HEWLETT PACKARD

Domain FORTRAN Language Reference

Domain FORTRAN Language Reference

Order No. 000530-A01

Apollo Systems Division
A subsidiary of
HEWLETT
PACKARD

First Printing: November 1981
Last Printing: December 1990

10 9 8 7 6 5 4 3 2 1

Preface

The *Domain FORTRAN Language Reference* describes Domain FORTRAN, an extended version of the FORTRAN language as described by ANSI standard X3.9–1978 (also known as FORTRAN 77). In addition to describing all FORTRAN statements (including Domain extensions to the ANSI standard), this manual describes how to compile, bind, execute, and debug FORTRAN programs on the Domain system.

We've organized this manual as follows:

Chapter 1	Introduces Domain FORTRAN and provides an overview of its extensions.
Chapter 2	Defines Domain FORTRAN building blocks (such as the length of an identifier) and describes the structure of the main program.
Chapter 3	Explains all the Domain FORTRAN data types.
Chapter 4	Contains alphabetized listings describing all the statements you can use in the code portion of a program.
Chapter 5	Explains how to write and call subroutines, functions, statement functions, and block data subprograms.
Chapter 6	Details compiling, binding, debugging, and executing.
Chapter 7	Describes how to call subprograms written in Domain Pascal or Domain/C.
Chapter 8	Contains an overview of the I/O resources available to Domain FORTRAN programmers.
Chapter 9	Describes compiler diagnostic messages and how to handle them.

Appendix A	Lists Domain FORTRAN keywords.
Appendix B	Contains an ISO Latin-1 table, which includes ASCII characters.
Appendix C	Lists the FORTRAN intrinsic functions.
Appendix D	Describes how to get the best floating-point performance on MC68040-based workstations.

Summary of Technical Changes

The *Domain FORTRAN Language Reference* documents technical changes to Domain FORTRAN that have been made since the last printing of this manual in July 1988. These and other changes are marked by change bars in the margin. Specifically, the manual documents the following new features:

- Alternate syntax for pointer extension (Section 3.9)
- Bitwise operators: **.and.**, **.not.**, and **.or.** (Section 4.2)
- Compiler directives: **%begin_inline**, **%end_inline**, **%begin_noinline**, **%end_noinline**, and **%line** (Chapter 4)
- Compiler variants for cross-compiling between MC680x0-based machines and the Series 10000 workstation (Section 6.2)
- **-cpu** arguments: **m68k**, **a88k**, **mathchip**, **mathlib**, and **mathlib_sr10** (subsection 6.5.8)
- Data type: **byte** (Section 3.3)
- **f77** compiler options: **-A**, **-T1**, and **-W** (Section 6.4)
- **ftn** compiler options: **-alnchk** (subsection 6.5.2), **-[no]bounds_violation** (subsection 6.5.4), **-bx** (subsection 6.5.5), **-mp** (subsection 6.5.13), **-natural** (subsection 6.5.24), **-nclines** (subsection 6.5.25), **-overlap** (subsection 6.5.27), and **-[n]prasm** (subsection 6.5.27)
- Intrinsic functions: **dfloat**, **dreal**, **lshift**, and **rshift** (Appendix C)
- **O** (octal) edit descriptor (Chapter 4)
- Defining user interface with Open Dialogue and Domain/Dialogue (subsection 6.10.3)
- **-opt 3** optimization: software pipelining (subsection 6.5.26)
- Program development with NFS (Section 6.11)

- Statements: **atomic, discard, implicit none,** and **options** (Chapter 4)

In addition, the manual includes new sample programs to illustrate new and existing features and adds clarifying or new information on the following topics:

- Aligning data in **common** blocks for the Series 10000 workstation (Chapter 4 and subsection 6.5.24)
- **–cpu** arguments and performance (subsection 6.5.8)
- Domain files: external and internal (Section 8.2)
- **entry** statement (Chapter 4 and subsection 5.2.3)
- Floating–point representation (subsection 3.4.2) and performance (Appendix D)
- **–[n]frnd** compiler option (subsection 6.5.13)
- Passing arguments between FORTRAN and Domain/C (Subsections 7.7.3, 7.7.4, 7.7.5, and 7.7.6)
- Saving data in static storage with **save** statement (Chapter 4) and **–save** option (subsection 6.5.30)
- Runtime errors (Section 9.3)
- **–[n]uc** compiler option (subsection 6.5.34)
- Reorganized Chapters 5 and 6 for clarity

Related Manuals

The file **/install/doc/apollo/os.v.***latest software release number*__**manuals** lists current titles and revisions for all available manuals.

For example, at SR10.3 refer to **/install/doc/apollo/os.v.10.3__manuals** to check that you are using the correct version of manuals. You may also want to use this file to check that you have ordered all of the manuals that you need.

(If you are using the Aegis environment, you can access the same information through the Help system by typing **help manuals.**)

Refer to the *Apollo Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) for a complete list of related documents. For more information related to Domain FORTRAN, refer to the following documents:

- *Aegis Command Reference* 002547
- *Analyzing Program Performance with Domain/PAK* 008906
- *BSD Command Reference* 005800
- *Creating User Interfaces with Open Dialogue* 011167
- *Customizing Open Dialogue* 011166
- *Domain Distributed Debugging Environment Reference* 011024
- *Domain Floating-Point Guide* 015853
- *Domain Graphics Primitives Resource Call Reference* 007194
- *Domain Pascal Language Reference* 000792
- *Domain/C Language Reference* 002093
- *Domain/Dialogue User's Guide* 004299
- *Domain/OS Call Reference* 007196
- *Domain/OS Programming Environment Reference* 011010
- *Engineering in the DSEE Environment* 008790
- *HP Concurrent User's Guide* 017996
- *HP/OSF Motif Style Guide* 98794-90007
- *Open Dialogue Reference* 012807
- *Programming with Domain GPR* 005808
- *Programming With Domain/OS Calls* 005506
- *Series 10000 Programmer's Handbook* 011404
- *SysV Command Reference* 005798
- *Using NFS on the Domain Network* 010414

You can order Apollo documentation by calling **1-800-225-5290**. If you are calling from outside the U.S., you can dial **(508) 256-6600** and ask for **Apollo Direct Channel**.

Does This Manual Support Your Software?

This manual was released with Domain FORTRAN Version 10.8. It runs on Software Release 10.0 or a later version of Domain/OS. To verify which version of operating system software you are running, type:

bldt

If you are running Domain/IX on a release of the operating system earlier than SR10.0, type:

/com/bldt

To check the version of Domain FORTRAN, type:

/com/ftn -version

If you are using a later version of software than that with which this manual was released, use one of the following ways to check if this manual was revised or if additional manuals exist:

- Read Chapter 3 of the release document that shipped with your product. The release document is online:

  /install/doc/apollo/ftn.v.10.8.m__notes

  /install/doc/apollo/ftn.v.10.8.mpx__notes

  /install/doc/apollo/ftn.v.10.8.p__notes

  /install/doc/apollo/ftn.v.10.8.pmx__notes

- Check with your system administrator if you cannot find the release document.
- Telephone **1-800-225-5290**. If you are calling from outside the U.S., dial **(508) 256-6600** and ask for **Apollo Direct Channel**.
- Refer to the lists of manuals described in the preceding section, "Related Manuals."

To determine which of two versions of the same manual is newer, refer to the order number that is printed on the title page. Every order number has a 3-digit suffix; for example, **-A00**. A higher suffix number indicates a more recently released manual. For example, a manual with suffix **-A02** is newer than the same manual with suffix **-A01**.

Problems, Questions, and Suggestions

If you have any questions or problems with our hardware, software, or documentation, please contact either your HP Response Center or your local HP representative.

You may call the Tech Pubs Connection with your questions and comments about our documentation:

- In the USA, call 1-800-441-2909
- Outside the USA, call (508) 256-6600 extension 2434

The recorded message that you will hear when you call includes information about our new manuals.

You may use the Reader's Response Form at the back of this manual to submit comments about our documentation.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user-supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
sample user input	In samples, information that the user enters appears in color.
Domain extensions	Domain-specific features of FORTRAN appear in color.
`output`	`System output appears in this typeface.`
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices.

< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.
. . .	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
	Change bars in the margin indicate technical changes from the last revision of this manual. Because Appendix D is completely new to this revision, it does not have change bars.
	This symbol indicates the end of a chapter.

Contents

Chapter 1 Introduction

Chapter 2 Blueprint of a Program

Chapter 3 Data Types

Chapter 4 Code

Chapter 5 Subprograms

Chapter 6 Program Development

Chapter 7 Cross–Language Communication

Chapter 8 Input and Output

Chapter 9 Diagnostic Messages

Appendix A Domain FORTRAN Keywords

Appendix B ISO Latin-1 Table

Appendix C Summary of Intrinsic Functions

Appendix D Optimizing Floating–Point Performance on MC68040–Based Domain Workstations

Figures

Tables

Chapter 1

Introduction

Domain FORTRAN is an extended implementation of the full FORTRAN language as defined in ANSI publication X3.9-1978*.

You should be somewhat familiar with FORTRAN before attempting to use this manual. If you are not, please consult a FORTRAN tutorial. (The Preface includes a list of some good tutorials.) If you are familiar with FORTRAN, you should be able to write programs in Domain FORTRAN after reading this manual.

1.1 A Sample Program

The best way to get started with Domain FORTRAN is to write, compile, and execute a simple program. Figure 1-1 shows a simple program that you can use to get started. You are welcome to type in this program yourself, but this program is available online. (See the "Online Sample Programs" section of this chapter for details.)

```
*  A simple program to try out.
      program easy
      integer*2 i
      integer*4 j
      print *, 'Enter an integer:'
      read *, i
      j = i * 2
      print 10, i, j
10    format ('When you double ', I2, ' you get ', I3)
      end
```

Figure 1-1. Sample Program

* American National Standard Programming Language FORTRAN, American National Standards Institute, New York, N.Y., 1978.

1.2 Two Ways to Call FORTRAN

There is only one Domain FORTRAN compiler; however, it can be called by invoking either of the following two files:

```
f77
ftn
```

Use **f77** to invoke FORTRAN in UNIX environments. Use **ftn** to invoke FORTRAN in an Aegis environment. The two commands support different options and different default behavior. See Chapter 6 for a complete discussion of these differences.

If you want to invoke **ftn** from a UNIX shell, use the full pathname:

/usr/apollo/lib/ftn

If you want to invoke **f77** from an Aegis shell, use one of these full pathnames:

/bsd4.3/usr/bin/f77

/sys5.3/usr/bin/f77

Thus, you can compile the sample program in Section 1.1 in either of two ways:

- If you use **f77**, your source filename must include the **.f** suffix. Suppose you store the sample program in a file that you name **getting_started.f**. You can compile it by entering the full name of the source file, including the **.f** suffix. For example,

  ```
  $ f77 getting_started.f
  ```

  The above command line produces an executable object file named **a.out.**

- If you use **ftn**, then you may optionally use the **.ftn** suffix for your source filenames. Furthermore, if the file has this suffix, you may omit the suffix on the command line. Thus, suppose you store the sample program in a file that you name **getting_started.ftn**. You can use either of the following command lines:

  ```
  $ ftn getting_started.ftn
  $ ftn getting_started
  ```

  Each of the **ftn** command lines above produces an executable object file named **getting_started.bin.**

  If the file has a suffix other than **.ftn**, you must specify the complete filename.

To run the object files produced by the compiler, just enter the name of the executable file. Table 1-1 summarizes the whole process.

Table 1-1. Compiling and Executing a Simple Program

Under Aegis environment	Under UNIX environment

`$ ftn getting_started.ftn`
`no errors, no warnings, no informational messages`
 	
`$ getting_started.bin`
` Enter an integer:`
`15`
`When you double 15 you get   30`
	`% f77 getting_started.f`
	`no errors, no warnings, no informational messages`
	
	`$ a.out`
	` Enter an integer:`
	`15`
	`When you double 15 you get   30`

NOTE: If you use the suffix **.f** for all of your FORTRAN source files, you can compile them with either **f77** or **ftn**. The only difference is that, if you compile a file called *prog_name*.**f** with **ftn**, the executable file will be named *prog_name*.**f.bin**. If you use the **.f** suffix for source files compiled with **ftn**, be sure to specify the full pathname for the source file on the command line.

1.3 Online Sample Programs

Many of the programs from this manual are stored online, along with sample programs from other Domain manuals. These programs come automatically with the Domain FORTRAN product. They illustrate features of the Domain FORTRAN language, and demonstrate programming with Domain graphics calls and system calls. You retrieve these on-line sample programs with the **getftn** utility.

1.3.1 What You Get When You Install Sample Programs

When you (or your system administrator) load the Domain FORTRAN product, the install procedure asks if you want to install sample programs. We recommend that you answer yes. If you answer yes, the install program stores the following three files in the **/domain_examples/ftn_examples** directory:

examples — This is a master file containing all the sample FORTRAN programs.

list_of_examples — This is a help file describing all the sample FORTRAN programs.

getftn — This is the utility that retrieves the sample programs out of the examples file.

Since the sample programs take up a lot of disk space (more than 350 KB), we recommend that you install the sample programs in only one site on the network and have users link to them.

1.3.2 Creating Links to the Sample Programs

If you want to be able to invoke **getftn** from any directory on the system, you must create the appropriate links. The links differ according to your type of environment.

If you are using the Aegis environment, you can set up the following link:

$ crl ~/com/getftn *//node_name*/domain_examples/ftn_examples/getftn

If you are in a UNIX environment, you can set up the following link:

% ln -s *//node_name*/domain_examples/ftn_examples/getftn *a_dir*/getftn

//node_name is the name of the disk where the examples are stored and *a_dir* is the name of a directory in your path.

If *//node_name* is not your node, then you should also create the following link. (Pick one of the following depending on your operating system environment.)

$ crl /domain_examples/ftn_examples *//node_name*/domain_examples/ftn_examples

% ln -s *//node_name*/domain_examples/ftn_examples /domain_examples/ftn_examples

where *//node_name* is the name of the disk where the examples are stored.

After creating the links, you can invoke **getftn** from any directory.

> **NOTE:** An alternative to creating links is to set your working directory to the *//node_name*/**domain_examples/ftn_examples** directory and invoke **getftn** from there.

1.3.3 Invoking getftn

You invoke **getftn** in the same manner regardless of the operating system environment.

There are two ways to invoke **getftn**. The first, and simplest, is to specify its name on any shell command line. For example, in an Aegis environment:

$ getftn

After you invoke **getftn**, the utility prompts you for the appropriate information. If you want to rename the program (for example, to change the suffix from the default **.ftn** to **.f**), enter the new name when **getftn** prompts you for it.

The second way to invoke **getftn** is to indicate the desired information on the command line itself. To do so, issue a command with the following format:

getftn *sample_program_name output_file_name*

For example, the following command line finds the sample program named **getting_started** and writes it to pathname **//dolphin/fortran_programs/getting_started.f**:

getftn getting_started //dolphin/fortran_programs/getting_started.f

If you do not specify an output file, the program will be listed on standard output.

For syntactic information on using **getftn**, issue the following command:

getftn -usage

1.4 Overview of Domain FORTRAN Extensions

Domain FORTRAN supports many extensions to ANSI 1978 standard FORTRAN. The purpose of this section is to provide an overview of these extensions. Extensions to the standard are marked in color like this or are noted explicitly in text as an extension. Naturally, the more you take advantage of Domain FORTRAN extensions, the less portable your code will be. Therefore, if you are very concerned with portability, you should avoid using the features described in this section.

NOTE: Beginning with the SR10 version, Domain FORTRAN is case-sensitive for filenames. Be sure that any filenames you refer to are case-correct.

1.4.1 Extensions to Program Organization

Chapter 2 describes the organization of a Domain FORTRAN program contained within one file. With Domain FORTRAN's extensions, you can:

- Specify an underscore (_) or dollar sign ($) in an identifier.
- Declare identifiers that are up to 4096 characters long.
- Use flexible lines to write your code, rather than columnar spacing.

- Specify integers in any of three bases: octal, decimal, and hexadecimal.
- Specify line comments in three ways and use a default or user–defined character to designate in–line comments.
- Use uppercase and lowercase letters interchangeably for variable names.
- Use up to 32,767 continuation lines.
- Intersperse **data** statements with specification statements.

1.4.2 Extensions to Data Types

Chapter 3 describes the data types that Domain FORTRAN supports. As extensions to the standard, Domain FORTRAN supports these additional data types:

- **byte**
- **integer*2**
- **logical*1**
- **logical*2**
- **logical*4**
- **complex*16** (this is the same as **double complex**)

1.4.3 Extensions to Code

Chapter 4 describes the action portion of your program. Domain FORTRAN supports a number of extensions to statements. You can:

- Mix character and noncharacter data types in the same **common** area.
- Use the **pointer** statement with routines written in other languages that return pointers.
- Use extended **do** ranges.
- Specify a **do while** loop.
- Use the **A** (character) and **Z** (hexadecimal) format specifier for any data type.
- Use the FORTRAN 66 statements **encode** and **decode** and the Hollerith syntax.
- Use the **namelist** statement to define a synonym for a list of variables and array names.

- Open a file with any of four additional statuses: 'append', 'write', 'readonly', or 'overwrite'.
- Specify a variety of compiler directives that enable features like include files and conditional compilation.
- Use the **O** format specifier to edit any type of data into octal format.
- Use the **discard** statement to call a function as a subroutine.
- Use the **implicit none** statement to override the default typing rules.
- Use the **options** statement to insert compiler options in the source code.

1.4.4 Extensions to Subprograms

Chapter 5 describes subroutines, functions, statement functions, and block data subprograms. The chapter also lists the additional intrinsic functions available, and documents the fact that Domain FORTRAN supports recursive subprograms.

1.4.5 Extensions to Program Development

Chapter 6 explains how to compile, bind, debug, and execute your program. Program development tools are an implementation-dependent feature of FORTRAN; that is, there is no standard for these tools.

1.4.6 Cross-Language Communication

Chapter 7 explains how to write a program that accesses code or data written in other programming languages. This entire chapter describes implementation-dependent features.

1.4.7 Extensions to I/O

Chapter 8 describes input and output from a Domain FORTRAN programmer's point of view. Domain FORTRAN supports all the standard I/O procedures. As an extension to the standard, Domain FORTRAN gives you easy access to the operating system's I/O and formatting system calls.

1.4.8 Compiler Messages

Chapter 9 lists compile-time and run-time compiler messages and explains how to deal with them. Compiler messages are an implementation-dependent feature of FORTRAN.

Chapter 2

Blueprint of a Program

This chapter describes the building blocks and organization of a Domain FORTRAN program.

2.1 Building Blocks of Domain FORTRAN

This section defines identifiers, integers, real numbers, complex numbers, logicals, character strings, and comments. It also explores case sensitivity, the significance of blanks, column conventions, statement labels, and the spreading of source code across multiple lines.

2.1.1 Identifiers

We refer to identifiers throughout the manual. An **identifier** is any sequence of characters that meets the following criteria:

- The first character is a letter (ASCII decimal values 65 through 90 and 97 through 122)
- The remaining characters are any of the following:

  A...Z and a...z (ASCII decimal values 65 through 90 and 97 through 122)
  0...9 (ASCII decimal values 48 through 57)
  _ (underscore) (ASCII decimal value 95)
  $ (dollar sign) (ASCII decimal value 36)

As an extension to standard FORTRAN, which limits identifiers to six characters, Domain FORTRAN allows variable names and other identifiers to have up to 4096 characters; all of the characters are significant.

By default, the compiler is case-insensitive for identifiers—uppercase and lowercase characters in identifiers are treated the same. You can tell the compiler to be case-sensitive for identifiers by using the **-U** option when you compile your source code. Refer to Sections 6.4 (**f77**) and 6.5 (**ftn**) for details about this option.

Here are some examples of valid and invalid names:

Valid	Invalid	
FOUR$PRICE	4TRAN	{starts with a digit}
yo_yo	Ps&Qs	{contains an ampersand}
T	$sign	{starts with a dollar sign}
STREAM_$GET_REC		

2.1.2 Integers

The first character of an integer must be a positive sign (+), a negative sign (–), or a digit. Each successive character must be a digit. (Refer to Section 3.3 for the ranges of the different integer data types.)

FORTRAN assumes a default of base 10 for integers. If you want to express an integer in octal or hexadecimal, use the following syntax:

base#value

where *base* indicates the base, either 8 or 16, and *value* is the number in that base. For example, 8#100, 16#40, and 64 all represent the same value. When expressing a number in hexadecimal, use the letters A through F (or a through f) to represent digits with the values 10 through 15.

Octal and hexadecimal constants can be used anywhere an integer is valid. For instance, you can assign an octal value to an integer variable, but you cannot represent statement label numbers in octal or hexadecimal.

The following integer assignments illustrate how to indicate the base of the constant:

```
half_life = 5260       {default     (base 10)}
hexagrams = 16#1c6     {hexadecimal (base 16)}
wheat = 8#723          {octal       (base 8 )}
```

2.1.3 Real Numbers

Domain FORTRAN supports both real and double-precision numbers. A real number is a floating-point value, consisting of a whole number, a decimal fraction, or both. A double-precision number is similar, except with twice the accuracy.

