UNIT, 184
INT, 329
INTEGER, 28, 186
integer
and logical, 74
constants, 38
editing, 284
functions, 348
logical, mixed expressions, 73
operand with logical operator, 74
short, 39
INTEGER*2, 28
INTEGER*4, 28
INTEGER*8, 28
internal files, 272
INTRINSIC, 189
intrinsic functions, 325, 344
arithmetic, 327
character, 335
environmental inquiry, 337
mathematical, 333
memory allocation and deallocation, 338
special VMS, 374
trigonometric, 331
type conversions, 329
invalid characters for data, 14
iobin, 269
IOSTAT OPEN specifier, 201
IQINT, 329
ishift, 343
ISO 1539-1980 FORTRAN standard, 11

L
format specifier, 286
label of statement, 12
leading spaces or zeros, hex and octal output, 289
left-to-right
exception, 71
precedence, 71
len, declared length, 109
length
function length specifier, 161, 163, 373
LEN function, 109
line of source code, 18
names, 14
string, 109
variable length records, 200
line
formats, 17
length, 18
tab-format, 17, 372
line feed, 77
linked list, 215
list-directed
I/O, 312
input, 312
output, 312
output to a print file, 267
literal constant, 12
literals type REAL*16, 373
loc, 65, 338
location of scratch files, 201
LOGICAL, 29, 191
logical
assignment, 81, 94
constants, 39
editing, 286
expression, 80
expression meaning, 81
file names in the INCLUDE, 178
file names, VMS, 375, 376
IF, 173
integer, mixed, 74
LOGICAL*1 data type, 25
operator precedence, 80
unit preattached, 269
units, 263
LOGICAL*1, 29

J
join strings, 76

K
key word, 12
kind type parameter in constants, 46
LOGICAL*2, 29
LOGICAL*4, 29
LOGICAL*8, 30
long lines in source code, 18
lower case, 14, 15
lrshft, 343

M
malloc, 65, 338
man pages, 1
MANPATH environment variable, 1
MAP, 62, 194, 253
maximum
number of open files, 263
memory
get by malloc, 65
release by free, 66
memory allocation and deallocation functions, 338
mixed
integer and logical, 73, 74
mode, 72, 73
mixed mode, 73
mixing format of source lines, 18
modifying
carriage control, 279
multithreading
See parallelization

N
name of scratch files, 201
NAME option for OPEN, 374
NAMELIST, 195, 317, 319, 320
$, 318
@, 319
ask for names, 322
namelist-specifier, 317
NML=, 317
prompt for names, 322
restrictions, 316
WRITE, 317
namelist
data, 319, 322
data syntax, 320
END, 319
I/O, 315
negative values, hex and octal output, 289
nested substructure, 60
newline character, 36, 77
NML=, 318
noncharacter runtime format specifier, 372
none, implicit data typing, 372
nonexecutable statements, 16
nonstandard
PARAMETER, 375, 377
notation octal alternate, 39
null
character, 36
character constants, 34
data item, NAMELIST, 321
number of
continuation lines, 18
open files, 263
numeric constant, typeless, 373

O
ο
constant indicator, 43edit descriptor, 287
octal
alternate notation, 39
constant, 373
constants, 43
initialization, 43
octal and hex
format, 287
format samples, 288
input, 287
input rules, 288
output, 288, 289
offset of fields, 58, 245
omitted arguments, 372
OPEN
specifier
ACCESS, 199
BLANK, 201
ERR, 201
FILE, 199
FORM, 200
IOSTAT, 201
RECL, 200
STATUS, 201
UNIT, 198

OPEN
  options, 374
  print file, 266
  statement, 197, 202
open files, limit of, 263
operand, 69
operator, 69
  **, 70
  // concatenate string, 76
  : substring, 54
  character, 76
  concatenation, 76
  precedence, 71
  relational, 82
  two consecutive operators, 72, 373
  with extreme values, 367
optimization problems with pointers, 67
option
  DISPOSE for CLOSE, 374
  -e, 18
  i2 short integer, 28
  long lines, 18
  NAME for OPEN, 374
  number of continuation lines, 18
 OPTIONS, 205
order bit and byte, 369

POINTER, 212
pointer, 63, 212
  address assignment, 65
  address by LOC, 65, 212
  disallowed in NAMELIST, 316
  linked list, 215
  problems with optimization, 67
  restrictions, 66
  VOLATILE, 212
pointer-based variable, 66, 316
positional
  edit descriptor, 289
  format editing, 289
preattached
  files, 269
  logical units, 269
precedence
  logical operator, 80
  operators, 71
PRINT, 217
print file, 200, 266, 313
procedures, 16
PROGRAM, 220
program, 12
  names, 14
  units, 16
promote types, 72
prompt
  for namelist names, 322
  properties, file, 180