Both real and double-precision numbers can be expressed either as a string of digits containing a decimal point or in expanded notation. To express a real number in expanded notation (powers of 10), separate the mantissa from the exponent with the letter e or E, as in the following examples:

```
5.2      means  +5.2
5.2E0    means  +5.2
-5.2E3   means  -5200.0
5.2E-2   means  +0.052
```

To express a double-precision number in expanded notation (powers of 10), separate the mantissa from the exponent with the letter d or D, as in the following examples:

```
5.2D0    means  +5.2        {in 64 bits}
5.2D-5   means  +0.000052   {in 64 bits}
```

2.1.4 Complex Numbers

Domain FORTRAN supports two different kinds of complex numbers: **complex** and **double complex**. A **complex*8** is another name for **complex**, and **complex*16** is another name for **double complex**. A valid **complex** or **double complex** number must take the following format:

(*number*, *number*)

You must specify both parts of the complex number, separate the parts with a comma, and enclose the entire entity in parentheses. Although *number* can be any valid integer or real number, only **double complex** types can include double-precision numbers. For example, these are valid **complex** numbers:

```
(6.E2,5.2993)
(77.562, 4.E3)
(0,0)
```

And these are valid **double complex** numbers:

```
(6.D2,5.2993)
(77.562, 4.D3)
(0,0D)
```

2.1.5 Logicals

Logical, **logical*1**, **logical*2**, and **logical*4** constants can take one of two values: **.true.** or **.false.**

2.1.6 Character Strings

Domain FORTRAN character string conventions vary somewhat depending on whether you compile your source code using **ftn** or **f77**. We describe first the conventions common to both commands.

In Domain FORTRAN, a string is a sequence of characters that starts and ends with a single quote (') or with double quotes ("). Unlike an identifier, a string can contain any printable character. Here are some sample strings:

```
"This is a character string surrounded by double quotes."
'This is a character string surrounded by single quotes.'
'18'
'b[2~{q^%pl'
```

To include a single quote in a string, type it twice; for example:

```
'I can''t do it.'
'Then don''t try!'
```

> **NOTE:** Within a string, Domain FORTRAN treats an inline comment delimiter as an ordinary character rather than as a comment delimiter.

When you compile with **f77**, or **ftn** with the **-uc** option, you activate the following conventions, in addition to those described above:

- The compiler and the I/O system recognize the following backslash escapes:

  \n newline
  \t tab
  \b backspace
  \f form feed
  \0 null
  \' single quote (does not terminate a string)
  \" double quote (does not terminate a string)
  \\ a single backslash(\)
  x where *x* is any other character

- If a string begins with one variety of quote mark, the other variety may be embedded within it without using the repeated quote or backslash escapes. For example, the following strings will compile correctly:

```
"I can't do it."
"Then don't try!"
'She said "Hi."'
```

2.1.7 Comments

Comment lines have the letter C or an asterisk (*) in column 1. Comments typically provide explanatory text or mark various sections in a program. FORTRAN ignores comment lines and blank lines (regarding them as comment lines). Domain FORTRAN also supports lowercase (c) as a comment indicator.

As an extension to the ANSI standard, Domain FORTRAN allows inline comments—comments enclosed in braces { }, or another character that you select, which appear at the end of a statement line. (The second brace, }, is optional.) In-line comments cannot extend more than one line. Refer to Section 6.5.19 for information about defining your own comment character.

For example, here are some valid comments:

```
* This is a comment line.
C This is also a comment line.
      i=1          {Set i to 1. This is an in-line comment.}
      i=1          !Set i to 1. This is an in-line comment
                   !with a user-defined comment character.
```

Note that FORTRAN comments cannot stretch across multiple lines; for example, the following is an invalid comment:

```
* This invalid comment
  stretches across
  multiple lines - but it's illegal!
```

But:

```
* This
* is
* legal
```

2.1.8 Case-Sensitivity

While standard FORTRAN allows only uppercase letters in keywords and identifiers, Domain FORTRAN is case-insensitive for these entities: it allows both uppercase and lowercase letters.

For example, the following three uses of the keyword **end** are equivalent:

```
END
end
End
```

However, Domain FORTRAN *is* case-sensitive to strings and to filenames. For example, the following two character strings are *not equivalent:*

```
'The rain in Spain'
'THE RAIN IN SPAIN'
```

And the following filenames are *not equivalent*:

```
'myfile'
'MYFILE'
```

Be sure that filenames are case-correct.

> **NOTE:** You can tell the compiler to be case-sensitive for identifiers by using the **-U** option when you compile your source code. Refer to Sections 6.4 (**f77**) and 6.5 (**ftn**) for details about this option.

2.1.9 Blank Characters

Blanks are insignificant in FORTRAN, except in character strings. This means that FORTRAN doesn't care if you leave out the blank between two keywords (for example, **goto** or **enddo**). It also means it is valid to stretch out keywords or identifiers, although doing so may make your code more difficult to read. For instance, the following is acceptable in FORTRAN:

```
i n t e g e r * 4    s p a c e d_ o u t
```

although it's considerably easier to read if you write it this way:

```
integer*4 spaced_out
```

Blanks *are* significant in character strings. For example, the following two strings are *not equivalent*:

```
'Live long and prosper'
'L i v e   l o n g   a n d   p r o s p e r'
```

2.1.10 Column Conventions

Both **f77** and **ftn** accept source code that is in standard format. The standard expects source code—except for comment lines—to be in 72–column format. The rules for this format are as follows:

- The first five characters are the statement label.
- The sixth character is the continuation character.
- The next 66 characters are the body of the line.
- If there are fewer than 72 characters, the compiler pads with blanks.
- The compiler ignores characters after the 72nd character.
- Column 1 can contain a comment character (*, C, or c) or the beginning of a statement label.
- You can place the letter D or d in column 1 to mark a line for conditional compilation. Such lines are ignored during compilation unless you compile with the –cond option, which is described in Chapter 6. Column 1 can also contain the beginning of a compiler directive.
- You can use columns 73 through 80 for sequencing information.
- You can use a hard tab character in one of the first six positions of a line to signal the end of the statement number and continuation part of the line. Any characters entered after the hard tab character form the body of the line.

  NOTE: Because Domain FORTRAN ignores comments (both comment lines and inline comments), comments can extend past the 80th column.

When you compile with the **–ff** option, you activate the following column conventions, in addition to the standard described above:

- You can use the ampersand character (&) in column 1 to denote a continuation line—the remaining characters form the body of the line.
- Lines can be up to 1023 characters long.

2.1.11 Statement Labels

Any FORTRAN statement may have a unique identifying label, which can consist of up to five digits. For example, in this statement

```
100   format (3F6.2, 1X, I4)
```

100 is the statement label. FORTRAN ignores leading zeros and embedded and trailing blanks in statement labels. A statement label can start in any column from 1 through 5, but it cannot extend past column 5.

2.1.12 Spreading Source Code Across Multiple Lines

Both **ftn** and **f77** activate the following conventions for spreading source code across multiple lines:

- If you want to continue a FORTRAN statement over more than one line, you must put a continuation character in column 6. When column 6 contains a continuation character, columns 1 through 5 must be blank.

- You can use any character except blank or zero as a continuation character. This is a valid use of a continuation character:

```
12345678901234567890 . . .  {ruler to help you count columns}
      open (10, file='my_file', iostat=open_stat,
     +      recl=50, status='old')
```

- No more than 32,767 continuation lines are allowed per statement.

- Because blanks are not significant in FORTRAN—except in character strings—it is legal to spread keywords and identifiers across multiple lines. For example, you can do this:

```
12345678901234567890 . . .  {ruler to help you count columns}
      do while (this__long_section_name .ne.
     $          the_last_very_long_section_name)
```

- You can not continue compiler directives across lines.

- You can use a hard tab character in one of the first six positions of a line to signal the end of the statement number and continuation part of the line. Any characters entered after the hard tab character form the body of the line.

In addition, when you compile using the **-ff** option, you activate the following UNIX source file formatting feature:

- You can use the ampersand character, &, in the first position of a line to indicate a continuation line—the remaining characters form the body of the line.

Refer to Section 6.5.12 for details about this compiler option.

2.2 Organization

You can write a Domain FORTRAN program in one file or across several files. This section explains the proper structure for a program that fits into one file. Chapter 6 explains how to compile and bind a program that is in one file or that is spread across multiple files.

2.2.1 Program Units

In FORTRAN, the term **program unit** refers to any main program, subroutine, function, or block data subprogram. If you write a program that contains a main program and two subroutines, the program is said to contain three program units.

2.2.2 Statement Order

Domain FORTRAN conforms to the ANSI standard for statement order, as follows:

- The **options** statement, if present, must appear before a **program**, **function**, **block data**, or **subroutine** statement because it determines the compilation of the program unit. If it appears elsewhere, it is an error.

- The **program** statement, if present, must be the first statement of a main program unit. **Subroutine** and **function** must be the first statements of subroutine and function subprograms, respectively. **Block data** must be the first statement of a block data subprogram.

- Specification, type, and pointer statements must precede all statement function statements and executable statements. Specification statements are listed in Figure 2-1, and type statements are listed in Figure 2-2.

  NOTE: As an extension to the ANSI standard, Domain FORTRAN allows you to intersperse **data** statements with other specification statements.

- The **implicit** statement must precede all other specification statements except **parameter**. A given **parameter** statement must precede all statements that reference the constant that the **parameter** statement defines.

- Statement function statements must precede all executable statements.

- Executable statements follow next. Figure 2-3 lists the executable statements.

- **Format** statements may appear anywhere after a program or subprogram heading. An **entry** statement may appear anywhere in an executable subprogram except in a **do** loop or an **if** block.

- Comment lines and compiler directives may appear anywhere.
- The **end** statement must be the last statement in every program unit.

[illegible]	**external**	options
common	**implicit**	**parameter**
data	implicit none	**save**
dimension	**intrinsic**	
equivalence	**namelist**	

Figure 2-1. FORTRAN Specification Statements

[illegible]	**double precision**	logical*2
character	**integer**	logical*4
complex	**integer*2**	**real**
complex*8	**integer*4**	**real*4**
complex*16	**logical**	**real*8**
double complex	logical*1	

Figure 2-2. FORTRAN Type Statements

assign	do while	**inquire**
Assignment statements: arithmetic, character, logical	**else if . . . then**	**open**
backspace	encode	**pause**
call	**end**	**print**
close	end do	**read**
continue	**end if**	**rewind**
[illegible]	**endfile**	**stop**
[illegible]	**go to**	**write**
do	**if**	

Figure 2-3. Executable FORTRAN Statements

The following program is a labeled example of the structure of a Domain FORTRAN program. The program is available online and is named **labeled**.

```
      program labeled            {Optional program heading.         }

*   Specification and type statements for the main program unit.
*   Notice that these statements can be in any order.

       integer*4   num, sqr_num

       logical again
       data    again /.true./  {Data statement can be interspersed }
                               {with type and specification statements.}
       character*1 answer
       common /squares/ num, sqr_num


*   Executable statements in the main program unit.

       do while (again)
           print *, 'Enter an integer:'
           read (*, *) num
           call my_sqr()
           print 10, sqr_num
           print *, 'Again? Y or N'
           read (*, '(A1)') answer
           if ((answer .eq. 'N') .or. (answer .eq. 'n')) again = .false.
       end do

*   Format statement may appear anywhere in the program unit.

10     format (' The number squared is:', I5)

       end                       {End must be the last statement}
                                 {in each program unit.         }

************************************************************************
*   Subroutine must be the first statement in a subroutine subprogram.
       subroutine my_sqr()

*   Specification and type statements for subprogram my_sqr.
       integer*4 sub_num, sub_sqr_num
       common /squares/ sub_num, sub_sqr_num

*   Executable statements in the subroutine subprogram.
       sub_sqr_num = sub_num * sub_num

       end                       {Again, end must be the last     }
                                 {statement in each program unit.}
```

2.2.3 Program Heading

Your program may contain a program heading. The program heading has the following format:

program *name*

where *name* is the name you give this main program unit. If you omit the program heading, Domain FORTRAN by default names the main program unit according to the following rules:

- If you compile with **ftn**, the default name for the program unit is **$MAIN**.
- If you compile with **f77**, or **ftn** using the **-uc** option, the default name for the program unit is **main__**. (Note that there are two underscores at the end of the name **main__** .)

name must be an identifier. This identifier has no meaning within the program, but is used by the binder, the librarian, and the loader. (See the *Domain/OS Programming Environment Reference* for details about these utilities.)

2.2.4 Declarations and Specifications

The declarations and specifications part of a program is optional. It can consist of zero or more data type declaration and specification statements. Figure 2-1 lists Domain FORTRAN's available data type and specification statements.

2.2.4.1 Data Type Declarations

You declare variables with data type declarations. A declaration has two components: a data type and a name. The format for declarations is

data_type1 *identifier_list1*
.   .
.   .
.   .
data_typeN *identifier_listN*

An *identifier_list* consists of one or more identifiers separated by commas. Each identifier in the *identifier_list* has the data type of *data_type*. *Data_type* must be a predeclared Domain FORTRAN data type; that is, **character**len***, **complex**, complex*8, complex*16, double complex, **double precision**, byte, **integer**, **integer*2**, **integer*4**, logical, logical*1, logical*2, logical*4, **real**, **real*4**, or **real*8**.

For example, consider the following data type declarations:

```
integer        counter, x, y
real*4         angles
character      a_letter
logical        evil
character*15 names(10)     {Declare the 10-element character array }
                           {names. Each element has a length of 15.}
character*20 mystery_guest
```

In the preceding example, note that **counter**, **x**, and **y** are three variables that have the same data type (**integer**).

2.2.5 Action Part of a Main Program Unit

The action part of a main program unit starts with the first executable statement and finishes with the keyword **end**, as in the following example:

```
int = int * 100
print *, int
end
```

We detail Domain FORTRAN statements in Chapter 4.

2.2.6 Subprograms

A program can contain zero or more subprograms. There are three types of subprograms in Domain FORTRAN—subroutines, functions, and block data subprograms. A subprogram consists of three parts—a subprogram heading, an optional declaration and specification part, and an action part. Subprograms are described in detail in Chapter 5.

2.2.6.1 Subprogram Heading

Subprogram headings take the following format:

subroutine *name* [*argument_list*]

or

[*data_type*] **function** *name* [*argument_list*]

or

block data *name*

where:

- *name* is an identifier. You call the subprogram by this name.
- *argument_list* is optional. It is here that you declare the names of the dummy arguments that the subprogram expects from the caller. We detail the *argument_list* in Chapter 5.
- *data_type* is optional and specifies the data type of the value that the function returns. You can either specify an explicit data type, or use FORTRAN's default naming conventions.

The difference between a subroutine and a function is that the *name* of a subroutine is simply a name, but the *name* of a function is itself a variable with its own data type. You must assign a value to this variable at some point within the action part of the function. (It is an error if you don't.) You cannot assign a value to the name of a subroutine. (It is an error if you do.)

A block data subprogram is used exclusively to assign values to variables in **common** blocks. See Chapter 4 for information on **common** blocks.

2.2.6.2 Declaration Part of a Subprogram

The optional declaration part of a subprogram follows the same rules as the optional declaration part in the main program under the program heading. The variables are local to the subprogram that declares them (refer to Section 2.3).

2.2.6.3 Action Part of a Subprogram

The action part of a subprogram is almost identical to the action part of a main program unit. Both contain executable Domain FORTRAN statements, and both finish with **end**. However, a subprogram may contain the keyword **return**, which instructs FORTRAN to return control to the program unit that called this subprogram. For example, consider the following sample action part of a subprogram:

```
int = int * 100
print *, int
return
end
```

2.3 Scope of Variables

The variables in each program unit are local to that particular unit *only*, unless you define **common** blocks that state that variables are shared across units. For example, consider the following program:

```
      program scope
      integer*2 num, shared_num
      common /group/ shared_num
      num = 30
      shared_num = 100
      print *, 'In the main program, num equals ', num
      print *, 'And shared_num equals ', shared_num
      call x()
      end
************************************************************************
      subroutine x()
      integer*2 num, shared_num
      common /group/ shared_num
      print *, ' '
      print *, 'In the subroutine, num now equals ', num
      print *, 'But shared_num still equals ', shared_num
      end
```


This program is available online and is named **scope**. If you run the program, you get results like the following:

```
In the main program, num equals  30
And shared_num equals  100
In the subroutine, num now equals  -32582
But shared_num still equals  100
```

Even though the variables **num** have the same characteristics in the main program and subroutine, they are two different variables. The subroutine does not assign an explicit value to its **num**, so when you execute the program, FORTRAN assigns to **num** whatever value happens to be in the memory location for the variable. In this case, it's -32582, but it might be something else if the program were run another time or on another machine.

If you do want to share variables across program units, you must declare **common** areas. In this example, the variable **shared_num** is part of the group **common** block. This means that when the main program and subroutine refer to **shared_num**, they actually are referring to the same variable. The program output demonstrates the correspondence.

See the listing for **common** in Chapter 4 for more information about **common** blocks.

Chapter 3

Data Types

This chapter explains Domain FORTRAN data objects. It tells you how to declare variables using the Domain FORTRAN data types and how Domain FORTRAN represents data types internally.

3.1 Data Type Overview

Domain FORTRAN supports the following simple data types:

- Integers—Domain FORTRAN supports the predeclared integer data types byte, **integer**, integer*2, and integer*4.
- Real Numbers—Domain FORTRAN supports the predeclared real-number data types **real**, real*4, real*8, and **double precision.**
- Boolean/Logical—Domain FORTRAN supports the predeclared data types **logical**, logical*1, logical*2, and logical*4.
- Complex—Domain FORTRAN supports the predeclared data types **complex**, complex*8, complex*16 and double complex.
- Character—Domain FORTRAN supports the predeclared data type **character.**

In addition to the simple data types, Domain FORTRAN supports arrays of the simple data types.

As an extension to the ANSI standard, Domain FORTRAN supports a pointer statement that gives you access to the pointers returned by programs written in other languages.

The sections that follow describe the data types listed above, as well as the **pointer** statement. Before you learn how to explicitly declare variables of the individual data types, however, you should know about FORTRAN's default naming conventions.

3.2 Naming Conventions for Variables

It is not always necessary to explicitly declare variables in FORTRAN. By default, FORTRAN considers any undeclared variable beginning with a letter between I and N to be an integer, while an undeclared variable beginning with any other letter is a real number.

You can change the default naming conventions with the **implicit** statement so that a variable beginning with the letter "a" (for example) is implicitly typed as an integer. Also, you can use the **implicit none** statement to enforce explicit variable declaration. Both statements are fully described in Chapter 4.

The following sections describe how to explicitly declare each data type and how Domain FORTRAN represents the types internally.

3.3 Integers

This section explains how to declare variables as integers and how Domain FORTRAN represents integers internally.

3.3.1 Declaring Integer Variables

Domain FORTRAN supports the following four predeclared integer data types:

[illegible]

[illegible]

[illegible]

Integer — By default, **integer** is equivalent to **integer*4**. However, if you compile with the **–i*2** switch (described in Chapter 6), all **integers** become **integer*2** variables.

For example, consider the following integer declarations:

```
integer   high, low, middle
byte      little_int
integer*2 medium_int
integer*4 big_int
```

In this declaration, **high**, **low**, and **middle** are 32-bit integers unless you compile with the **-i*2** option (see subsection 6.5.14). In that case, they are 16-bit integers. If you know that you always want an integer to be a certain size, you should use the explicit **byte**, **integer*2**, or **integer*4** designations.

3.3.2 Internal Representation of Integers

Domain FORTRAN represents an 8-bit integer (**byte**) as one byte, as shown in Figure 3-1. Bit 7 contains the most significant bit (MSB) and Bit 0 contains the least significant bit (LSB).

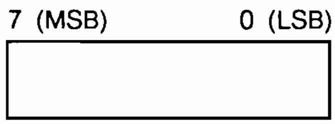

Figure 3-1. 8-Bit Integer Format

Domain FORTRAN represents a 16-bit integer (**integer*2**) as two contiguous bytes, as shown in Figure 3-2. Bit 15 contains the most significant bit and Bit 0 contains the least significant bit.

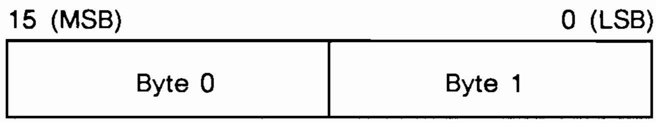

Figure 3-2. 16-Bit Integer Format

Domain FORTRAN represents a 32-bit integer (**integer*4**) as four contiguous bytes (one longword), as Figure 3-3 shows. The most significant bit in the integer is 31; the least significant bit is Bit 0.

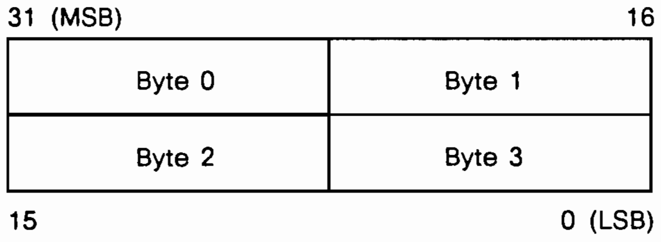

Figure 3-3. 32-Bit Integer Format

3.4 Real Numbers

This section explains how to declare variables as real numbers and how Domain FORTRAN represents **reals** internally.

3.4.1 Declaring Real Variables

Domain FORTRAN supports the **real**, real*4 (same as real), real*8, and **double precision** (same as **real*8**) data types for representing floating-point values. The **real** and **real*4** are single-precision floating-point types, and **real*8** and **double precision** are double-precision floating-point types. The following are sample declarations:

```
real                interest_rate
real*4              open, high, low, close
real*8              cpu_time
double precision    pi
```

The ranges of representable values for single-precision (**real** and **real*4**) and double-precision (**real*8** and **double precision**) floating-point values are listed in Table 3-1.

Table 3-1. Ranges for Single-Precision and Double-Precision Values

Range	Single-Precision Values	Double-Precision Values
Maximum Positive Normalized	3.403×10^{38}	1.798×10^{308}
Minimum Positive Normalized	1.175×10^{-38}	2.225×10^{-308}
Minimum Positive Denormalized	1.401×10^{-45}	7.905×10^{-323}
Zero	0	0
Precision	24 significant bits 7.2 significant figures	53 significant bits 16 significant figures

Most floating-point values are represented as normalized numbers. Values outside the normalized range are represented as denormalized numbers (if they are closer to zero) or as infinities (if they are farther from zero).

3.4.2 Internal Representation of Reals

Single-precision numbers (**real** or **real*4**) occupy four contiguous bytes, as shown in Figure 3-4.

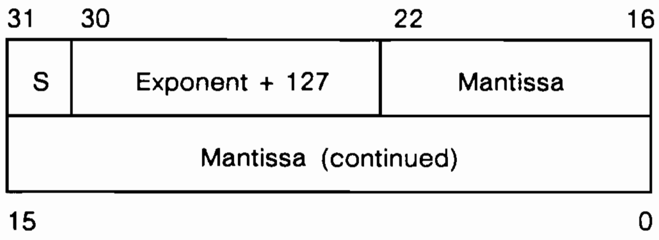

Figure 3-4. Single-Precision Floating-Point Format

Double-precision floating-point numbers (**real*8** and **double precision**) are represented in eight bytes (64 bits), as shown in Figure 3-5.

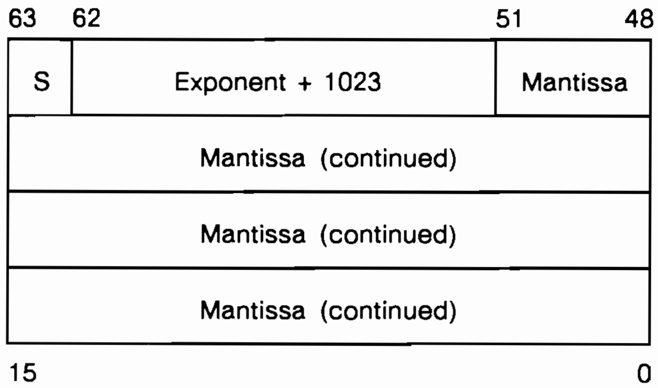

Figure 3-5. Double-Precision Floating-Point Format

For both single-precision and double-precision values, the fields within the format have the following meanings:

S — The sign bit. If *S* is 1, the represented value is negative. If *S* is zero, the value is positive.

Mantissa — The value to the right of an assumed binary point. For normalized numbers, a hidden 1 is assumed to the left of the binary point.

Exponent — The representation of the power to which the fraction is raised. For single-precision values, the actual exponent is obtained by subtracting 127 from *Exponent*; for double-precision values, the value subtracted is 1023.

The formulas in Figure 3-6 show how to obtain single-precision and double-precision values from the parts of the bit representation:

$$\textit{single-precision value} = (-1)^{\textit{sign}} \times 2^{\textit{exponent} - 127} \times 1.\textit{mantissa}_2$$

$$\textit{double-precision value} = (-1)^{\textit{sign}} \times 2^{\textit{exponent} - 1023} \times 1.\textit{mantissa}_2$$

Figure 3-6. Formulas for Single-Precision and Double-Precision Values

The following example shows how Domain/OS stores the single-precision floating-point value +100.5. The four bytes contain the bit pattern shown in Figure 3-7.

31	30							22							16
0	1	0	0	0	0	1	0	1	1	0	0	1	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15															0

Figure 3-7. Internal Representation of +100.5

The number breaks down into sign, exponent, and mantissa as follows:

sign	0	{positive}
exponent + 127	10000101	{133 in decimal}
significant part of mantissa	1001001	

Using the formula in Figure 3-6, we obtain the value 100.5, as Figure 3-8 illustrates:

$$\text{single-precision value} = (-1)^{0} \times 2^{133 - 127} \times 1.1001001_2$$

$$= 1 \times 2^{6} \times 1.1001001_2$$

$$= 1100100.1_2$$

$$= 100.5_{10}$$

Figure 3-8. Deriving a Single-Precision Value from Its Bit Representation

When you use floating-point data types, keep in mind that computer floating-point arithmetic does not obey the laws of arithmetic for real numbers. In particular, floating-point addition and multiplication are not associative, and addition and subtraction are not inverses. That is, for a given floating-point calculation, it is possible that

$$a + (b + c) \neq (a + b) + c$$

$$x + y - y \neq x$$

NOTE: In the exponential expression, x^4, where x is of data type **real*4** or **real*8** is an **integer *4**,the result is unspecified if the absolute value is greater than $2^{16} - 1$.

For complete information about floating-point computations on Domain systems, refer to the *Domain Floating-Point Guide*.

3.5 Logicals

This section explains how to declare variables as **logicals**, and how Domain FORTRAN represents **logicals** internally.

3.5.1 Declaring Logical Variables

In standard FORTRAN a **logical** is represented by four bytes. Domain FORTRAN supports the following predeclared logical data types:

logical*1	1-byte logical object
logical*2	2-byte logical object
logical*4	4-byte logical object
logical	4-byte logical object

Note that **logical** is equivalent to **logical*4**, unless you compile with the **-l*1** or **-l*2** option. Compiling with **l*1** causes **logical** to be equivalent to **logical*1**; compiling with **-l*2** causes **logical** to be equivalent to **logical*2**. Refer to Section 6.5.21 for more details about the **-l*1** and **-l*2** compiler options.

Any logical variable can have only one of two values: **.true.** and **.false.** Since the FORTRAN default naming conventions do not cover logical objects, you must always either explicitly declare them, or you must use the **implicit** statement (described in Chapter 4) to define them.

Here are two examples of explicitly defined logical variables:

```
LOGICAL winner, loser
LOGICAL*2 workday, holiday
```

You can use the smaller logical objects to make your program more efficient and provide greater compatibility with other programs. For example, **logical*1s** are particularly effective for communicating with Pascal and C programs, which use **booleans** and **chars**, respectively. The **logical*2** type can be useful for setting up equivalences with **integer*2** types.

Domain FORTRAN allows you to mix different-sized logical objects in an expression. For example, the following expression is valid:

```
LOGICAL*1   log1
LOGICAL*2   log2
LOGICAL*4   log4
   ...
IF (log1 .eq. log4) THEN
    log2 = .TRUE.
END IF
   ...
```

3.5.2 Internal Representation of Logical Variables

Domain FORTRAN represents logical objects as shown in Figure 3–9. All the bits in each byte are set to 1 to represent the value **.true.** and to 0 to represent the value **.false.**

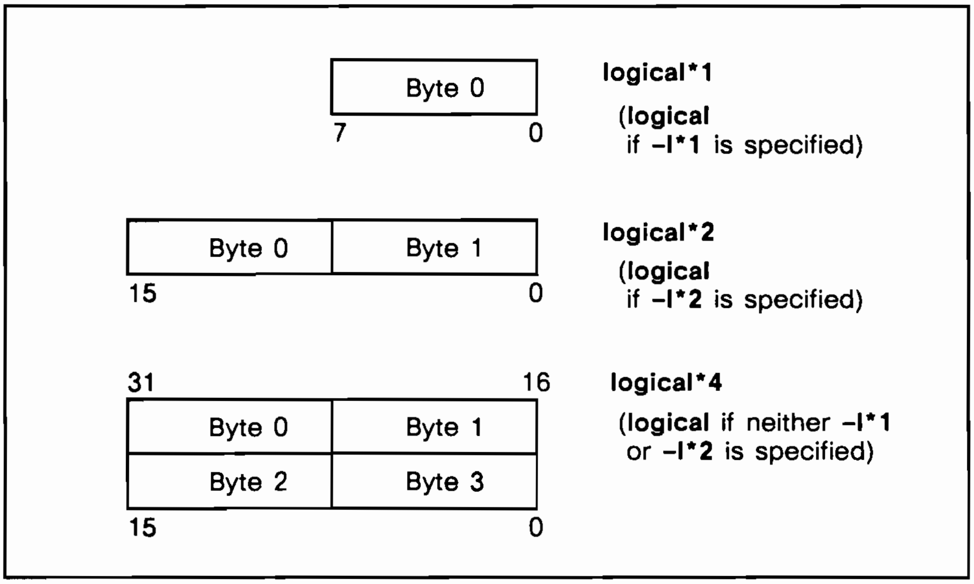

Figure 3–9. **Logical** *Format*

3.6 Complex Numbers

This section explains how to declare complex variables and how Domain FORTRAN represents complex numbers internally.

3.6.1 Declaring Complex Variables

According to standard FORTRAN, a **complex** number consists of two parts: a real part and an imaginary part. It requires eight bytes of storage. Domain FORTRAN supports the **complex** type as well as an additional double complex type, which requires sixteen bytes of storage. Another name for the complex type is complex*8. Similarly, another name for the double complex type is complex*16. Since the FORTRAN default naming conventions do not cover **complex, double complex, complex*8,** or **complex*16** numbers, you must always either explicitly declare them, or you must use the **implicit** statement (described in Chapter 4) to define them.

Here are some examples showing how to declare complex variables explicitly:

```
complex really, imagine
complex*8 truth, fantasy
double complex practical, romantic
complex*16 creative, fiction
```

3.6.2 Internal Representation of Complex Variables

Domain FORTRAN represents **complex** variables in eight contiguous bytes as shown in Figure 3-10. Each item is represented as an ordered pair of single precision floating-point numbers, where the first number in the pair is the real part and the second number is the imaginary part.

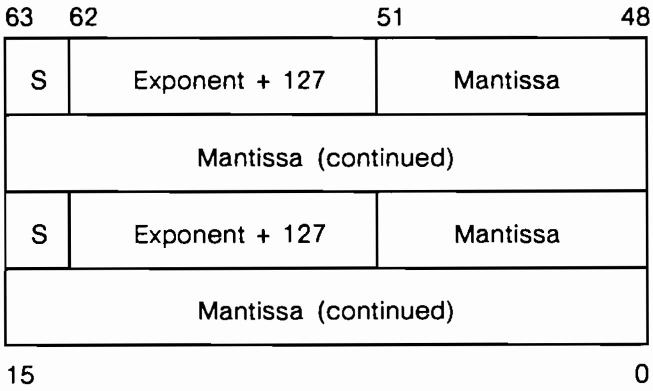

Figure 3-10. **Complex** *Data Representation*

Domain FORTRAN represents complex*16 variables in sixteen contiguous bytes as shown in Figure 3-11. Each item is represented as an ordered pair of double-precision floating-point numbers, where the first number in the pair is the real part and the second number is the imaginary part. Refer to Subsection 3.4.2 for more information about the representation of **real** numbers.

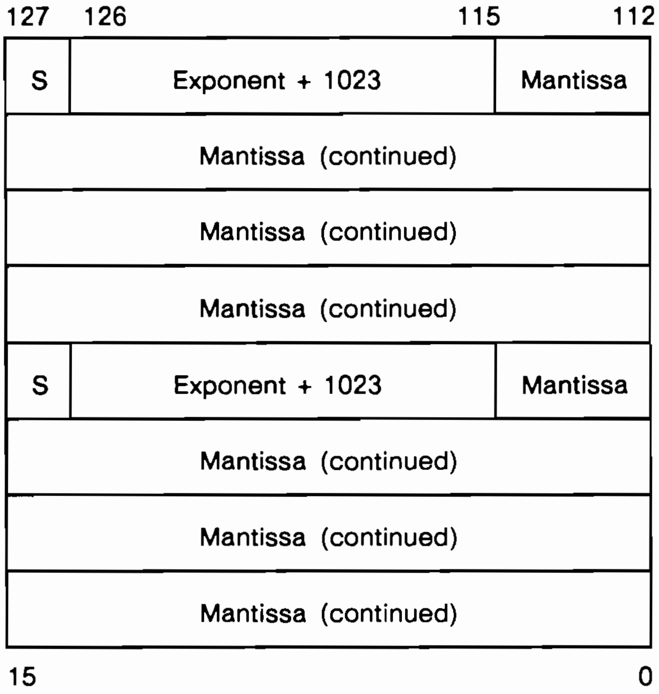

Figure 3-11. **Complex*16** *Data Representation*

3.7 Characters

This section explains how to declare variables as **characters**, and how Domain FORTRAN represents **characters** internally.

3.7.1 Declaring Character Variables

Use the **character** type to declare variables that hold character data. Each character in a Domain FORTRAN character data item occupies one byte. This is the format for declaring a **character** variable:

character [**len*] *name1* [. . . ,*nameN*]

In this format, *name1* . . . ,*nameN* are the names of the **character** variables you are declaring. The optional **len* is the length (in bytes) of the variable(s) preceded by an asterisk. If you omit *len,* the variable defaults to a 1-byte length. The value of *len* must always be a positive integer between 1 and 32768. You may enclose *len* in parentheses, but this is optional.

For example, suppose you declare the following:

```
character     one_letter
character*10  first_name
character*15  last_name
```

In this example, **one_letter** is the default 1-byte length, **first_name** is 10 bytes, and **last_name** is 15 bytes. In FORTRAN, character string lengths are determined at compile time; they do not vary.

Because *len* must be positive, a declaration like the following is incorrect:

```
character*(-10) neg_len        {Wrong!}
```

Despite the fact that string lengths are always fixed, there is one instance for which you can specify an indefinite length. If you have a dummy **character** argument in a subprogram, you can specify that its length is not determined this way:

```
character*(*) dummy_arg
```

See Chapter 5 for more information about dummy arguments, subprograms, and using indefinite lengths in arguments.

3.7.2 Internal Representation of Characters

Domain FORTRAN stores the ASCII value of each character in a **character** variable in one 8-bit byte. (The table of ISO-Latin1 characters in Appendix **B** includes ASCII values.) A **character** variable does not require a special character to denote its end.

3.8 Arrays

This section describes how to declare arrays in Domain FORTRAN. An array consists of a fixed number of elements of the same data type. This data type is called the element type. The element type can be any of the predeclared data types.

3.8.1 Specifying Array Indexes

You specify the size of the array with an index declaration. Indexes must be integers or expressions that resolve to integers by compile time.

Domain FORTRAN permits arrays of up to seven dimensions. Specify one index for each dimension. By default, index subscripts begin at 1, but you can define an array index to start at any other integer. The syntax for declaring the index only is:

([*start:*]*end*[,. . .])

where *start* is the number at which you want an individual array subscript to begin counting. If you omit this, the subscript begins at 1. The value of *end* is the highest value that the array subscript can take. If you omit *start* or you explicitly set it to 1, *end* also equals the maximum size of an individual array dimension. For example, in this declaration

```
integer nums(0:79)
```

nums is an 80-element integer array and its subscript begins at zero and ends at 79. However, in this declaration

```
real*4 table(10)
```

table's subscript by default starts at 1, and so the **end** subscript equals the maximum number of elements—10—that the array can hold.

If your array contains more than one dimension, separate the indexes with commas. For example, this declaration

```
logical truth_table(4,2)
```

defines a 4x2 array of **logicals**.

Domain FORTRAN provides two ways to define arrays. In the first method you declare the array variable and define its dimensions all in one step. In the second, the variable declaration and dimension definition can take place in separate statements. Because of FORTRAN's naming conventions, you don't always have to explicitly declare the variable before you define its dimensions. The next sections describe both methods in detail.

3.8.2 Declaring Arrays within Data Type Statements

You can use the names of data types to declare arrays. For all predeclared data types, such declarations take this form

data_type array_name([*start:*]*end*[,. . .]) . . .

where *data_type* can be **byte, integer, integer*2, integer*4, real, real*4, real*8, double precision, logical*1, logical*2, logical, complex, complex*8, double complex, complex*16,** or **character.** *array_name* is the name of the array. The previous section describes *start* and *end*. Notice that the only difference between this array declaration method and an ordinary variable declaration is that you specify the array's dimensions.

The following declares arrays that have varying data types and subscript starting points, and arrays with different numbers of dimensions:

```
integer           runs(9), hits(9), errors(9)
real*4            daily_rainfall(366)
double precision  test_data(0:4,0:1,0:2)
logical           truth_table (4,2)
```

For the **character** data type, an array declaration can include **len*, with which you specify the length of each character element in the array. A **character** array declaration takes this form:

character**len array_name*([*start1:*] *end1* [. . . [*,startN:*] *endN*]) . . .

For example, the declaration

```
character*15 last_name(10)
```

defines the 10–element array **last_name** and allocates 15 bytes for each element of the array.

3.8.3 Defining Arrays with the Dimension, Common, or Pointer Statements

FORTRAN is very lenient in its requirements for declaring the dimensions of an array. The only requirement is that you must provide the dimensions sometime before the program's first executable statement.

You can explicitly declare an array variable and then spell out the dimension information in a **dimension, pointer**, or **common** statement. Or you can use FORTRAN's naming conventions to determine the type an array variable is, and then spell out its size in a **dimension** statement. (See Section 3.9 for information on using the **pointer** statement in an array definition.)

The format for a **dimension** declaration is as follows:

dimension *array_name*([*start1:*] *end1* [. . . , [*startN:*] *endN*]) . . .

where *array_name* is the array for which the dimensions are being defined. The optional *start* parameter allows you to specify the integer at which an individual dimension's subscript begins. By default, all subscripts start at 1. *end* specifies the integer at which a dimension's subscript ends.

For example, consider the following array declarations:

```
integer   height_weight_table
dimension height_weight_table(0:9,0:9), other_data(5)
```

The first line declares that **height_weight_table** is an integer. The **dimension** statement defines that it is an array and gives its size. The **dimension** statement also defines the array **other_data** and its size. Since **other_data** begins with the letter 'o' and it has not been previously declared, FORTRAN's naming conventions dictate that **other_data** is an array of real numbers.

It is always possible to avoid using a **dimension** statement. Instead of the previous declarations, you can write:

```
integer height_weight_table(0:9,0:9)
real    other_data(5)
```

You can also dimension an array in a **common** statement. For example:

```
real batting_averages
character*15 player_name
common /baseball_team/ batting_averages(25), player_name(25)
```

Refer to Section 4.6 for information about the **common** statement.

3.8.4 Internal Representation of Arrays

In a single-dimension array, FORTRAN simply stores the elements one after another. That is, element 1 is followed by element 2, which is followed by element 3, and so on.

In a multidimensional array, the leftmost subscript varies fastest. If you define an array this way

```
real table(2,3)
```

FORTRAN stores the elements in the following order:

```
1,1
2,1
1,2
2,2
1,3
2,3
```

3.9 Pointers—Extension

In order to make it possible to use the pointers returned by programs written in other languages, Domain FORTRAN has a **pointer** statement. This statement does not declare an actual **pointer** variable; it simply gives FORTRAN programs access to the pointers returned by programs written in other languages.

There are two steps to declaring a **pointer** variable. They are:

1. Explicitly declare an **integer*4** variable that will hold the pointer value.

2. Use the **pointer** statement to associate that **integer*4** variable with a pointer.

The syntax for the **pointer** statement can take either of two different formats. The first is

pointer */int4_var/based_var_list*

where *int4_var* is the name of the variable that will hold the pointer. You must explicitly declare *int4_var* before using it in the **pointer** statement, and you must enclose it in a pair of slashes (/.../). It cannot be an array of **integer*4** variables.

based_var_list can contain one or more variables or arrays of any data type. Domain FORTRAN does not automatically allocate storage for these based variables. However, you can dimension an array that's part of the *based_var_list* within a **pointer** statement.

Here is a sample **pointer** declaration using this syntax format:

```
character*128 arg_text
integer*4     arg_ptr, nums
pointer       /arg_ptr/ nums(10,10), arg_text
```

Notice that the **integer*4** variable **nums** is defined as a 10x10 array within the **pointer** statement. When storage is allocated for **nums** and **arg_text, arg_ptr** will point to the address of a 528-byte area of memory—400 bytes for **nums** and 128 bytes for **arg_text**.

The alternate syntax for the **pointer** statement is

pointer *(int4_var, single_var)*

where *int4_var* is the name of the variable that will hold the pointer. You must explicitly declare *int4_var* before using it in the **pointer** statement. It cannot be an array of **integer*4** variables.

single_var is the name of a variable of any type.

Here is a sample **pointer** declaration using the alternate syntax format:

```
integer*4     ptr
character*10  line
pointer       (ptr, line)
```

See the listing for **pointer** in Chapter 4 for information about using **pointers**.

Chapter 4

Code

This chapter describes all the statements that make up the action part of a Domain FORTRAN program or subprogram. The first part of the chapter gives an overview of what is available. The remainder is a Domain FORTRAN encyclopedia, complete with many examples.

The overview is divided into the following categories:

- Branching and looping
- Mathematical operators
- Specification structures
- Input and output
- Miscellaneous statements

4.1 Branching and Looping

Domain FORTRAN supports the three standard FORTRAN forms of the **if** statement for conditional branching: logical **if**, arithmetic **if**, and block **if**. Domain FORTRAN also supports the **end if** statement, which delimits the end of a block **if** statement.

Domain FORTRAN supports **do**—the looping statement of standard FORTRAN. In addition, Domain FORTRAN supports the **do while** looping statement and the **end do** statement, which delimits the range of a **do** or **do while** loop.

Domain FORTRAN supports three forms of **go to**, which transfers control from the current statement.

Table 4-1 lists the branching and looping statements supported by Domain FORTRAN.

Table 4-1. Domain FORTRAN Branching and Looping Statements

Statement	Action
do	Executes a block of statements zero or more times.
do while	Executes a block of statements as long as the specified logical expression is true.
end do	Terminates the range of a **do** or **do while** statement.
arithmetic **if**	Branches to one of three paths, depending on the value of an arithmetic expression.
block **if**	Branches to a statement or block of statements if a specified expression is true.
end if	Marks the end of a block **if** statement.
logical **if**	Branches to a specified statement if a specified logical expression is true.
unconditional **go to**	Transfers control to a statement specified by a label.
assigned **go to**	Transfers control to a statement specified by an **integer*4**.
computed **go to**	Transfers control to one of several paths, depending on the value of an integer expression.