P
  p edit descriptor, 302
  packing character, 107
  padding, 19
PARAMETER
  nonstandard alternate, 375, 377
    statement, 57, 207, 245
parameter name, 14
PATH environment variable, 1
PAUSE, 210
permissions
  ACCESS in INQUIRE, 180
  pointee, 212
Q
  q edit descriptor, 300
  QCMLX, 330
  QEXT, 329
  QEXTD, 329
  QFLOAT, 329
  QREAL, 329
quad
  complex, 27
  complex constants, 37
  exponent, 42
  real constants, 42
  real data type, 30
Index

391

type REAL*16 literals, 373
quadruple precision, See quad quote, 375, 377
  character constants, 34
  format specifier, 293
  preceding octal constants, 39

R
  -r 4, 32
  radix, 294
  radix-50 constant, 372
  random
    I/O, 270
  READ, 221
  read
    into hollerith edit descriptor, 284
  REAL, 30, 227
  expressions, illegal, 373
  intrinsic, 329
  real
    arrays, 229
    constants, 39
    data representation of reals, 365
    editing, 294, 298
  REAL*16, 30, 42, 373
  REAL*4, 30, 40
  REAL*8, 30, 41
  RECL specifier in OPEN, 200
  rec1=1, variable length records, 200
  RECORD, 230
  record, 56
    argument that is a record, 58, 230
    assignment, 95
    AUTOMATIC, not allowed in, 231
    COMMON with a record, 58, 230
    DATA, not allowed in, 58, 231
    DIMENSION with a record, 58
    disallowed in NAMELIST, 316
    EQUIVALENCE, not allowed in, 58, 231
    NAMELIST, not allowed in, 58, 231
    PARAMETER, not allowed in, 231
    reference, 59
    SAVE, not allowed in, 58, 231
    size, unformatted, 375, 376
    specifier, direct-access, 221, 271, 374
    statement, 58
    STATIC, not allowed in, 231
    variable length, 200
  recursive, 96, 162, 240
  reference
    field, 59
    record, 59
  relational operator, 82
  release memory by free, 66
  repeat NAMELIST, 321
  representation of data, 365
  requesting namelist names, 322
  restrictions
    fields, 58, 245
    hex and octal output, 289
    NAMELIST, 316
    names, 14
    pointers, 66
    Q edit descriptor, 302
    records, 58, 230
    structures, 57, 244
    substructures, 62
  RETURN, 233
  return alternate, 233, 234, 374
  reverse solidus, 13
  REWIND, 235
  rshift, 343
  runtime formats, 217, 222, 258, 284, 305, 307

S
  S edit descriptor, 303
  same line response, 279
  sample statements, 355
  SAVE, 237
  scale
    control, 302
    factor, 302
  scratch files
    defined, 268
    location, 202
    naming, 201
  SCRATCH option for OPEN, 201
  SEQUENTIAL option for ACCESS in OPEN file, 199
input data, 319, 322
output, 317
statement, 316
OPEN statement, 197
record reference, 59
records, 58, 230
structure, 56, 244
unions, 62, 253

T
T edit descriptor, 289
tab, 13
central, 36
control, 289
format source, 18, 372
TASKCOMMON directive, 21
temporary files, 201
terminal
I/O, 279
termination control edit descriptor, 304
TMPDIR environment variable, 202
top of page, 267	
two consecutive operators, 373
TYPE, 249, 374
type
do coercing functions, 349
field names, 58, 245
REAL*16, 373
type, 250
typeless
do constants, 43
numeric constant, 373
types
array elements, 24
files, 265
functions, 24
summary of, 31

U
unary + or -, 373
unary operator, 71
unconditional GO TO, 168
underscore
in function or subprogram names, 14
names with, 14
unformatted
I/O, 307
record size, 375, 376
UNION, 253
union declaration, 62, 253
unit, logical unit preattached, 269
UNIT, OPEN specifier, 198
unsigned constant, 34
upper case, 14, 15

V
valid
do characters for data, 14
do characters in character set, 14
do characters in names, 14
does values, extreme for arithmetic operations, 367
does variable
boundary, 31
does definition of, 47
do Example FORMAT expression, 160
do name, 14
do part of iolist, 217
do variable formats, 159, 222, 258, 275, 284, 305, 306, 307
do variable-length records, 200
do vertical format control, 266
do $, 279
do space, 0, 1, +, 279
do vertical tab character, 36
VIRTUAL, 255, 372
VMS FORTRAN
do align structures, 376
do features with -x1
do backslash, 14, 36, 313
do D or d debug lines, 19
do debugging lines, 377
do logical file names, 178, 375, 376
do parameter form, 207, 209, 377
do quotes, 93
do record length, 200, 376
features with -x1
do record length, 184
unsupported extensions, 378
VOLATILE, 256

W
width defaults for field descriptors, 276
word boundary, 31
WRITE, 257

X
  constant indicator, 43
  edit descriptor, 289
-xl, 19, 26, 34, 36, 93, 374, 375, 377
-xld, 377
-xtypemap, 32

Z
  constant indicator, 43
  edit descriptor, 287
zero, leading, in hex and octal output, 289
zero-extend functions, 350