4.2 Mathematical Operators

Domain FORTRAN supports all the standard arithmetic, logical, and relational operators. Table 4–2 contains these operators.

Table 4–2. Domain FORTRAN Operators

Operator	Meaning	Precedence
**	Exponentiation	highest precedence
* /	Multiplication Division	↑
+ -	Addition Subtraction or negation	
//	Concatenation	
.eq. .ne. .lt. .le. .gt. .ge.	Equal to Not equal to Less than Less than or equal to Greater than Greater than or equal to	
.not.	Logical NOT for **logical** operands Bitwise NOT for **byte**, **integer*2**, and **integer*4** operands	
.and.	Logical AND for **logical** operands Bitwise AND for **byte**, **integer*2**, and **integer*4** operands	
.or.	Logical AND for **logical** operands Bitwise AND for **byte**, **integer*2**, and **integer*4** operands	↓
.eqv. .neqv.	Logical equivalence Logical nonequivalence	lowest precedence

Operators that are grouped together (for example, multiplication and division) have the same precedence. See "Assignment Statements" in Section 4.6 for information on using these operators.

4.3 Specification Structures

Domain FORTRAN supports the statements listed in Table 4-3 for specifying subprogram or variable types or for declaring how certain values are to be stored in memory. For details on all of these statements except **dimension**, refer to Section 4.6. Section 3.8.3 includes discussions of **dimension**.

Table 4-3. Domain FORTRAN Specification Statements

Statement	Action
[illegible]	[illegible]
common	Defines common storage areas and lists the variables and arrays to be stored in those areas.
dimension	Defines the size (dimension) of an array.
equivalence	Associates two or more data entities with the same storage area.
external	Allows an external function or subroutine to be used as an actual argument.
implicit	Associates user-defined initial letters with specific data types.
[illegible]	[illegible]
intrinsic	Identifies a specific or generic intrinsic function.
[illegible]	[illegible]
parameter	Assigns a symbolic name to a constant.
save	Saves the values of a variables in static storage.

4.4 Input and Output

Domain FORTRAN supports the I/O statements shown in Table 4–4. For details about the statements, refer to Section 4.6 and Chapter 8.

Table 4–4. Domain FORTRAN I/O Statements

Statement	Action
backspace	Explicitly positions a file before the record most recently read or written.
close	Closes a file.
decode	Transfers data from memory to listed I/O items.
encode	Formats data and transfers it to memory.
endfile	Places an end-of-file marker in a file opened for sequential access.
include	Inserts a file into the source file.
inquire	Reports the existence and/or attributes of a unit or file.
open	Opens a file.
print	Transfers data to standard output.
read	Transfers data from a file to internal storage.
rewind	Positions a sequential file before its first record.
write	Transfers data from internal storage to an output file.

4.5 Miscellaneous Statements

Several Domain FORTRAN elements do not fit neatly into categories. Table 4–5 lists these elements.

Table 4–5. Miscellaneous Statements

Statement	Action
assign	Assigns a statement label to an **integer*4** variable.
call	Calls and passes control to a subroutine.
continue	Marks a place in a program unit for a statement label.
data	Sets initial values of variables, array elements, and substrings.
end	Marks the end of a program unit.
entry	Defines a secondary entry point in a subprogram.
format	Specifies the format of data to be read, written, or printed.
pause	Temporarily stops a program until a user intervenes.
pointer	Gives FORTRAN programs access to pointers returned by programs in other languages.
program	Names the main program unit.
return	Returns control to the calling program unit from a subprogram.
stop	Terminates a program's execution.

4.6 Encyclopedia of Domain FORTRAN Statements

The remainder of this chapter contains an alphabetical listing of all the keywords that you can use in the action part of a Domain FORTRAN program or subprogram. It also contains listings for several FORTRAN concepts. Figure 4-1 contains the keyword listings, and Figure 4-2 contains the conceptual listings.

assign	**end**	namelist
atomic	end do	**open**
backspace	**endfile**	options
call	**end if**	**parameter**
close	**entry**	**pause**
common	**equivalence**	pointer
continue	**external**	**print**
data	**format**	**program**
decode	**go to**	**read**
dimension☆	**if**	**return**
discard	**implicit**	**rewind**
do	implicit none	**save**
do while	include	**stop**
else if ... then	**inquire**	**write**
encode	**intrinsic**	

☆**See Section 3.8.3 for details about dimension.**

Figure 4-1. Keyword Listings in Encyclopedia

array operations	compiler directives
assignment statements	I/O attributes

Figure 4-2. Conceptual Listings in Encyclopedia

array operations

Subsection 3.8.3 describes how to declare and dimension an array. This listing explains how to use arrays in your programs.

ASSIGNING VALUES TO ARRAYS

To assign a value to an array variable you must supply the following information:

- The name of the array variable.
- An index expression enclosed in parentheses.
- A value of the component type. That is, if the array is of type logical, you must provide a logical value for the array element.

Assigning Values to Single Array Elements or Substrings

The following program fragment assigns values to single elements of arrays:

```
integer      num, levels(10)
real         wage_scale
dimension    wage_scale(12,10)
logical      x(10)
character*1  letters(26)
    .
    .
    .
levels(4) = 100
wage_scale(1,5) = 7.25
x(1) = .true.
num = 2
letters(num+3) = 'e'  {Notice arithmetic expression for index,}
                      {rather than simple constant.           }
```

In the preceding example, the **character** array has elements that are only one byte each. However, **character** arrays often have multibyte elements. If you want to assign a value to a substring of one of those multibyte elements, you must supply both the element number and the substring range.

For example:

```
character*15 city_name(10)
     .
     .
     .
city_name(3) = 'CopXXXagen'
city_name(3) (4:6) = 'enh'
print *, 'The city''s name is: ', city_name(3)
```

This fragment declares the 10-element array **city_name** and specifies that each element is a 15-byte **character** variable. The line

```
city_name(3) (4:6) = 'enh'
```

tells FORTRAN to look at the third element of array **city_name**, and assign **'enh'** to the fourth, fifth, and sixth bytes of that element. The output looks like this:

```
The city's name is: Copenhagen
```

Assigning Values to Multiple Elements

In addition to assigning values to single array elements or to substrings, you probably will want to assign values to many or all of the elements in an array. FORTRAN provides a variety of looping structures to do so. Both of the following examples initialize all array elements to zero:

```
integer i, my_array(10)
do i = 1,10           {First method; simple do loop.}
    my_array(i) = 0
enddo

integer i, nums_array(10)
data nums_array/10*0/  {Second method; data statement}
                       {and repeat count.           }
```

See the listings for **data** and **do** later in this encyclopedia for more information.

ARRAY STORAGE

In a single-dimension array, FORTRAN simply stores the elements one after another. That is, element 1 is followed by element 2, which is followed by element 3, and so on.

In a multidimension array, the leftmost subscript varies fastest. So if you define an array this way

```
real table(2,3)
```

FORTRAN stores the elements in the following order:

```
1,1
2,1
1,2
2,2
1,3
2,3
```

To access such an array in the same order as FORTRAN, your loops should look like this:

```
do j = 1,3
  do i = 1,2
    print *, table(i,j)
  enddo
enddo
```

EXAMPLE

```
*   This program accepts five user-entered numbers, loads them into
*   an array, then reads back through the completed array to find the
*   largest value entered.

      program array_example
      integer*2 i
      real*4 input(5), big_num
      do i = 1,5
          print 10
10        format ('Enter a number: ', $)
          read *, input(i)
      enddo
      big_num = input(1)
      do i = 2,5
          if (input(i) .gt. big_num) big_num = input(i)
      enddo

      print 20, big_num
20    format ('The largest number you entered was: ', F10.3)
      end
```

USING THIS EXAMPLE

This program is available online and is named **array_example**. Following is a sample execution of the program:

```
Enter a number: 555.7
Enter a number: 9999.32
Enter a number: 9999.33
Enter a number: 12
Enter a number: 63.8
The largest number you entered was:   9999.330
```

assign Assigns a statement label to an **integer*4** variable.

FORMAT

assign *label* **to** *intvar*

ARGUMENTS

label The label of an executable statement or a **format** statement in the current program unit.

intvar The name of an **integer*4** variable.

DESCRIPTION

Assign associates the **integer*4** variable *intvar* with *label*, enabling the program to refer to the variable name in subsequent assigned **go to** statements, or to use the variable name as a format identifier in an input or output statement.

The **assign** statement can help make code more readable. For instance, suppose your program contains the following:

```
      integer*4 i, error
      assign 100 to error
          . . .
      if (i .lt. 0) goto error
          . . .
100   {Error handling routine}
```

Because label 100 has a meaningful name (**error**) assigned to it, the **goto**'s purpose is obvious: it is transferring control to an area of the program that will handle an error.

Note, however, that using assigned **goto**'s can cause your program to run significantly more slowly. (Refer to the listing for **go to** in this encyclopedia for more details about using **go to** statements.)

EXAMPLE

```
*  This program uses the assign statement to associate names with
*  both a format statement and two areas of the program  - the
*   error-handling section and the end.

      program assign_example

      integer*4 error, line, num, finish        {Declare variables}
      assign 30 to error                        {and assign label }
      assign 20 to line                         {numbers to them. }
      assign 999 to finish

      print *, 'Can you guess what number I''m thinking of?'
      print *, 'Enter an integer between 1 and 10'
10    read *, num
      if ((num .lt. 1) .or. (num .gt. 10)) goto error
      print *, 'That wasn''t the number. Too bad.'
      goto finish

20    format (I4, ' is not between 1 and 10. Try again.')
30    print line, num
      goto 10

999   end
```

USING THIS EXAMPLE

This program is available online and is named **assign_example**. Following is a sample run of the program:

```
Can you guess what number I'm thinking of?
Enter an integer between 1 and 10
100
100 is not between 1 and 10. Try again.
-3
-3 is not between 1 and 10. Try again.
6
That wasn't the number. Too bad.
```

assignment statements

FORMAT

var = *exp*

ARGUMENTS

var The variable, array element, or character substring to be assigned a value.

exp An arithmetic variable, constant, or expression, or a variable, constant, or expression of type **logical** or **character**.

DESCRIPTION

An assignment statement assigns the value of an expression to a variable, array element, or character substring. The expression (*exp*) appears to the right of the equal sign; the variable (*var*) acquiring the value appears to the left.

There are three kinds of assignment statements: arithmetic, logical, and character. The following subsections describe each type of assignment statement.

Arithmetic Assignment Statements

Table 4-6 lists the standard operators used by arithmetic assignment statements.

Table 4-6. Arithmetic Assignment Operators

Arithmetic Operator	Meaning
+	Addition
–	Subtraction or negation
*	Multiplication
/	Division
**	Exponentiation

Arithmetic assignment statements work as you would expect. Consider these examples:

```
* Add b and c and assign the result to y.

      y = b + c

* Square both A and B, multiply the squares together, and assign the
* result to C.

      C = (A**2) * (B**2)

* Multiply a by b, divide the product by 2.0, and assign the
* result to the statement function 'area'. (See the description of
* statement functions in Chapter 5.)

      area(a,b) = a*b/2.0
```

No two operators can be side by side in an expression. For example, this is illegal:

```
      c = a*-b        {Illegal!}
```

The correct way to write such an assignment is to use parentheses to organize the expression into subexpressions; that is,

```
      c = a * (-b)
```

You can also use parentheses to specify the order in which you want FORTRAN to evaluate your expression. FORTRAN normally evaluates expressions by performing the operations in order of precedence. (Table 4–2 lists all operators and their precedence.) If two operators have the same precedence, FORTRAN evaluates them from left to right—except in the case of exponentiation (see below). However, if you use parentheses to make subexpressions, FORTRAN evaluates quantities in parentheses first.

For example, the statement

```
      y = (b + c) * (d + e)
```

tells FORTRAN to evaluate **b+c** first, then to evaluate **d+e**, and finally, to multiply the value of those two subexpressions and assign the result to y. Since multiplication has a higher precedence than addition, if you had written the statement without parentheses, FORTRAN would have multiplied **c** and **d**, added **b** and then **e** to the result, and assigned the answer to y.

If an expression contains nested parentheses, FORTRAN first evaluates the quantities in the innermost pair, then the quantities in the next–innermost pair, and so on.

Exponentiation Evaluation

Although FORTRAN performs most operations in a left-to-right order, if you have two or more exponentiation operations in an expression, FORTRAN evaluates them in right-to-left order. This is in keeping with the rules of mathematical notation. Thus, the expression

```
a**b**c
```

is actually

```
a**(b**c)
```

or "**a** raised to the power of **b** raised to the power of **c**."

Assigning Arithmetic Values to Complex Variables

If you want to assign a value to a **complex** variable, you must specify both its real and imaginary parts, separated by a comma and enclosed in parentheses. For instance:

```
complex complicated
complex*16 tricky
    .
    .
    .
complicated = (6.E2,5.2993)
tricky = (77.333,4.E3)
```

Mixed Data Types in Arithmetic Assignment Statements

Usually, the variable and the value of the expression in an arithmetic assignment statement have the same data type. However, you can mix data types in such assignment statements. In translating these statements, the compiler first evaluates *exp*, and then converts it to the data type of *var*. If the expression is **real** and the variable is **integer**, this requires truncation: deleting the decimal point and all digits to the right of the decimal point. [illegible] FORTRAN does a sign ex- [illegible]

There are several intrinsic functions for explicit type conversion, including: **real**, which converts an integer expression to the real data type; **int**, which converts a real expression to integer; and **nint**, which rounds the value of a real expression before converting it to an integer. Except for **nint**, the type conversion functions also perform truncation. See Appendix C for more information on intrinsic functions.

You can also mix operands of different types within a single expression. For example, the following is legal:

```
integer score
real*4  degree_of_difficulty, result
    .
    .
    .
result = ((score * degree_of_difficulty) + (score * 2.0))
```

Operations involving both **real** and **integer** operands produce **real** results. And, except for exponentiation to an integer power (for example, 6.2**2), FORTRAN converts the **integer** to **real** before evaluating the expression.

Logical Assignment Statements

A logical assignment statement assigns the value **.true.** or **.false.** to a variable or array element, depending on whether the expression to the right of the equal sign is true or false. If the expression, *exp*, evaluates to "true," the variable *var* is true; if *exp* evaluates to "false," *var* is false. In order to use a logical assignment statement, *var* must be of type **logical**, and *exp* must evaluate to a **logical** result.

A logical expression can consist simply of a **logical** variable or constant. For instance:

```
logical the_truth
    .
    .
    .
the_truth = .true.
```

Relational Operators in Logical Assignment Statements

A logical expression can consist of more than just a **logical** variable or constant. It can also consist of relational expressions—statements that describe the numerical relationship between two values or variables. Table 4–7 lists the relational operators.

Table 4–7. Domain FORTRAN Relational Operators

Relational Operator	Meaning
.eq.	Equal to
.ne.	Not equal to
.lt.	Less than
.le.	Less than or equal to
.gt.	Greater than
.ge.	Greater than or equal to

Suppose your program contains the following fragment:

```
logical   done
integer*4 amt_on_hand, enough
    .
    .
    .
done = amt_on_hand .ge. enough
```

In this example, FORTRAN compares the value of **amt_on_hand** with that of **enough**. If **amt_on_hand** is greater than or equal to **enough**, the expression is true, and FORTRAN assigns the value **.true.** to variable **done**. If the expression is false (**amt_on_hand** is less than **enough**), FORTRAN assigns the value **.false.** to **done**.

The operands of a relational operator don't have to be simple variable names. They can be arithmetic expressions. For example, this is legal:

```
done = ((b+c) .le. (a*e))
```

You can use all the relational operators to express a relationship between any of FORTRAN's numerical data types *except* **complex**. The only valid relational operators for **complex** variables are **.eq.** and **.ne.** If you try to use any of the others, you get a compile-time error. This code fragment triggers such an error:

```
complex deep, deeper
    .
    .
    .
if (deep .lt. deeper) then      {Illegal!}
     {statement}
```

You cannot use any of the relational operators to express a relationship between **logical** variables. Instead, use the logical operators. The next subsection describes these operators.

Logical Operators in Logical Assignment Statements

In addition to using relational operators to make a logical assignment, you can also use the logical operators. They are: **.and.**, **.or.**, **.not.**, **.eqv.**, and **.neqv.**

.and. and **.or.** combine variables, constants, and relational subexpressions in an expression. **.not.** is a unary operator. The following examples show how these operators are used.

```
      logical   done, part_done
      integer*4 p, q, r, s
          .
          .
          .
      done = ((p .lt. q) .and. (r .gt. s))
{True if both subexpressions are true. }

      part_done=((p .lt. q) .or. (r .gt. s))
{True if one or both subexpressions are true.}

      done = .not. (p .lt. q)
{True if subexpression is false.}
```

.Eqv. and **.neqv.** determine whether two logical expressions have the same truth value. For example, if your program contains the following

```
      logical done1, done2, result
        .  .  .
      result = done1 .eqv. done2
```

result is assigned the value **.true.** if **done1** and **done2** have the same value (both true or both false), and is assigned **.false.** if **done1** and **done2** have different values.

.Neqv. is the complement of **.eqv.**

For information on the precedence of logical operators, see Table 4–2.

Character Assignment Statements

Although character assignment statements are much like the arithmetic or logical assignments already described in this section, they do have a few extra characteristics. You can make a character assignment to a variable or substring. The statement takes this form:

var(begin_num:end_num) = 'string' | exp

Var is the name of the variable to which you are making the assignment, and *string* is the string of characters, enclosed in single quotes, that you are assigning to *var*. If you don't specify a literal string, you must provide an *exp* that resolves to a string.

If you are assigning a value to only some of the elements of a character string, you must specify a *begin_num* and an *end_num* to represent the first and last elements to be assigned in the string.

Here are some examples of character assignment statements:

```
character*24 scale              {First example. }
scale = 'do re mi fa sol la ti do'

character*15 giant              {Second example.}
giant = 'fee aaa foe fum'
giant(5:7) = 'fie'
print *, 'Giant string equals: ', giant
end
```

In the second example, the string 'aaa' in elements 5, 6, and 7 is replaced with the string 'fie'. So if you execute the example, this is the result:

```
Giant string equals: fee fie foe fum
```

You can use an assignment statement to equate elements in a character string. For example:

```
character*15 giant
giant = 'aaa fie foe fum'
giant(1:3) = giant(13:15)
print *, 'Giant string equals: ', giant
```

In this case, the result is:

```
Giant string equals: fum fie foe fum
```

However, you should be careful of overlapping assignments. There's no problem if you assign elements later in a string to those earlier in it. For example, suppose the string **giant** contains the following

```
         1         2
12345678901234567890{Ruler to help you see individual bytes.}
fee fie foe fum     {Contents of giant.}
```

and you make this assignment:

```
giant(5:7) = giant(6:8)
```

The assignment works and the result is

```
Giant string equals: fee ie  foe fum
```

The potential problem comes if you make an overlapping assignment of elements early in a string to those later in the string.

For example, if you make this assignment

```
giant(6:8) = giant(5:7)
```

your results will vary depending on the machine on which you are running the program.

In addition to making assignments to a given substring, you can reference a substring in an array of character strings as follows:

```
character*12 city_name(10)
city_name(7) = 'VancXXXXr   '
city_name(7) (5:8) = 'ouve'
print *, 'The city''s name is: ', city_name(7)
```

This example declares the 10-element array **city_name** and specifies that each element is a 12-byte character variable. The line

```
city_name(7) (5:8) = 'ouve'
```

tells FORTRAN to look at the seventh element of array **city_name**, and assign 'ouve' to the fifth, sixth, seventh, and eighth bytes of that element. The output looks like this:

```
The city's name is: Vancouver
```

Mixed Length Assignments in Character Assignment Statements

The string or substring you assign to a character variable need not be the same length as the variable; FORTRAN can compensate for the difference. If *exp* is longer than the declared length of *var*, FORTRAN truncates the expression from the right before assigning it to the variable.

For example, given

```
character*10 scale
    . . .
scale = 'do re mi fa sol la ti do'
print *, scale
```

FORTRAN truncates the string to 10 characters, and prints **scale** as **do re me f**.

If the expression is shorter than the declared length of *var*, FORTRAN pads the end of the expression with blanks before assigning it to the variable. This means that the line

```
city_name(7) = 'VancXXXXr
```

in the previous section could have been written this way (without the explicit blanks):

```
city_name(7) = 'VancXXXXr'
```

Concatenation

The concatenation operator, //, joins strings or substrings in a character expression, as the following sample program illustrates:

```
*  Concatenation example
      character*6  re
      character*14 dundant
      character*26 result

*  Character assignment statements

      re = ' Again'
      dundant = ' and again and'
      result = re//dundant//re

*  Print the concatenated string

      print *, result
      end
```

This program is available online and is named **concat_example**. If you execute this program, you get this result:

```
Again and again and Again
```

Comparing Character Expressions

You can use the relational operators **.eq.**, **.ne.**, **.lt.**, **.le.**, **.gt.**, and **.ge.** to compare the values of two character expressions. "Value," in this case, means each character's collating value. Domain FORTRAN uses the standard ASCII collating sequence; see the ISO Latin–1 table in Appendix B for a listing of ASCII values.

When comparing strings, FORTRAN takes the initial letter in the first string, compares it with its counterpart in the second string, and so on, until it reaches the first character position at which the two strings differ. At that point, FORTRAN determines which of the two characters is "larger," based on the collating sequence, and ranks the strings accordingly. If one of the strings is shorter than the other, FORTRAN treats the shorter string as if it were padded with blanks to equal the length of the longer string.

In the following fragment, **answer** gets the value **.true.** in each case:

```
logical answer

answer = 'DEVOTE' .lt. 'DEVOTEE'
answer = 'yes' .gt. 'y'
answer = 'kode' .gt. 'code'
```

Mixed Data Types in Character Assignment Statements

Domain FORTRAN allows you to mix integer variables and array elements with character and Hollerith constants in assignment statements and comparisons. You can also initialize variables, arrays, or array elements of any type to character or Hollerith constants with a **data** statement. FORTRAN places the first character of the constant in the leftmost byte of the variable, and the subsequent characters in subsequent bytes. Any leftover bytes are filled with blanks. However, if the character constant is longer than the variable, FORTRAN returns an error.

Comparisons are performed in a similar manner: left to right, one byte at a time.

atomic   Protects a shared variable in a multiprocessor environment. (Extension)

FORMAT

atomic *varlist*

ARGUMENT

varlist   A list of shared variables that are to be protected.

DESCRIPTION

The **atomic** specification statement is for use in programs designed to execute in the multiprocessor environment of the Series 10000 workstation. You use this statement to declare a shared variable as atomic, thus preventing multiple processors from attempting to update the variable simultaneously.

To execute the assignment statement

```
x = x + 1
```

the processor loads **x** into a register, evaluates the expression, and then stores the result in the variable. On a multiprocessor Series 10000 workstation, if **x** is in shared memory, another processor may change the variable's value between the load and the store operations. By declaring the variable as atomic, however, you prevent other processors from accessing the variable between load and store operations.

The protection provided by the **atomic** statement applies only during simple updates. It does not apply to an atomic variable that is passed to a function whose return value is being assigned to the same variable. In the following examples, assume that the variable **x** has been declared as atomic.

```
* Protection applies when processor loads x, adds 1, and assigns the new
* value to x.
        x = x + 1

* Protection does not apply.  Other processors may access x during the
* execution of this statement.
        x = func(x)

* Protection applies only when processor loads x, adds the value
* returned by func(x), and assigns the new value to x.  Protection does
* not apply while a processor computes the return value of func(x).
        x = x + func(x)

* The previous statement executes in this order:
        z = func(x)
        x = x + z
```

EXAMPLE

The following fragment uses the **atomic** statement to declare variables x and y as atomic.

```
real*4 x
integer*4 y
atomic x, y
```

backspace Explicitly positions a file before the preceding record.

FORMATS

backspace *unitid* {short form}

backspace ([**unit** =] *unitid* [**,iostat** = *sfield*] [**, err** = *label*]) {long form}

ARGUMENTS

unitid The integer assigned to the unit you wish to **backspace**. The phrase **unit=** is optional if *unitid* is the first argument.

sfield I/O status specifier: *Sfield* must be an **integer*4** variable. FORTRAN returns 0 to *sfield* if the operation completes without error. If the operation results in an error, FORTRAN returns a 32–bit system status code.

label Error statement specifier: *label* designates a statement to which control goes in case of an error. If you omit this phrase and an error occurs, the program terminates.

See the listing for "I/O attributes" later in this encyclopedia for more information on these arguments.

DESCRIPTION

Backspace allows you to reread or rewrite a record. To do so, it repositions the file connected to the specified unit before the record that was just read or written. You can also use **backspace** to position the file before its end–of–file mark, in order to write more data.

The **backspace** statement can take one of two forms: a short form or a long form. With the short form you simply identify the unit on which you want the **backspace** to work. The long form allows you to handle error conditions.

For example:

```
*   Short form backspace.
*   Perform backspace on the file connected to unit 4.
       backspace 4

*   Long form backspace.
*   Perform backspace on the file connected to unit 3.
*   Return the I/O status into variable 'back_status.'
*   If there is an error, go to the statement labeled 999.
```

```
integer*4 back_status
         . . .
      backspace (3, iostat=back_status, err=999)
```

If the file is at its starting point, **backspace** has no effect.

Since **backspace** works on a file, you must use the **open** statement to open the file before you **backspace** through it. See the listing for **open** later in this encyclopedia for information on opening a file and for an explanation of the *unitid* numbers that are preassigned. Note that you must explicitly specify the *unitid*. You cannot use an asterisk (*) as a unit identifier with **backspace**.

Backspace works on UASC files, fixed-length record files, and variable-length record files. See *Programming with Domain/OS Calls* for details about these three file structures.

EXAMPLE

See the example in the listing for **endfile** later in this encyclopedia.

call   Calls and passes control to a subroutine subprogram.

FORMAT

call *sub_name* [*(arg1 . . . ,argN)*]

ARGUMENTS

sub_name	The name of the subroutine being called.
arg	One or more actual arguments to be passed to the subroutine. If you use multiple arguments, separate them with commas, and enclose the entire argument list in parentheses.

DESCRIPTION

Call suspends execution of the calling program unit and passes control to the first executable statement in the target subroutine.

When calling a subroutine, you must supply an actual argument (*arg*) for each dummy argument in the subroutine.

Actual arguments may be constants, variable names, array names, array element names, expressions, function names, or subroutine names. Actual arguments must agree with their corresponding dummy arguments in type, number, and order. Array arguments should agree in dimension.

If a dummy argument is to get a new value during the subroutine's execution, do not use a constant or an expression that requires evaluation as its corresponding actual argument, as this can lead to unpredictable results.

For example, the following incorrect example tries to alter the value of **num**, which has been assigned a constant value of 5 by the **parameter** statement.

```
program badcall        {This program is Wrong!}
integer*4 num
parameter (num=5)
call oops (num)
print *, num
end

subroutine oops (new_num)
integer*4 new_num
new_num = 4
end
```

If you try to run a program like this, it terminates with an access violation because you are attempting to write to a read-only variable.

Unless you specify an alternate return, the subroutine returns control to the next statement in the calling program unit (the statement after the **call**).

Domain FORTRAN permits recursive subroutines and functions; that is, a subroutine or function can call itself.

EXAMPLE

```
      program call_example

      integer*2  i
      real*4     hours_worked, hourly_wage, salary
      logical    again
      again = .true.

*  Loop to compute employee salaries.  After the user supplies the
*  hours worked and the hourly wage, the subroutine compute_wages is
*  called.  In the call statement's argument list, the first two are
*  input and the last is an output argument.  The second subroutine
*  call lets a user decide whether to continue computing salaries.

      do while (again)
          print 20
20        format ('Enter the hours worked: ', $)
          read *, hours_worked
          print 25
25        format ('Enter the employee''s hourly wage: ', $)
          read *, hourly_wage
          call compute_wages (hours_worked, hourly_wage, salary)
          print 30, salary
30        format ('The salary is ', F7.2)
          call find_answer(again)
      end do                           {close do/while statement }
      end
*********************************************************************
*  This subroutine computes an employee's wages based on the hours
*  worked and his/her hourly rate.

      subroutine compute_wages (hours_worked, hourly_wage, salary)
      real*4 hours_worked, hourly_wage, salary, reg_pay, overtime

      salary = 0.0
      if (hours_worked .le. 40.0) then
          salary = hourly_wage * hours_worked
      else                         {the employee worked overtime }
          reg_pay = hourly_wage * 40.0
```

```
        overtime = 1.5 * (hourly_wage * (hours_worked - 40.0))
        salary = reg_pay + overtime
      endif                         {close if statement          }

      end                           {return to print statement   }
                                    {just after subroutine call  }

*****************************************************************************
*  This subroutine lets a user decide whether to continue computing
*  employee salaries.

      subroutine find_answer(repeat)
      logical   repeat
      character answer

      print 100
100   format (/, 'Again? (Y or N) ', $)
      read (*, '(A1)') answer
      if ((answer .eq. 'n') .or. (answer .eq. 'N')) repeat = .false.

      end                           {return to end do statement   }
                                    {just after subroutine call   }
```

USING THIS EXAMPLE

This program is available online and is named **call_example**. Following is a sample execution of the program:

```
Enter the hours worked:
Enter the employee's hourly wage:
The salary is  804.44

Again? (Y or N)
Enter the hours worked:
Enter the employee's hourly wage:
The salary is  183.00

Again? (Y or N)
```

close Terminates the connection between the specified unit and the file.

FORMAT

close ([**unit** =] *unitid* [**,iostat** = *sfield*] [, **err** = *label*] [, **status** = *status*])

ARGUMENTS

unitid The integer assigned to this unit by a previous **open**; must be an **integer** constant, variable, or expression. The phrase **unit** = is optional if *unitid* is the first argument.

sfield I/O status specifier: *sfield* must be an **integer*4** variable. FORTRAN returns 0 to *sfield* if the operation completes without error, or a 32-bit system status code if the operation results in an error.

label Error statement specifier: *label* designates a statement to which control will go in case of an error. If you omit this and there is an error, the program terminates.

status One of the *status* specifiers: 'keep', which tells the system to retain the file after the **close** statement, or 'delete', which tells the system to delete the file after closing it. 'Keep' is the default for all files except those given 'scratch' status at **open**.

See the listing for "I/O attributes" later in this section for more information on these arguments.

DESCRIPTION

Close breaks the connection, previously established by **open**, between the unit that *unitid* specifies and the associated file. The connection is broken for all program units, not just the program unit in which the **close** appears. Note that you cannot **close** standard input and standard output files.

A **close** statement can look as simple as this:

```
close(4)       {Close the file that is connected to unit 4.}
```

After issuing **close**, you may use another **open** statement to reconnect the unit to the same file (if this file still exists) or to a different file. Similarly, you may reconnect the associated file, if it is named, to the same or a different unit.

By default, the file continues to exist after a **close** unless it is opened with **status**='scratch', or you specify **status**='delete' in the **close** statement.

EXAMPLE

```
*  This program demonstrates the use of the close statement.  It
*  prints names from the file 'names_data' and then closes the file.
*
*  NOTE: You must obtain file "names_data" before running this
*  program, and you must store names_data in the same directory as
*  close_example.bin.

      program close_example

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/fio.ins.ftn'
%include '/sys/ins/error.ins.ftn'

      integer*4    closestat
      character*26 line
      character*10 first_name
      character*1  middle_initial
      character*15 last_name, word
      open(10, file='names_data', recl=26, status='readonly')

*  This section reads names from a file and formats them for output.
*  The format statements takes the lengths of the character fields
*  from the variable declarations.

      print *, 'Here are the names:'
5     read(10, '(A26)', end=100) line
      first_name(1:10) = line(1:10)
      middle_initial = line(11:11)
      last_name(1:15) = line(12:26)
      write (*,10) first_name, middle_initial, last_name
10    format (A,1X,A,1X,A)
      go to 5

*  Close the file.  The phrase 'unit=' is optional since the unitid
*  argument (10) is the first in the statement.
100   close(unit=10, iostat=closestat, status='keep')
      if (closestat .ne. 0) then
      call error_$print(closestat)
      endif
      end
```

USING THIS EXAMPLE

This program is available online and is named **close_example**. You get the following output if you run the program:

```
Here are the names:
Stewart    M Franklin
Kayla      J Brady
Pierre     Y Giroux
Maddie     A Hayes
Sterling   R Gillette
Ilsa       L Lazlo
```

common Defines named or unnamed **common** storage areas and lists the variables and arrays to be stored in those areas.

FORMAT

common [*/common_name1/*] *list1* [. . . , [*/common_nameN/*] *listN*]

ARGUMENTS

common_name — The name of this **common** area. If you give a **common** area a name, enclose the name in a pair of slashes (/ /). You can optionally use an empty pair of slashes to indicate an unnamed **common** area.

list — The variables and arrays to be stored in this **common** area. If you specify more than one *list* item, separate them with commas.

DESCRIPTION

The **common** statement defines one or more named or unnamed storage areas to be used in common by different program units, and identifies the variables or arrays to be stored in those areas.

Common also causes an additional area to be created for a program. When you compile a program, its statements are stored in one area and its data in another. If you specify that you want some variables to be stored in **common**, FORTRAN creates a third area. Figure 4–3 shows the two structures.

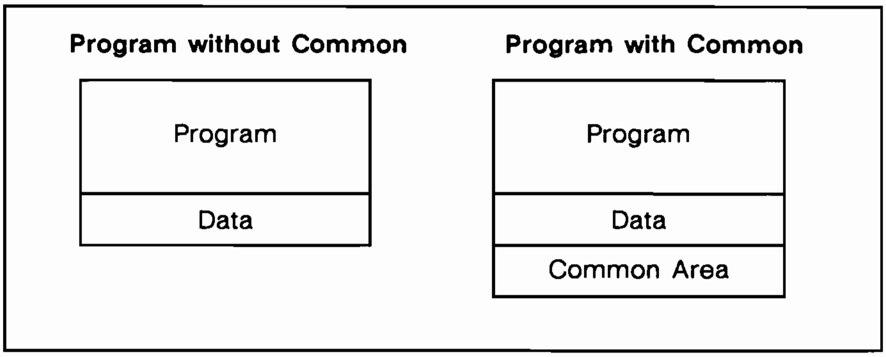

Figure 4–3. Program without and with **Common**

Common blocks enable various program units to share the same storage area. That is, when you declare common blocks with the same name in different program units (for example, the main program and several subroutines) these blocks all share the same storage area.

Although Domain FORTRAN does not require it, a named **common** block should have the same data type storage sequence and length in different program units. That's because the variables and arrays listed for the **common** block are assigned storage on a one-to-one basis. If the corresponding variables are of different types, your program may behave unpredictably. This is the correct way to declare **common**:

```
integer*4    first, second
logical      why
character*15 last_name(5)
common /hold/ first, why, second, last_name
    .
    .
    .
subroutine mine()
integer*4    third, fourth
character*15 name_last(5)
logical      because
common /hold/ third, because, fourth, name_last
```

Notice that the data types for corresponding variables and arrays match in the **common** block named **hold**. That is, **first** and **third** are both **integer*4** variables, **why** and **because** are both **logical** variables, and so on.

Unnamed **common** blocks need not have the same length in all program units. You may only have one unnamed **common** block per program unit.

As an extension to the ANSI standard, Domain FORTRAN allows a common block (named or unnamed) to contain both character and noncharacter data. In addition, named common blocks can overlay Domain Pascal data sections (refer to Section 7.3).

Typically, you use **block data** subprograms to initialize data that is in **common** (although Domain FORTRAN allows you to declare named and unnamed **common** blocks in any program unit). You can create an insert file of **common** declarations, and then use either the **include** statement or the **%include** directive to include that file in your source code. (See the descriptions of **include** and **%include** later in this encyclopedia. Note that **%include** is listed under the Compiler Directives section.)

Although Domain FORTRAN stores elements within a **common** block contiguously, two or more **common** blocks may not necessarily occupy contiguous storage.

A variable or array name can appear in both a **common** and an **equivalence** statement in the same program unit. However, there are some restrictions. For information on those restrictions, see the entry for **equivalence** later in this encyclopedia.

NOTE: If you compile your program with **f77**, or if you use the **-uc** switch with **ftn**, the compiler appends an underscore (_) to the name of a **common** block whenever the name does not start or end with an underscore (_) and whenever it does not contain a dollar sign ($). The underscore distinguishes the block from a C procedure or external variable with the same user-assigned name. This feature is useful if you plan to call C programs from Domain FORTRAN. Refer to Section 6.5.34 for details about the **-uc** option.

Data Alignment in Common Blocks

You must be careful about data alignment when setting up a **common** block that contains character variables with an odd number of bytes. If a **common** list contains a character variable consisting of an odd number of bytes and it is followed by a noncharacter variable, that second variable will be aligned on an odd-byte boundary, causing the compiler to issue an error message. The solution is to arrange **common** blocks so that character variables always appear last in the list, as in the preceding example.

The semantics of FORTRAN require that, when laying out objects in a **common** block, the compiler stores them in contiguous memory locations. The compiler always sets the first object in the block on its natural boundary, but it can guarantee only word alignment for succeeding objects that are larger than a character. For example, in a **common** block containing both **integer*2** objects and **integer*4** objects, if the smaller object is first, it is possible that the compiler will lay out an **integer*4** object on a word boundary, making it not naturally aligned.

The following declarations illustrate how a **common** block can cause alignment problems:

```
integer*4    four_byte
integer*2    two_byte
logical      one_byte
common /not_naturally_aligned/ two_byte, four_byte, one_byte
```

Given this arrangement of data in the **common** block, the compiler can only guarantee natural alignment for **two_byte**. The layout rules require that the compiler arrange **four_byte** on the next available word boundary, not the 4-byte boundary it needs if it's to be naturally aligned.

There are several ways to correct this problem. The best way—the way that is good programming practice in any case—is to arrange data in the **common** block from largest to smallest. This arrangement will always guarantee natural alignment for each object. Taking this approach with the example **not_naturally_aligned**, you would arrange the **common** block as follows:

```
integer*4     four_byte
integer*2     two_byte
logical       one_byte
common /naturally_aligned/ four_byte, two_byte, one_byte
```

A second approach to ensuring natural alignment within a **common** block is to insert empty storage—padding—before objects that would otherwise not be naturally aligned. In the example **common** block **not_naturally_aligned**, you could force the compiler to lay out **four_byte** on a 4-byte boundary by inserting a two-byte padding just before it, as follows:

```
integer*4     four_byte
integer*2     two_byte, pad
logical       one_byte
common /pad_aligned/ two_byte, pad, four_byte, one_byte
```

You can let the compiler insert the padding for you by compiling the program with the **-natural** option (refer to Subsection 6.5.24). This option applies only to **common** blocks, and its effect is to insert padding (also called "holes") where appropriate to force natural alignment. One difference between inserting padding by hand and letting the compiler do it for you is that, when the compiler does it, you can't access the "holes".

You must take precaution when using the compiler's **-natural** option: when compiling a program that consists of several source files that communicate through a **common** block, if you compile one of the source files with the **-natural** option, you must be sure to compile the others with the **-natural** option as well. If you don't, the objects in the **common** blocks will be inconsistently aligned and produce bad data. This problem is compounded by the fact that the compiler cannot detect the inconsistent alignment. Therefore, when compiling a multiple source-file program, use the **-natural** option either on all the modules or on none of them.

The issue of natural alignment has the greatest impact on programs that will run on a Series 10000 workstation, although programs written for any Apollo workstation are likely to run better if their data is naturally aligned. For a full discussion of natural alignment issues and the Series 10000 workstation, refer to the *Series 10000 Programmer's Handbook*.

EXAMPLE

```
*  This program uses a common block to compute permutations; that is
*  the number of permutations for n things taken m at a time.  The
*  mathematical formula is: P(n,m) = n!/(n-m)!

      program common_example

      integer*2 n, m
      integer*4 perm
      common /numbers/ n, m, perm   {put variables in a common block}

5     print 10
      print 20
10    format ('Enter the numbers for the permutation (n things')
20    format ('taken m at a time) separated by a space: ', $)
      read *, n, m
      if (m .gt. n) then            {make sure input data is valid}
          print 30
30        format('First number must be greater than the second.', /)
          goto 5
      endif

*  Call subroutine that computes the permutation.
      call permute()
      print 40, n, m, perm
40    format (I2, ' things taken ', I2, ' at a time is ', I5)

      end
**********************************************************************
      subroutine permute()

*  Declare variables - notice the ones in the common block have
*  different names from those in the main program unit - and specify
*  that they are in the common storage area, "numbers".

      integer*2 things_taken, at_a_time, i
      integer*4 result, total1, total2
      common /numbers/ things_taken, at_a_time, result
      data total1, total2 /1,1/

      do i = 1, things_taken                {compute n!    }
          total1 = total1 * i
      enddo
      do i = 1, (things_taken - at_a_time)  {compute (n-m)!}
          total2 = total2 * i
      enddo
      result = total1/total2

      end
```

USING THIS EXAMPLE

This program is available online and is named **common_example**. Following is a sample run of the program:

```
Enter the numbers for the permutation (n things
taken m at a time) separated by a space: 3 4
The first number must be greater than the second.

Enter the numbers for the permutation (n things
taken m at a time) separated by a space: 4 3
 4 things taken  3 at a time is    24
```

DESCRIPTION

The Domain FORTRAN compiler understands the directives shown in Table 4-8. All directives except **%config** must begin in column 1, and the first character for all except **debug** must be a percent sign (%). To use a directive, specify its name after the percent sign.

Table 4-8. Compiler Directives

Directives	Action
%begin_inline	Marks the beginning of a subprogram that you want expanded to inline code.
%end_inline	Marks the end of a subprogram that you want expanded to inline code.
%begin_noinline	Marks the beginning of a subprogram that you do *not* want expanded to inline code.
%end_noinline	Marks the end of a subprogram that you do *not* want expanded to inline code.
☆ **%config**	Lets you easily set up a warning message if you forget to compile with the **-config** compiler option.
debug	Directs Domain FORTRAN to compile lines prefixed by this directive (or simply with D or d) when you use the **-cond** compiler option. If you don't use **-cond** when you compile, prefixed lines don't get compiled.
%eject	Directs Domain FORTRAN to put a formfeed in the listing file at this point.
☆ **%else**	Specifies that a block of code should be compiled if a preceding **%if** *predicate* **%then** directive is false.
☆ **%elseif** *predicate* **%then**	Directs the compiler to compile the code up until the next **%else**, **%elseif**, or **%endif** directive, if and only if the *predicate* is true.
☆ **%elseifdef** *predicate* **%then**	Checks whether additional *predicate*s have been declared with a **%var** directive.
☆ **%enable**	Sets compiler directive variables to true.
☆ **%endif**	Marks the end of a conditional compilation area of the program.

☆ is a directive described in "Directives Associated with the **-config** Option," following this table.

(Continued)

Table 4-8. Compiler Directives (Cont.)

Directives	Action
☆ **%error** *'string'*	Prints *'string'* as an error message whenever you compile.
☆ **%exit**	Directs the compiler to stop conditionallly processing the file.
☆ **%if** *predicate* **%then**	Directs the compiler to compile the code up until the next **%else**, **%elseif** or **%endif** directive, if and only if the *predicate* is true.
☆ **%ifdef** *predicate* **%then**	Checks whether a *predicate* was previously declared with a **%var** directive.
%include *'pathname'*	Causes Domain FORTRAN to read input from the specified file.
%line = *line* ['*pathname*']	Sets the current line number of source file.
%list	Enables the listing of source code in the listing file.
%nolist	Disables the listing of source code in the listing file.
☆ **%var**	Lets you declare variables that you can then use as predicates in compiler directives.
☆ **%warning** *'string'*	Prints *'string'* as a warning message whenever you compile.
☆ is a directive described in "Directives Associated with the **-config** Option."	

DIRECTIVES ASSOCIATED WITH THE -CONFIG OPTION

This subsection describes the following compiler directives: **%if**, **%then**, **%elseif**, **%else**, **%endif**, **%ifdef**, **%elseifdef**, **%var**, **%enable**, **%config**, **%error**, **%warning**, and **%exit**.

The conditional directives mark regions of source code for conditional compilation. This feature allows you to tailor a source module for a specific application. You invoke conditional processing by using the **-config** option when you compile.

Several of the directives take a predicate. A predicate can consist of special variables (declared with the **%var** directive) and the optional keywords **not**, **and**, or **or**. Given that **color** and **mono** are special variables, here are some possible predicates:

- **color**
- **not(color)**
- **mono or color**
- **(mono and color)**

%if *predicate* **%then**

If the *predicate* is true, Domain FORTRAN compiles the code after **%then** and before the next **%else**, **%elseif**, or **%endif** directive.

For example, to specify that a block of code is to be compiled for a color node, you might choose an attribute name such as **color** to be the predicate. Then write:

```
%var color {Tell the compiler that 'color' can be used in a predicate}
.
.
.
%if color %then

   code

%endif
```

To set **color** to true, you can either use the **%enable** directive in your source code or the **-config** option in your compile command line.

%else

The **%else** directive is used in conjunction with **%if** *predicate* **%then**. The **%else** directive specifies a block of code to be compiled if the predicate in the **%if** *predicate* **%then** clause evaluates to false. For example, consider the following fragment:

```
%var color  {Tell the compiler that 'color' can be used in a predicate.}
.
.
.
%if color %then

   code

%else        {Compile this code if color is false.}

   code

%endif
```

%elseif *predicate* **%then**

The **%elseif** *predicate* **%then** directive is used in conjunction with **%if** *predicate* **%then**. Its purpose is like that of the FORTRAN statement

else if (*cond*) **then**
    statement

For example, suppose you want to compile one sequence of statements if the program is going to run on a color node, and another sequence of statements if the program is going to run on a monochromatic node. To accomplish that, you could organize your program in the following way:

```
%var color mono {Tell the compiler that 'color' and 'mono' can be }
                {used in a predicate.                             }
.
.
.
%if color %then     {Compile the following code if color is true.}
   Code for color nodes

%elseif mono %then  {Compile the following code if mono is true.}
   Code for monochromatic nodes

%endif
```

To set **color** or **mono** to true, you can either use the **%enable** directive in your source code or the **–config** option in your compile command line. If **color** and **mono** are both true, Domain FORTRAN compiles the code for color nodes since it appears first. Note that you can put multiple **%elseif** directives in the same block.

%endif

The **%endif** directive tells the compiler where to stop conditionally processing a particular area of code.

%ifdef *predicate* **%then**

Use **%ifdef** *predicate* **%then** to check whether a variable has already been declared with a **%var** directive. If you accidentally declare the same variable more than once, Domain FORTRAN issues an error message; **%ifdef** is a way of avoiding this error message. The **%ifdef** directive is especially helpful when you don't know if an include file declares a variable.

For example, consider the following use of **%ifdef**:

```
%include 'bitmap_init.ins'{Source code that may or may not have }
                          {used %var to declare the variable 'color'.}

%ifdef not(color) %then   {If color has not been declared with }
    %var color            {%var, declare it now.               }
%endif
```

The difference between **%if** and **%ifdef** is the following. Variables in an **%if** predicate are considered true if you set them to true with **%enable** or **-config**; however, variables in an **%ifdef** predicate are considered true if they have been declared with **%var**.

%elseifdef *predicate* **%then**

%elseifdef is to **%ifdef** as **%elseif** is to **%if**. Use **%elseifdef** *predicate* **%then** to check on whether additional variables were declared with **%var**; for example:

```
%include 'bitmap_init.ins' {Source code that may or may not have }
                           {used %var to declare the variables   }
                           {'color' or 'mono.'                   }

%ifdef not(color) %then    {If color has not been declared with  }
    %var color             {%var, declare it now.                }

%elseifdef not(mono) %then {Otherwise, if mono has not been      }
    %var mono              {declared with %var, declare it now.  }

%endif
```

%var

The **%var** directive lets you declare variable and attribute names that will be used as predicates later in the program. You cannot use a name in a predicate unless you first declare it with the **%var** directive. The following example declares the names **code.old** and **code.new** as predicates:

```
%VAR  code.old  code.new
```

The compiler preprocessor issues an error if you attempt to declare with **%var** the same variable more than once. (Use **%ifdef** or **%elseifdef** to avoid this error.)

%enable

Use the **%enable** directive to set a variable to true. (The **%enable** directive and the **-config** compiler option perform the same function.) You create variables with the **%var** directive. If you do not specify a particular variable in an **%enable** directive or **-config** option, Domain

FORTRAN assumes that the variable is false. For example, the following example declares three variables—**code.sr9.5**, **code.sr9**, and **code.sr8**—and sets **code.sr9.5** and **code.sr8** to true:

```
%var    code.sr9.5  code.sr9  code.sr8
%enable code.sr9.5  code.sr8
```

The compiler issues an error message if you attempt to set (with **%enable** or **-config**) the same variable to true more than once.

%config

The **%config** directive is a predeclared attribute name. You can only use **%config** in a predicate. The Domain FORTRAN preprocessor sets **%config** to true if your compiler command line contains the **-config** option, and sets **%config** to false if your compiler command line does not contain the **-config** option. The purpose of the **%config** directive is to remind you about using the **-config** option when you compile; for example:

```
%if color %then
   .
   .
   .
   {This is the code for color nodes.}
   .
   .
   .
%elseif mono %then
   .
   .
   .
   {This is the code for monochromatic nodes.}
   .
   .
   .
%elseif %config %then
   %warning('You did not set color or mono to true.');
%endif
```

> **NOTE:** Do not attempt to declare **%config** in a **%var** directive.

%error *'string'*

This directive causes the compiler to print '*string*' as an error message. You must place this directive on a line by itself. For example, suppose you want the compiler to print an error message whenever you compile with the **-config mono** option. In that case, set up your program like this:

```
%var color mono

%if color %then
   .
   .
   .
   {code for color node.}
   .
   .
   .
%elseif mono %then
   %error 'I have not finished the code for a monochromatic node.'
%endif
```

If you then compile with the **-config mono** option, Domain FORTRAN prints the following error message:

```
(00005)     %error 'I have not finished the code for a monochromatic
node.'
**** Error #91 on Line 5: Conditional compilation user error
1 errors, no warnings in $MAIN, Fortran version n.nn
```

> **NOTE:** Because of the error, Domain FORTRAN does not create an executable object.

%exit

%exit directs the compiler to stop conditionally processing the file. For example, if you put **%exit** in an **include** file, Domain FORTRAN only reads in the code up until **%exit**. It ignores the code that appears after **%exit**. (See the descriptions of **include** and **%include** later in this encyclopedia for details about **include** files. Note that **%include** is listed under the Compiler Directives section.)

%exit has no effect if it's in a part of the program that does not get compiled.

%warning *'string'*

This directive causes the compiler to print *'string'* as a warning message. You must place this directive on a line by itself. For example, suppose you want the compiler to print a warning message whenever you forget to compile with the **-config color** option. In that case, set up your program like this:

```
%var color mono

%if color %then
   .
   .
   .
   {code for color node.}
   .
   .
   .
%else
   %warning 'You forgot to use the -config color option.'
%endif
```

Then, if you don't compile with the **-config color** option, Domain FORTRAN prints the following message:

```
(00005)      %warning 'You forgot to use the -config color option.'
**** Warning #90 on Line 5: Conditional compilation user warning
no errors, 1 warnings in $MAIN, Fortran version n.nn
```

A warning does not prevent the compiler from creating an executable object.

DIRECTIVES NOT ASSOCIATED WITH THE -CONFIG OPTION

The remaining compiler directives are not specifically associated with the **-config** compiler option.

%begin_inline and %end_inline

The **%begin_inline** and **%end_inline** compiler directives are delimiters for defining subprograms for inline expansion. Inline expansion means that the compiler generates code for a given subprogram wherever a call to that subprogram appears. Inline expansion of a given subprogram allows you to avoid the overhead of calling a subprogram. When used with small subroutines, this can increase execution speed.

Follow these rules when using **%begin_inline** and **%end_inline**:

- Place **%begin_inline** on a line in the source file before you begin any appropriate subprogram definitions.
- Place **%end_inline** on the line following the last subprogram that you define for inline expansion.
- Begin both directives in column 1.

Suppose that a program contains this function declaration:

```
*
%begin_inline
*
*     Function to test if a real number is positive
*     Input argument: number
*     Output: true or false

      logical function test_pos (number)
*
      test_pos = number .ge. 0.0

      return
      end
*
%end_inline
```

Whenever you call the function **test_pos** in your main program, the compiler generates code for the function at that point, instead of transferring control to the function.

%begin_noinline and %end_noinline

The compiler directives **%begin_noinline** and **%end_noinline** are delimiters for subprograms that the compiler never expands inline. These delimiters are the converse of **%begin_inline** and **%end_inline**. They are used when you compile a program with either the **–opt** 3 or **–opt** 4 compiler option but you want to selectively disable inline expansion of certain subprograms. (Refer to subsection 6.5.26 for information on how **–opt** 3 and **–opt** 4 affect inline subprogram expansion.)

Follow these rules when using **%begin_noinline** and **%end_noinline**:

- Place **%begin_inline** on a line in the source file before you begin any appropriate subprogram definitions.
- Place **%end_inline** on the line following the last subprogram that you define for inline expansion.
- Begin both directives in column 1.

Suppose that the program in the following example were compiled with the **-opt 4** option, which causes the compiler to perform inline expansion of all subprograms:

```
      program do_nothing
      integer x, i, y, a
      foo(i) = i + 1
      call bar(a)
      print*, a
      y = foo(1)
      print*, y
      end
%begin_noinline
      subroutine bar(a)
      integer a
      a = 3
      return
      end
%end_noinline
```

The compiler will expand the statement function foo to inline code, but not subroutine bar, which is delimited by the **%begin_noinline** and **%end_noinline** compiler directives.

debug

NOTE: Do not precede the **debug** compiler directive with a percent sign (%). It is an error if you do.

The **debug** directive marks source code for conditional compilation. The "condition" is the compiler switch **-cond**. If you do compile with the **-cond** option, Domain FORTRAN compiles the lines that begin with **D** or **d**. If you do not compile with the **-cond** switch, Domain FORTRAN does not compile the lines that begin with **D** or **d**.

The following fragment illustrates how to use the **debug** directive:

```
       value = data + offset
d      print *, 'Current value is: ', value
```

The preceding fragment contains one **debug** directive. If you compile with the **-cond** option, the system executes the **print** statement at run time. If you compile without the **-cond** option, the system does not execute the **print** statement at run time. Therefore, you can compile with the **-cond** option until you are sure the program works the way you want it to work, and then compile without the **-cond** option to eliminate the (now) superfluous **print** message.

The **debug** directive applies to one physical line only, not to one Domain FORTRAN statement. Therefore, in the following example **debug** applies only to the **do** clause. If you compile with **-cond**, Domain FORTRAN compiles all three statements. If you compile without **-cond**, Domain FORTRAN compiles only the **print** and **continue** statements (and thus there is no loop).

```
D     do 10 j = 1,max_size  {Notice shorter form of debug directive.}
         print *, iarray(j)
10    continue
```

%eject

The **%eject** directive does not affect the **.bin** file; it only affects the listing file. (The **-l**, **-exp**, and **-xref** compiler options cause the compiler to create a listing file.) The **%eject** directive specifies that you want a page eject (formfeed) in the listing file. The statement that follows the **%eject** directive appears at the top of a new page in the listing file.

%include *'pathname'*

Use the **%include** directive to read in a file (*'pathname'*) containing Domain FORTRAN source code. This file is called an **include** file. The compiler inserts the file where you placed the **%include** directive.

Many system programs use the **%include** directive to insert global type, subprogram, and function declarations from common source files, called **insert** files. The Domain system supplies insert files for your programs that call system routines. The insert files are stored in the **/usr/include** directory; see subsection 7.8.1 for details.

A program can contain a maximum of 1200 **%include** statements.

Domain FORTRAN allows you to nest include files. That is, an include file can itself contain an **%include** directive. The include file nesting limit is 16 files.

An include file may *not* consist of an entire program unit. For example, it is not permissible to create an entire subprogram in one file and then use **%include** to insert that subprogram into your Domain FORTRAN program. At least one line of the subprogram must appear in the program unit in which you're using the **%include** appears. If you try to include an entire program unit in your source file, you get a compile-time error.

The compiler option **-idir** enables you to select alternate pathnames for insert files at compile time. See subsection 6.5.15 for details.

This directive has no effect if it's in a part of the program that does not get compiled.

%line = *line_number* [*'pathname'*]

The **%line** compiler directive allows you to set the compiler's knowledge of the current source line number and, optionally, the current source file. You must place this directive on a line by itself. *line_number* is an integer constant that specifies the new line number; *pathname*, in single or double quotes, is the new name of the source file.

The compiler reports errors in terms of the line numbers set by this option. In addition, the debugger line-number table is built with these line numbers. The debugger source file option is given the last filename in the source, if that file exists. (The compiler verifies the existence of the source file before it creates the debug entry.) Most programmers are unlikely to need the **%line** directive, and it is probably useful only in a few unusual situations:

- If your source file is extremely large and you want to break it up into smaller units for separate compilation, you could use **%line** directives at the start of each smaller file to renumber the lines of the smaller file in accordance with the line numbering of the original file, and to make the filename that of your original file. If you use the debugger, the debugger will point to the lines of your original file, so that you can revise and maintain your original source file.

  For example, the following **%line** directive makes the subroutine start at line 105 and gives it another filename:

  ```
  %line = 105 'anotherfile.ftn'
        subroutine line_example

        integer*4 i, j
        i = 5
        j = i * 5
        print *, i, j
        return
        end
  ```

  The listing file for the subroutine shows the changed line numbering:

  ```
  (00001) %line = 105 'anotherfile.ftn'
  (00105)         subroutine line_example
  (00106)
  (00107)         integer*4 i, j
  (00108)         i = 5
  (00109)         j = i * 5
  (00110)         print *, i, j
  (00111)         return
  (00112)         end
  ```

- If you use a software engineering tool that changes your source code—for instance, a tool that restructures loops—you can use **%line** directives so that the line numbers of unchanged parts of the revised code are the same as the line numbers of the original code.

%list and **%nolist**

The **%list** and **%nolist** directives do not affect the **.bin** file; they only affect the listing file. (The **-l** compiler option causes the compiler to create a listing file.) **%List** enables the listing of source code in the listing file, and **%nolist** disables the listing of source code in the listing file. For example, the following sequence disables the listing of the two insert files, and then re-enables the listing of future source code:

```
. . .
%nolist
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/error.ins.ftn'
%list
. . .
```

%list is the default.

continue Marks a place in the program unit for a statement label.

FORMAT

continue

ARGUMENTS

None.

DESCRIPTION

Continue marks a place for a statement label; it has no effect on the program itself. You can use a **continue** statement anywhere in a program; execution simply continues with the next statement.

A **continue** is often used as a terminating statement for a **do** loop, in which case it must be labeled. Otherwise, a **continue** statement does not need to be labeled.

See **do** later in this encyclopedia for more information about **do** loops.

EXAMPLE

```
*  This program shows how to use a continue statement to terminate
*  a do loop.  The loop itself takes a number, doubles it, and
*  prints out the result.
      program continue_example

      integer*4 i, double, old_double         {Declare and          }
      old_double = 1                          {initialize variables.}

*  Double and print number

      do 20 i=1,10
          double = old_double*2
          print 50, old_double, double
          old_double = double
20    continue                                {Close do loop.}
50    format('If you double ', I3, ' you get ', I4)
      end
```

USING THIS EXAMPLE

This program is available online and is named **continue_example**. Following is a sample run of the program:

```
If you double   1 you get    2
If you double   2 you get    4
If you double   4 you get    8
If you double   8 you get   16
If you double  16 you get   32
If you double  32 you get   64
If you double  64 you get  128
If you double 128 you get  256
If you double 256 you get  512
If you double 512 you get 1024
```

data Sets initial values of variables, arrays, array elements, and substrings.

FORMAT

data *var_list1/constant_list1/* [. . . ,*var_listN/constant_listN/*]

ARGUMENTS

var_list — The names of the variables, arrays, array elements, or substrings to be assigned values. Separate the items in *var_list* with commas.

constant_list — A list of numerical, logical, or character constants, each of which corresponds to an item in *var_list*. Separate the constants with commas, and enclose the list in slashes (/ /).

DESCRIPTION

Data provides a shortcut for initializing variables, arrays, array elements, or substrings. It is a nonexecutable statement that is nearly equivalent to the assignment statement *var* = *exp*, where *var* is a variable and *exp* is the value assigned to that variable. The difference is that with **data**, initialization occurs before execution starts. (See "Assignment Statements" earlier in this section for more on such statements.)

There must be a one-to-one correspondence between the items in the *var_list* and the values in the *constant_list*. In the following fragment

```
integer*4 hi_temp, low_temp
real      test_data(10,10)

data hi_temp, low_temp, test_data(1,3)/65, 48, 123.945/
```

the **data** statement is equivalent to these arithmetic assignment statements:

```
hi_temp = 65
low_temp = 48
test_data(1,3) = 123.945
```

If you want to initialize multiple array elements to the same constant, you may use a repeat count within the *constant_list*. It takes this form:

*num*constant*

where ***num*** is the number of times you want to repeat *constant*. For example,

```
integer interest_rates(12,3)
data interest_rates/36*0/
```

assigns the constant 0 to each element in the 36–element array **interest_rates**. It is an error if ***num*** does not match the number of elements in the array.

If the data type in *var_list* does not match that of its corresponding entry in *constant_list*, FORTRAN converts the constant to the appropriate data type. So if the *var_list* entry is **integer** and the constant is **real**, FORTRAN truncates the constant (removes the decimal point and all digits to the right of the decimal point). Likewise, if the variable is a **real** and the constant is **integer**, FORTRAN adds a decimal point after the rightmost digit. Character constants are padded with blanks on the right or truncated from the right, if necessary, to match the lengths of their corresponding character variables.

You can use implied **do** loops with **data** statements. For example:

```
integer i, j, matrix(10,5)
data ((matrix(i,j),j=1,2),i=5,10)/12*-1/
```

This **data** statement assigns –1 to each of the 12 elements in **i** and **j**'s specified ranges for array **matrix**.

If you use **data** statements to assign values to local variables in subprograms, those variables are allocated static storage, and consequently, retain their values from one invocation of the subprogram to the next.

EXAMPLE

```
*  This program uses the data statement to initialize all the elements
*  of a 4x5 integer array.

      program data_example
      integer i, j, matrix(4,5)

*  Use implied do loops in data statement to initialize
*  each column of the array, matrix
      data ((matrix(i,j), j=1,1), i=1,4)/4*1/
      data ((matrix(i,j), j=2,2), i=1,4)/4*2/
      data ((matrix(i,j), j=3,3), i=1,4)/4*3/
      data ((matrix(i,j), j=4,4), i=1,4)/4*4/
      data ((matrix(i,j), j=5,5), i=1,4)/4*5/

*  Print out matrix
      do i=1,4
          print *, (matrix(i,j), j=1,5)
      enddo
      end
```

USING THIS EXAMPLE

This program is available online and is named **data_example**. If you run the program, you get the following output:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

decode Transfers data from memory to iolist items. (Extension)

FORMAT

decode (*len, fmt, var*) *iolist*

ARGUMENTS

len An unsigned **integer** constant, **integer** variable, or **integer** expression specifying the record length (in bytes) of the target memory area.

fmt Format specification for the data: can be any one of the following:

- The label of a **format** statement in the same program unit.
- An **integer*4** variable previously assigned a **format** statement label via the **assign** statement.
- A character array or character expression that contains a format string. In a character expression, a format string must be within parentheses and enclosed within single quotes; for example, '(I5)'.

List-directed formatting is legal in a **decode** statement.

var A variable, array, or array element specifying the beginning of the target memory area.

iolist Input/Output List: the entities (for example, variable names) to which the data will be transferred.

DESCRIPTION

The **decode** statement, a Domain FORTRAN extension, formats data that is stored in memory and transfers it to one or more *iolist* entities.

Decode and **encode** are typically used to read and write character data in large memory areas that are declared as numeric types. These statements are analogous to reads and writes to internal files, as specified in the ANSI 1978 FORTRAN standard. (See the descriptions of **read** and **write**.)

The **decode** arguments *len* and *var* define the byte length of the target memory area and its starting address, respectively. The *fmt* argument specifies a format for the data. **Decode** edits the data according to the specified format and then assigns it to the **iolist** entities.

FORTRAN assumes that the target memory area is a "*len*-length" record. If *len* is less than the record length that the *fmt* argument implies, FORTRAN treats the record as if it were padded with blanks to equal the length *fmt* does imply. For example, in this fragment

```
      decode (20,10,line) nums
10    format (5f6.2)
```

the **decode** statement specifies a *len* of 20. But the **format** statement implies a *len* of 30 (because it specifies a 6-digit number repeated five times). FORTRAN simply acts as if the record **nums** has a *len* of 30, and continues.

If you specify the data format within the **decode** statement, instead of giving the label of a statement at which the format is defined, you must enclose the format in parentheses within single quotes. For example:

```
      decode (40,'(5f4.2)',in_data) decode_data
```

Encode, a complementary statement to **decode,** transfers data from an **iolist** to memory. See the description of **encode** later in this encyclopedia.

EXAMPLE

```
*  This program illustrates the decode and encode statements.  The
*  decode statement translates the data in LINE into an internal
*  form, storing it in the variables OPEN, HIGH, LOW, and CLOSE.
*  The encode statement re-translates the data in those variables
*  back into character format, storing it in the character array
*  NEW_LINE.

      PROGRAM DECODE_ENCODE_EXAMPLE

      CHARACTER*40 LINE, NEW_LINE
      REAL         OPEN, HIGH, LOW, CLOSE
      DATA LINE/'  6301  6550  6285  6350'/

*  Decode line specifies that the length is 40, the format statement
*  is at the statement labeled 10, the variable LINE is where the
*  data is kept, and the data should be put in the four listed
*  variables.

      DECODE (40, 10, LINE) OPEN, HIGH, LOW, CLOSE
10    FORMAT (4F6.2)
      WRITE (*, 20) OPEN, HIGH, LOW, CLOSE
20    FORMAT (4F8.3)

*  Encode line specifies that the length is 40, the format string
*  is 4F7.3, that the data is to be stored in the character array
*  NEW_LINE, and that the data is to be transferred from the four
*  listed variables.

      ENCODE (40, '(4F7.3)', NEW_LINE) OPEN, HIGH, LOW, CLOSE
      PRINT *, NEW_LINE

      END
```

USING THIS EXAMPLE

This program is available online and is named **decode_encode_example**. If you run the program, you get the following output:

```
  63.010  65.500  62.850  63.500
 63.010 65.500 62.850 63.500
```

discard Calls a function as a subroutine. (Extension)

FORMAT

discard (*function_call*)

ARGUMENT

function_call Any valid function-call statement.

DESCRIPTION

The discard statement allows you to call a function as a subroutine. The effect is that the system ignores and discards the function's return value.

The discard statement is useful when you need to call a function for its side effects rather than for the value it returns. In this situation, you can discard a value that you do not use and avoid a compiler warning.

EXAMPLE

```
* This program uses the discard statement to call a function
* (ec2_$wait) and ignore its return value.
      program discard_example

      include '/sys/ins/base.ins.ftn'
      include '/sys/ins/ec2.ins.ftn'
      include '/sys/ins/time.ins.ftn'

      integer*2 ec_ptr(3), ec_count
      integer*4 event_ptr, status, ec_value
      pointer /event_ptr/ ec_ptr

* Ask the system to give us the address of the quarter-second timer
      call time_$get_ec (time_$clockh_key, event_ptr, status)
*
      if (status.ne.0) then
         call error_$print (status)
      else
         print *, 'I shall wait now!'
* Get the current value of the timer, and add 15 seconds to it
         ec_value = ec2_$read (ec_ptr)
         ec_value = ec_value + 15*4     { Wait ~ 15 seconds }
```

```
* Wait for the timer to expire; we don't care about the return value
* of ec2_$wait, so ignore the return value
          ec_count = 1
          DISCARD (ec2_$wait (event_ptr, ec_value, ec_count, status))
*
          if (status.ne.0) then
              call error_$print (status)
          else
              print *, 'It is now 15 seconds later'
          endif
      endif

      stop
      end
```

USING THIS EXAMPLE

This program is available online and is named **discard_example**. If you run the program, you get the following output:

```
I shall wait now!
It is now 15 seconds later
```

do Executes a block of statements zero or more times.

FORMAT

do [*label*] [,] *var* = *init_value*, *last_value* [,*inc_value*]

ARGUMENTS

label — Optional statement label indicating the end of the do loop range. (The comma after *label* is optional.)

var — **Do** loop index variable; can be of type **integer*2**, **integer*4**, **real**, or **double precision**.

init_value — Initial value of the **do** loop index variable; can be a constant, variable, or expression of type **integer*2**, **integer*4**, **real**, or **double precision**.

last_value — Limit value of the **do** loop index variable; can be a constant, variable, or expression of type **integer*2**, **integer*4**, **real**, or **double precision**.

inc_value — Increment value; can be a constant, variable, or expression of type **integer*2**, **integer*4**, **real**, or **double precision**. The index variable is increased by *inc_value* after each pass of the **do** loop. If you omit this, the increment value is 1.

DESCRIPTION

Do begins the execution of a loop that extends from the **do** statement to a labeled terminal statement or to an unlabeled **end do** statement. The terminal statement must be in the same program unit as the **do**.

If you omit the terminal statement label (*label*), you must use an unlabeled **end do** as the terminal statement. Otherwise, the terminal statement can be a labeled **end do** or any other executable statement *except*:

- **go to** (unconditional or assigned)
- **if** (arithmetic or block)
- **else if**
- **else**
- **end if**
- **return**
- **stop**

- **end**
- another **do**

Continue and **end do** are frequently used as terminal statements of a **do** range.

After the specified number of passes through the **do** loop, control goes to the next statement after the terminal statement.

The index variable *var* and the variables *init_value*, *last_value*, and *inc_value* (if present) dictate how many times the **do** loop executes. The index variable is initialized to *init_value*, and the loop continues executing until the value of *var* is greater than *last_value*.

The increment variable, *inc_value*, determines the amount by which the index variable changes with each pass of the loop. If you leave this variable out, the index increases by one each time. For example:

```
do i=1,20      {no inc_value, so i increases by 1 each time}
    .
    .
    .
end do

do i=0,20,2    {inc_value is 2, so i increases by 2 each time}
    .
    .
    .
end do
```

You cannot assign a new value to the index variable within a **do** loop. The value of the index variable automatically changes with each pass of the loop.

If you compile using **-opt 3** or **-opt 4**, the optimizer substitutes a register for *var* if both of the following conditions are true:

- It can safely eliminate all uses of *var* in the loop.
- It determines that there are no future uses of *var*.

Because FORTRAN tests **do** loop conditions before executing the loop, it's possible to construct loops that are never executed. For example:

```
      a = 3
       .
       .
       .
      do 40, i=10, a*4-3
       .
       .
       .
40    continue
```

In this case, *last_value* evaluates to 9 (a*4-3), while *init_value* is 10. Since *last_value* is less than *init_value*, the loop is not executed.

A **do** loop can contain any number of nested **do** loops, provided the range of each inner loop does not extend beyond the range of the outer loops. Here are some examples of properly and improperly nested loops:

```
*  Properly nested loops                    *  Improperly nested loops
      do 20, i=1,10                               do 20, i=1,10
          do 10, j=1,5                                do 10, j=1,5
              .                                           .
              .                                           .
              .                                           .
10        continue {close j loop}               20      continue
20    continue     {close i loop}               10 continue
```

Although you need to be careful that the range of an inner **do** loop does not extend *past* that of an outer loop, the two loops can end at the same place. For example, the following construction is acceptable, although not recommended:

```
      integer*4 matrix(3,2)
      integer*2 i, j

      do 10, i=1,3
          do 10, j=1,2
              matrix(i,j)=0
10    continue                   {close i and j loops}
```

In nested **do** loops, you can transfer control from an inner loop to an outer loop, but you are not allowed to transfer control from an outer loop to an inner loop. Furthermore, if two or more nested **do** loops share the same terminal statement, you can only transfer control to that statement from *within the range of the innermost loop*. Any other transfer to that statement would constitute a transfer from an outer loop to an inner loop, since the shared statement is part of the range of the innermost loop.

The following is an example of an improper transfer of control from an outer loop to an inner loop, where both loops share the same terminal statement:

```
* This is an example of improper transfer of control.
         do 100 i=1,10
         if (i.eq.3) then
                   write (*,*) 'branching ', ' i= ',i
                   go to 100
                      {outer loop jumps to shared terminal statement}
         endif        {WRONG!}
         do 100 j=1,5
         write (*,) 'bottom loop',' i= ',i,' j= ',j
100      continue    {shared terminal statement is part of inner loop}
```

Extended Do Ranges--Extension

Domain FORTRAN supports extended **do** ranges, a feature of FORTRAN 66 that the ANSI 1978 standard does not allow. A **do** loop has an extended range if it contains a statement that transfers control outside the loop and if, after executing the outside statement(s), another outside statement returns control to the loop.

As the term suggests, such a construction extends the range of the loop to include those outside statements. The statement that transfers control back to the **do** loop must be within the extended range. In addition, the extended range cannot change the **do** loop's control variable, *var*.

Implied Do Loops

Some FORTRAN statements, such as **data**, simulate **do** loops by allowing you to assign values to entities in a variable list. For example:

```
integer i, j, matrix(10,5)
data    ((matrix(i,j),j=1,2),i=5,10)/12*-1/
```

This **data** statement assigns −1 to each of the 12 elements in **i** and **j**'s specified ranges for array **matrix**.

It is common to use implied **do** loops in **read** and **write** statements. For instance:

```
write (*,*) (temp_data(i), i=1,9)
```

This **write** statement writes the values of **temp_data(1)** through **temp_data(9)** all on one line.

EXAMPLE

```
*  This program uses nested do loops to sort students' test scores
*  in ascending order.  It then uses a separate do loop to add
*  those scores together so that the class average can be computed.

      program do_example
      integer*2  SIZE, i, j, temp, test_scores(10)
      integer*4  total_score
      real*4     average
      parameter  (SIZE=10)

      print *, 'This program sorts students'' test scores.'
      print *, 'Enter 10 integers separated by commas.'

*  This read statement is an implied do loop
      read *, (test_scores(i), i=1,10)

*  Nested do loops to sort scores.  Notice the inner loop ends
*  with a labeled continue statement, while the outer ends with
*  an unlabeled enddo.

      do i = 1, SIZE - 1
          do 10, j = i + 1, SIZE
              if (test_scores(i) .gt. test_scores(j)) then
                  temp = test_scores(i)
                  test_scores(i) = test_scores(j)
                  test_scores(j) = temp
              endif
10        continue
      enddo

      print *, 'The sorted array contains:'
      print *, test_scores

      total_score = 0
      do 20, i = 1, SIZE, 1
          total_score = total_score + test_scores(i)
20    end do

      average = total_score / SIZE
      print 30, average
30    format (' The class average is: ', F5.2)

      end
```

USING THIS EXAMPLE

This program is available online and is named **do_example**. Following is a sample execution of the program:

```
This program sorts students' test scores.
Enter 10 integers separated by commas.
[illegible]
The sorted array contains:
16 32 54 65 77 80 88 92 97 100
The class average is: 70.00
```

do while Executes a block of statements as long as the specified logical expression is true. (Extension)

FORMAT

do [*label*] [,] **while** (*exp*)

ARGUMENTS

label Optional statement label indicating the end of the **do while** range. (The comma after *label* is optional.)

exp A logical expression, enclosed in parentheses, that evaluates to either true or false.

DESCRIPTION

Do while, a Domain FORTRAN extension, begins the execution of a loop that extends from the **do while** to a labeled terminal statement or to an unlabeled **end do** statement. You must use an unlabeled **end do** as the terminal statement if you omit *label* from your **do while** statement.

Do while is similar to the **do** statement, except that instead of looping a set number of times, a **do while** continues looping as long as its logical expression, *exp*, is true.

Domain FORTRAN tests the expression at the beginning of each execution of the loop, including the first. If the expression is true, the compiler executes the statements in the **do while** loop. If the expression is false, control goes to the next statement after the loop. Because FORTRAN tests the expression before the loop is executed, it is possible that the loop will not be executed even once.

EXAMPLE

```
*  This program finds the summation of an integer that a user
*  supplies, and the summation of the squares of that integer.

      program do_while_example

      integer*4  num, sum, square_sum        {Declare and}
      character answer                       {initialize }
      answer = 'y'                           {variables. }

      do while ((answer .eq. 'y') .or. (answer .eq. 'Y'))
          print 5
5         format ('Enter an integer: ', $)
          read *, num
          sum = (num*(num+1))/2
          square_sum = (num*(num+1)*(2*num+1))/6
          print 10, num, sum
10        format ('The summation of ', I2, ' is: ', I4)
          print 20, square_sum
20        format ('The summation of its squares is: ', I5)
          print 30
30        format (/, 'Again? (Y or N) ', $)
          read (*, '(a1)') answer
      end do                     {Close do/while statement.}

      end
```

USING THIS EXAMPLE

This program is available online and is named **do_while_example**. Following is a sample execution of the program:

```
Enter an integer:
The summation of 10 is:   55
The summation of its squares is:   385

Again? (Y or N)
Enter an integer:
The summation of 25 is:  325
The summation of its squares is:  5525

Again? (Y or N)
```

else if . . . then	Refer to **if** in this encyclopedia.

... transfers it to memory. (Extension)

FORMAT

encode(*len, fmt, var*) *iolist*

ARGUMENTS

len — This is an unsigned **integer** constant, **integer** variable, or **integer** expression specifying the record length (in bytes) of the target memory area.

fmt — Format specification for the data can be any one of the following:

- The label of a **format** statement in the same program unit.
- An **integer*4** variable previously assigned a **format** statement label via the **assign** statement.
- A character array or character expression that contains a format string. In a character expression, a format string must be in parentheses and enclosed with single quotes; for example, '(I5)'.

  List-directed formatting is legal in an **encode** statement.

var — A variable, array, or array element specifying the beginning of the target memory area.

iolist — Input/Output List: the entities (for example, variable names) from which data will be transferred.

DESCRIPTION

The **encode** statement, a Domain FORTRAN extension, formats *iolist* entities and stores them in the specified memory area.

Encode and **decode** typically are used to read and write character data in large memory areas that are declared as numeric types. The statements are analogous to **read** and **write** statements to internal files, as specified in the ANSI 1978 FORTRAN standard. (See the descriptions of **read** and **write**.)

The first **encode** argument, *len*, defines the byte length (number of characters) of the target memory area; the third argument, *var*, defines the start of the memory area.

Encode edits the items in the *iolist* into a sequence of characters, arranges them into a *len*–length record, and stores that in memory, starting at the address specified by *var*.

FORTRAN assumes that the target memory area is a *len*-length record. If *len* is less than the record length implied by the *fmt* argument, FORTRAN treats the record as if it were padded with blanks to equal the length *fmt* implies. For example, in this fragment

```
      encode (20,10,line) nums
10    format (5f6.2)
```

the **encode** statement specifies a *len* of 20. But the **format** statement implies a *len* of 30 (because it specifies a 6-digit number repeated five times). FORTRAN simply acts as if the record **nums** has a *len* of 30, and continues.

If you specify the data format within the **encode** statement, instead of listing the label of a statement at which the format is defined, you must enclose the format in parentheses within single quotes; for example:

```
      encode (40,'(5f7.2)',my_data) encode_data
```

Decode, a complementary statement to **encode**, transfers data from memory to **iolist** entities. See the description of **decode** earlier in this section, which also includes an example of **encode**.

end Marks the end of a program unit.

FORMAT

end

ARGUMENTS

None.

DESCRIPTION

End marks the end of a program unit. Program units include main programs, subroutines, functions, and block-data subprograms. The **end** statement must be the last statement of every one of these program units, including all subprograms, and it must be the only statement on the line.

In a main program unit, **end** stops the program without a termination message (unlike **stop**, which is described in this encyclopedia). Control returns to the process or program that invoked the main program.

In a subprogram, **end** returns control to the next executable statement after the subprogram call in the calling program unit.

EXAMPLE

```
* This program computes employee wages and illustrates two uses of the
* end statement.  There is an end for both the main program and the
* subroutine `compute_salary'.
      program end_example

      real*4     hours_worked, hourly_wage, salary  {Declare and}
      character  answer                             {initialize }
      logical    process_more                       {variables. }
      process_more = .true.

      do while (process_more)
          print 10
          read *, hours_worked
          print 15
          read *, hourly_wage
          call compute_salary(hours_worked,hourly_wage,salary)
          print *, 'The salary is ', salary
          print 20
          read (*, '(a1)') answer
          if ((answer .eq. 'n') .or. (answer .eq. 'N')) then
              process_more = .false.
          end if     {Close if statement.}
      end do         {Close do/while statement.}

10    format ('Enter the hours worked: ', $)
15    format ('Enter the employee''s hourly wage: ', $)
20    format (/, 'Again? (Y or N) ', $)

      end            {End of main program unit.}

**********************************************************************
*
*  Simple subprogram for computing an employee's salary.

      subroutine compute_salary(x,y,z)
      real*4 x, y, z

      z = x * y
      end            {End of subroutine - control returns to the print}
*                    {statement just after the subroutine call.      }
```

USING THIS EXAMPLE

This program is available online and is named **end_example**. Following is a sample execution of the program:

```
Enter the hours worked: 40
Enter the employee's hourly wage: 15.35
 The salary is  614.0000

Again? (Y or N) Y
Enter the hours worked: 25.5
Enter the employee's hourly wage: 5.40
 The salary is  137.7000

Again? (Y or N) N
```

end do Terminates the range of a **do** or **do while** statement. (Extension)

FORMAT

end do

ARGUMENTS

None.

DESCRIPTION

End do terminates the range of a **do** or a **do while** statement. You can label an **end do** statement or leave it unlabeled; that is, either of the following is acceptable:

```
* Unlabeled end do                    * Labeled end do
      do i=1,10                             do 15, i=1,10
         .                                     .
         .                                     .
         .                                     .
      end do                          15    enddo
```

If an **end do**'s corresponding **do** or **do while** includes the label of the terminal statement, as in the example on the right above, you must label the **end do** terminal statement. However, if the label of the terminal statement is not included, you can use either a labeled or unlabeled **end do**.

NOTE: Both **end do** and **enddo** are acceptable.

EXAMPLE

See the listings for **do** and **do while** earlier in this encyclopedia for examples of the **end do** statement.

endfile Places an end-of-file marker in a file opened for sequential access.

FORMATS

endfile *unitid* {first form}

endfile ([**unit** = *unitid*] [**,iostat** = *sfield*] [**,err** = *label*]) {second form}

ARGUMENTS

unit = *unitid* ID number of the unit to which the desired file is connected. The phrase **unit** = is optional if this is the first argument and does not occur at all in the first form of **endfile** shown.

iostat = *sfield* Optional I/O Status Specifier. *sfield* must be an **integer*4** variable. FORTRAN returns a 32-bit system or compiler status code to *sfield* if the **endfile** fails; 0 if the **endfile** succeeds.

err = *label* Optional Error Statement Specifier. *label* is the label of a statement in the same program unit as the **endfile**. If the **endfile** fails, FORTRAN transfers control to the statement designated by *label.*

For more information on these arguments, see the "I/O Attributes" listing in this encyclopedia.

DESCRIPTION

Endfile appends an end-of-file marker after the last record read or written in a file opened for sequential access. Note that **endfile** works for sequential files only, not direct access files; that is, it is an error to apply it to a file that has been opened with the specifier **access**='direct'. (Sequential files are the Domain FORTRAN default. However, you can also explicitly declare a file to be sequential by opening it with the specifier **access**='sequential'.)

For example, this **endfile** statement

```
endfile(12, iostat=endfield, err=100)
```

tells FORTRAN to put an end-of-file marker on the file connected to unit 12, return the error status code into **endfield**, and then, if there is an error, jump to the statement labeled 100. Presumably, an error-handling routine begins at statement 100.

Domain FORTRAN automatically puts an end-of-file marker in any file it creates, so it is not necessary to use **endfile** for all sequential access files. In fact, it's a waste of time and code to explicitly append an end-of-file marker to every file. But there are times when **endfile** is useful.

For example, suppose you have a file containing 100 records, the last 25 of which you no longer need. You could explicitly delete each of the 25 records, or you could position the file just before the extra records and issue an **endfile**.

After executing **endfile**, FORTRAN positions the file just *after* the end-of-file marker. Further data transfer at this point produces unspecified results. If you want to **read** from the file or **write** to it, you must explicitly reposition it with a **rewind** or **backspace** statement so that the file is positioned *before* the end-of-file marker.

If you execute an **endfile** for a file that is connected to a unit number but does not yet exist, the **endfile** creates the file.

Endfile doesn't work if the file is opened on more than one unit, or if another user has opened the file.

EXAMPLE

```
*  This program creates and opens a file for sequential access (the
*  default), writes data to the file, appends an end-of-file marker,
*  backspaces before it, and adds more data.

      program endfile_example

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/fio.ins.ftn'
%include '/sys/ins/error.ins.ftn'

      integer*4    openstat, endstat
      integer*2    in_data, new_data, out_data1, out_data2
      data         in_data/1234/
      data         new_data/5678/

      open (unit=2, file='endtest', iostat=openstat, status='new')

      if (openstat .ne. 0) then
      call error_$print(openstat)
      endif

      write (2, '(I4)') in_data     {Write the data to the file.}

      endfile (2, iostat=endstat)   {Write the end-of-file marker.}
```

```
        if (openstat .ne. 0) then
        call error_$print(endstat)
        endif

        print *, 'The file contains the following: ', in_data

*     Use backspace to position the file just before the
*     end-of-file marker, and write more data.
        backspace (2)
        write (2, '(I4)') new_data

*     Rewind to the beginning, read contents of the file into
*     different variables to show the file really contains what
*     it is supposed to contain, and print the results.
        rewind (2)
        read (2, '(I4)/(I4)') out_data1, out_data2
        print *, 'The file now contains: ', out_data1, out_data2
        close(2, status='delete')     {Close and delete 'endtest'.}

        end
```

USING THIS EXAMPLE

This program is available online and is named **endfile_example**. If you execute the program, this is the result:

```
 The file contains the following:  1234
 The file now contains:  1234 5678
```

end if Marks the end of a block **if** statement.

FORMAT

end if

ARGUMENTS

None.

DESCRIPTION

End if defines the end of a block **if**. A block **if** is a series of conditional statements that begins with a block **if** and may include one or more **else if** statements and an **else** statement.

You must terminate each block **if** with an **end if**. This is the only purpose for which you may use **end if**.

> NOTE: Both **end if** and **endif** are acceptable.

For an example of **end if**, see the listing for **if** in this encyclopedia.

entry Defines a secondary entry point in a subprogram.

FORMAT

entry *entry_name* [(*arg1* [, . . . *argN*])]

ARGUMENTS

entry_name The name of the secondary entry point.

arg One or more dummy arguments to be passed to the entry. If you specify multiple arguments, separate them with commas. Enclose the entire list in parentheses.

DESCRIPTION

Entry is a nonexecutable statement that marks an alternate entry point into a subprogram. Typically, you use **entry** if you need to enter a subprogram at some point other than the beginning. If the **entry** is in a subroutine, you can simply call it from another program unit. If it is in a function, reference it just as you would a complete function subprogram.

You can use an **entry** statement anywhere in a subprogram after the initial **subroutine** or **function** statement, except in a **do** loop or an **if** block. If a subprogram has more than one *entry_name*, each one must be unique.

When **entry** appears in a function, *entry_name* has a data type, just as the function name does. If the function name is of a character data type, all *entry_names* within that function must also be of character data type and must have the same length specification as that of the function. If function name is of a noncharacter data type, then *entry_name* can be of any data type *except* the character data type. However, if you invoke the function by its *entry_name* but access the return value that was assigned to the function name, you should be sure that both *entry_name* and the function name are of the same data type (see the example at the end of this listing).

For example, the following is not correct because the function is of type **character*10**, while the *entry_name* defaults to **real** under FORTRAN's naming conventions:

```
character*10 function pass_chars(x,y)
    .
    .
    .
entry here(x)
```

The way to fix this problem is to explicitly declare **here**, as in the following:

```
        character*10 function pass_chars(x,y)
         character*10 here
               .
               .
               .
         entry here(x)
```

You can supply dummy arguments to **entry**, provided the calling program passes corresponding actual arguments. The dummy argument list need not agree with that of the subprogram's **function** or **subroutine** statement, nor with the argument list of any other **entry** statement in the subprogram. If there are discrepancies, however, some arguments may not be accessible through other entry points. For example:

```
        subroutine draw_pic(x_val, y_val, rpt_count)
               .
               .
               .
         entry draw_line(x_val)
```

Notice that calls to **draw_pic** require three actual arguments, while calls to **draw_line** require only one. This means that if the calling unit calls **draw_line**, it won't be able to access the actual arguments passed to **draw_pic**'s arguments **y_val** and **rpt_count**.

EXAMPLE

This first example illustrates the use of **entry** in a subroutine.

```
***********************************************************************
*
*  This program demonstrates the use of the entry statement.  It
*  prompts for user input and calls a subroutine to compute the size
*  of an array or the average of its elements, depending on the entry
*  point.


      program entry_example

      integer*2 array_size, i
      real*4    nums(500), result
      print 5
5     format ('Enter the number of numbers you want to total: ', $)
      read (*, '(I3)') array_size    {User can enter a number up to}
                                     {500, since that's the array's}
                                     {maximum size.                }
      do 10, i=1,array_size          {Load array values.}
         nums(i) = i
```

```
10    continue

*  Enter the subroutine compute at its alternate entry point, sum.
      call sum(nums, array_size, result)
      write (*, 20) array_size, result
20    format ('The sum of the first ', I3, ' numbers is: ', F8.1)
*  Enter the subroutine compute at its beginning.
      call compute(nums, array_size, result)
      write (*, 30) result
30    format ('and the average is: ', F8.1)

      end

**********************************************************************
*  This subroutine totals up the elements of an array, or computes out
*  their average, depending on the point at which it is entered.

      subroutine compute(array, size, result)

      integer*2 size, j
      real*4    array(size), result, total
      logical   switch
      save      total
      switch = .true.
      goto 120
*  Alternate entry point for this subroutine.
      entry sum(array, size, result)
      total = 0.0
      do 100, j=1,size                  {Total up array values.}
          total = total + array(j)
100   continue
      switch = .false.
120   if (switch) then
          result = total/size
      else
          result = total
      endif
      end
```

USING THIS EXAMPLE

This program is available online and is named **entry_example**. Following is a sample execution of the program:

```
Enter the number of numbers you want to total:
The sum of the first  99 numbers is:   4950.0
and the average is:     50.0
```

EXAMPLE

This next example illustrates how to use **entry** in a function.

```
***********************************************************************
*  This program demonstrates the use of the entry statement in a
*  function.  It prompts for user input and calls a function to
*  compute the average of a list of grades in an array.  If the
*  function is by its name, it returns the average score after
*  scaling each grade.  If called by its entry name, it returns
*  the average without scaling.

      PROGRAM FUNC_ENTRY_EXAMPLE

      INTEGER I, GRADE(5), CHOICE
      REAL AVERAGE, SCALE, NO_SCALE, RESULT
      DATA GRADE/93, 85, 50, 75, 85/

      PRINT 15
   15 FORMAT ('Enter 1 if you want to find the scaled average')
      PRINT 20
   20 FORMAT ('or 2 if you want the average without scaling: ', $)
      READ *, CHOICE
      IF (CHOICE .EQ. 1) THEN
*  Enter the function SCALE at its beginning.
           RESULT = SCALE(GRADE, .TRUE.)
      ELSE
*  Enter the function at its alternate entry point, NO_SCALE.
           RESULT = NO_SCALE(GRADE, ARRAY_SIZE, .FALSE.)
      END IF
      PRINT 30, 'The average grade is ', RESULT
   30 FORMAT (A, F4.1)

      END

***********************************************************************
*  This function totals up the elements of an array, or figures out
*  their average, depending on the point at which it is entered.

      REAL FUNCTION SCALE(GRADE)
      INTEGER SIZE, J, SCALE_FACTOR, HIGHEST, GRADE(5)
      PARAMETER (SIZE = 5)
      REAL TOTAL

*  Normal entry point for this function:  Compute the average
*  test score after scaling each grade.

      HIGHEST = 0
      DO 10, J = 1, SIZE               {Find the highest grade}
           IF (GRADE(J) .GT. HIGHEST) THEN
```

Chapter 4. Code

```
                HIGHEST = GRADE(J)
            END IF
   10 CONTINUE

      SCALE_FACTOR = 100 - HIGHEST      {Compute scale factor}
      DO 20, J = 1, SIZE                {Add it in to each grade}
          GRADE(J) = GRADE(J) + SCALE_FACTOR
   20 CONTINUE

*  Alternate entry point:  Compute the average test score without
*  scaling the grades.

      ENTRY NO_SCALE(GRADE)
      TOTAL = 0
      DO 30, J = 1, SIZE               {Total up grades}
          TOTAL = TOTAL + GRADE(J)
   30 CONTINUE

      SCALE = TOTAL/SIZE               {Compute average}

      RETURN
      END
```

USING THIS EXAMPLE

This program is available online and is named **func_entry_example**. Following is a sample execution of the program:

```
Enter 1 if you want to find the scaled average
or 2 if you want the average without scaling:
The average grade is 84.6
```

equivalence Associates two or more variables, arrays, array elements, or character substrings with the same storage area.

FORMAT

equivalence *(nlist1)* [. . . ,*(nlistN)*]

ARGUMENT

nlist The variables, arrays, array elements, or character substrings that you want to share a storage area. The entities on the list constitute an **equivalence** class. Separate each entity with a comma, and enclose the entire list in parentheses. If you declare multiple *nlists*, separate each parenthesized list with a comma.

DESCRIPTION

The **equivalence** statement associates two or more variables, arrays, array elements, or character substrings with the same storage area. You can use **equivalence** to assign the same storage space to variables of any type. For example:

```
integer*4 whole_num, x, y
real*4    decimal_num, z
equivalence (whole_num, decimal_num), (x, y, z)
```

This fragment associates variables **whole_num** and **decimal_num** (an equivalence class) with one storage area, and **x**, **y**, and **z** (another equivalence class) with another storage area.

As an extension to the ANSI standard, Domain FORTRAN allows you to use the equivalence statement to associate character variables with noncharacter variables so that they both share the same storage area.

Equivalence establishes a one-to-one storage relationship between variables. Remember that in FORTRAN, storage is allocated to various data types as shown in Table 4-9.

Table 4-9. Amount of Storage by Data Type

Data Type	Number of Bytes per Variable
byte	1
character*n	n
complex, complex*8	8
double complex, complex*16	16
☆**integer**	4
integer*2	2
integer*4	4
☆☆**logical**	4
logical*1	1
logical*2	2
logical*4	4
real, real*4	4
real*8, double precision	8

☆ If you compile with **-i*2**, **integer** variables are allocated 2 bytes (refer to subsection 6.5.14 for details).

☆☆ If you compile with **-l*1**, **logical** variables are allocated 1 byte, and if you compile with **-l*2**, then **logical** variables are allocated 2 bytes (refer to subsection 6.5.21 for details).

This means, for instance, that if you **equivalence** two **real*4** variables and one **complex** variable, they share exactly the same storage area because they both take a total of eight bytes.

You might want to define such an **equivalence** if you have a large array of **complex** numbers and you need at times to separate the numbers into their two **real*4** component parts. Rather than taking the time to explicitly break apart each number, the **equivalence** statement implicitly makes the break, saving significant compute time.

When you associate two **real*4** variables and one **complex** variable by **equivalence**, as in the example above, the sizes match exactly. However, there is no requirement that the sizes be identical before you can specify an **equivalence**. For example, the following is legal:

```
integer*2        small_num
double precision great_big_num
equivalence (small_num, great_big_num)
```

In this case, the two variables begin at the same byte. **great_big_num** just happens to run on for six more bytes after **small_num**.

You should be careful, however, if you **equivalence** variables with different sizes, or your program may not produce the results you intend. For example, if you **equivalence** a **double precision** array to one of type **real*4**, be aware that each **double precision** element occupies twice as many bytes as each **real*4** element. References in your code to array elements must take the difference into account.

Remember, too, that there are differences in the ways the various data types are stored internally. Chapter 3 describes the data types' internal representation.

> **NOTE:** When you **equivalence** a local variable with another variable, you disable optimization on both variables. This may make your program run more slowly.

Equivalence with Arrays

You can use **equivalence** to associate two or more arrays (or elements within them) with the same storage area, or to associate variables and array elements. This can be useful if you have data that you need to use in two different ways. For example, suppose you have data about dates that you need to manipulate arithmetically at some times and want to print out as characters at others. Instead of storing the same data in two different storage locations (each with different data type), you can use **equivalence** statement to associate two different variable names with the same storage location, as follows:

```
      integer*4   month, day, year
      character*4 date_data(3)
      equivalence (date_data(1), month), (date_data(2), day),
     +            (date_data(3), year)  { the + in column 6 is the  }
                                        { continuation character    }
```

In FORTRAN, the leftmost subscript of arrays changes most rapidly. This means that this array

```
      real*8 A(2,3)
```

is stored as follows:

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

Thus, given the statements

```
      real*8 A(2,3), B(3,2)
      equivalence (A,B)
```

the elements in arrays A and B share storage as follows:

A(1,1)	A(2,1)	A(1,2)	A(2,2)	A(1,3)	A(2,3)
↕	↕	↕	↕	↕	↕
B(1,1)	B(2,1)	B(3,1)	B(1,2)	B(2,2)	B(3,2)

When you **equivalence** one element of an array to one element of another, you implicitly establish a one-to-one correspondence between the other elements in the arrays. For example

```
integer*2    this(6), that(4)
equivalence (this(3), that(2))
```

not only associates **this(3)** and **that(2)** with the same storage unit, but also makes these associations:

```
this(2) and that(1)
this(4) and that(3)
this(5) and that(4)
```

Careful use of **equivalence** enables you to refer to an element in a multidimensional array with just one subscript. Assume, for example, that you want to use only one subscript to refer to each element in a 3x3 array. To do this, define another array of nine elements, and declare the two arrays to be equivalent:

```
integer multi(3,3), single(9)
equivalence (multi, single)
```

Given these statements, the references **multi(3,2)** and **single(6)**, for example, refer to the same array element.

Equivalence and Common

A variable or array name can appear in both a **common** and an **equivalence** statement in the same program unit. However, there are some restrictions, as the following list notes:

- You can use an **equivalence** statement to implicitly extend an unnamed **common** area, but *only* if you're extending the area beyond its last element. For example

  ```
  real*8 e, f, g, h(3)
  common e, f, g
  equivalence (h(2), g)
  ```

  works because, even though it adds a storage unit to the unnamed **common** (to accommodate **h(3)**), the unit is added to the end of the **common** area. However, the following **equivalence** statement is illegal, since it attempts to extend the **common** area before its first element.

```
real*8 e, f, g, h(3)
common e, f, g
equivalence (h(3), e)            {Illegal!}
```

That is, since the **equivalence** assigns **h(3)** and **e** to the same storage area, and **e** is the beginning of the **common** area, **h(1)** and **h(2)** would have to be inserted before **e**. However, you can't extend the **common** area before its first element, so this assignment is illegal.

- The areas must be the same length in all program units. Thus, if you use **equivalence** to implicitly extend a named **common** area in one program unit, you should adjust that named **common** accordingly in every other program unit in which it appears.

- Within any given program unit, no two variables or arrays may be declared in **common** and also defined as equivalent. For example

```
character*15 last_name, manager_name, emp_name
common       last_name, manager_name, emp_name
equivalence (last_name, emp_name)              {Illegal!}
```

  won't work because it gives variable **emp_name** two contradictory storage assignments.

- An **equivalence** statement must not specify that consecutive storage units are nonconsecutive. So this is illegal:

```
real*4            temps(2)
double precision  d(2)
equivalence (temps(1), d(1)), (temps(2), d(2)) {Illegal!}
```

- You cannot **equivalence** variables or arrays in different **common** blocks. This is illegal because the **equivalence** statement declares that the variables are stored in the same place, but the separate **common** blocks specify that they are stored in different areas.

NOTE: If you **equivalence** a variable in **common** with another variable, the compiler cannot optimize references to the variable in **common** when it is used in loops. As a result, your program may run more slowly.

external Allows an external function or subroutine to be used as an actual argument.

FORMAT

external *name1* [. . . , *nameN*]

ARGUMENT

name The name of a subroutine or function that is external to the calling program unit.

DESCRIPTION

External declares one or more functions or subroutines to be outside the program unit in which the **external** declaration appears. In essence, **external** tells the compiler to look elsewhere for the specified entity, because it is not a variable in this program unit.

You must declare as **external** any subprogram—excluding intrinsic functions—if you intend to use it as an actual argument in a function reference or **call** statement. You must also use this statement when referencing an external function or subroutine that has the same name as a FORTRAN intrinsic function (to distinguish it from the intrinsic function). You would do this if you decided to write your own version of one of the intrinsic functions.

If you do write a routine with the same name as one of the intrinsic functions, and so declare your routine to be **external**, you cannot use the Domain FORTRAN version of that routine in the program unit in which the **external** declaration appears.

Also, you cannot declare a **statement** function (a function that consists of a single statement) to be **external**.

EXAMPLE

```
*  This program computes the roots of x in an equation where
*  ax2 + bx + c = 0 when a <> 0 and the roots are real.  If the
*  roots are complex, then it reports that information.  The program
*  uses the external function my_sqr and it also uses the intrinsic
*  function sqrt.

      program external_example
      external my_sqr                    {Make external declaration.}
      real a, b, c, result1, result2
      print *, 'Enter values for a, b, and c'
      print *, 'Leave spaces between the three values'
      read (*,*) a, b, c

*  The parameter list includes the external function my_sqr.

      call quadratic(my_sqr, a, b, c, result1, result2)
      end
*******************************************************************
*  Subroutine to solve the quadratic equation.

      subroutine quadratic(sqr1, a1, b1, c1, answer1, answer2)
      real sqr1, a1, b1, c1, answer1, answer2, hold
      hold = sqr1(b1) - (4*a1*c1)    {Call to external function.}

      if (hold .lt. 0) then
           print *, 'There are two complex roots.'
      else
           answer1 = (-b1 + (sqrt(hold))) / (2*a1) {Call to intrinsic}
           answer2 = (-b1 - (sqrt(hold))) / (2*a1) {function, sqrt.}
           write (*,50) answer1, answer2
50         format ('The real roots are: ', f7.3, ' and ', f7.3)
      endif
      return
      end

********************************************************************
*  External function my_sqr.
      real function my_sqr(b2)
      real b2
      my_sqr = b2*b2
      return
      end
```

USING THIS EXAMPLE

This program is available online and is named **external_example**. Following is a sample execution of the program:

```
Enter a, b, and c

The answers are:  -0.500 and  -1.000
```

format Specifies the format of data to be read, written, or printed.

FORMAT

label **format** (*specs*)

ARGUMENTS

label	The statement label of this **format** statement. Like that of any other statement, *label* must appear somewhere in columns 1 through 5.
specs	Format specifications: one or more repeatable or nonrepeatable edit descriptors indicating the format of the data.

DESCRIPTION

Formatting is essential to FORTRAN I/O; without a format directive of some kind, the compiler cannot correctly read or write formatted (nonbinary) data. There are several ways to specify a format in a FORTRAN **read, write,** or **print** statement:

- With a **format** statement.
- With a character expression (a character constant, a character variable or a character array). The expression must be enclosed in parentheses within single quotes.
- With an **integer*4** variable you've previously assigned to a **format** statement label via the **assign** statement.
- With an asterisk (*) for **list-directed** formatting, which directs the compiler to format a variable according to its data type.

This section focuses on the **format** statement which, with character expression formatting, is called **edit-directed formatting**. At the conclusion, there are brief descriptions of the other formatting methods.

The Format Statement

The **format** statement is one of the most common methods of specifying an I/O format in FORTRAN; you write a **format** statement that includes one or more edit descriptors for the data, label the **format** statement, and then refer to the label in an I/O statement (**read, write,** or **print**).

REPEATABLE EDIT DESCRIPTORS

If the I/O statement has an *iolist*—that is, a list of specific variables to be read or written—the corresponding **format** statement must contain at least one **repeatable edit descriptor**. In the following simple **write** and **format** statements, **x**, **y**, and **z** make up the *iolist* and F is the repeatable edit descriptor.

```
      real*4 x, y, z
          .
          .
          .
      write(4,100) x, y, z
 100  format(3F6.2)
```

As the name implies and as the example shows, repeatable edit descriptors may be preceded by a **repeat count**. The repeat count (3) tells the compiler to apply the edit descriptor to more than one variable—in this case, to apply the description '**F6.2**' to **x** and y and **z**.

In general, the repeatable edit descriptors are for use with logical, character, or numeric data. Except for the **A**, **Z**, and **O** descriptors, each edit descriptor must match its corresponding variable in type. Table 4–10 lists the repeatable edit descriptors.

Table 4–10. FORTRAN Repeatable Edit Descriptors

Edit Descriptor	**What It Does**
I	Edits numeric data into integer format.
F	Edits numeric data into real, double-precision, or complex format.
E, D, or G	Edits numeric data into real, double-precision, or complex format with an explicit exponent.
L	Edits logical (**.true.** or **.false**) data.
A	Edits data into character format.
Z	Edits data into hexadecimal format.
O	Edits data into octal format.

The following subsections describe each of the repeatable edit descriptors in detail.

NOTE: While keywords usually appear in lowercase **bold** letters in text, we are using uppercase **bold** for the edit descriptors. That's to help make them more readily visible, and to differentiate them from variable names. In examples, we also use uppercase letters for these descriptors. That means that even though it's correct to do this:

```
10    format(a1, 3i3, f5.2)
```

it appears like this:

```
10    format(A1, 3I3, F5.2)
```

NONREPEATABLE EDIT DESCRIPTORS

In addition to the repeatable edit descriptors, there are **nonrepeatable edit descriptors**. These descriptors control other formatting issues, such as sign output, blank interpretation, record position, and scale (movement of decimal points). Table 4-11 lists the nonrepeatable edit descriptors.

Table 4-11. FORTRAN Nonrepeatable Edit Descriptors

Edit Descriptor	Function
S or **SS**	Sign control; restores normal sign handling for the rest of this format specification. For output only; ignored on input.
SP	Sign control: forces plus sign (+) output for all positive **I**, **F**, **D**, **E**, and **G** values within the rest of this specification. For output only; ignored on input.
BZ	Blank handling; treats blanks as zeros. For input only; ignored on output.
BN	Blank handling; treats blanks as nulls (ignores blanks). For input only; ignored on output.
*k***P**	Scale factor used with **F**, **E**, **D**, or **G** edit descriptors. Moves the decimal point *k* places to the right or left.
*n***H**	Indicates a Hollerith constant (same as single quote).
*n***X**	Skips *n* character positions on input or output. *n* is required, even it if is 1. By specifying "1X" at the beginning of each record, you can force blank (single space) carriage control on output, provided the file has the FORTRAN carriage control attribute (STREAM_$F77_CC).
T*n*	Sets absolute tab position to *n*.
TL*n*	Tabs left *n* characters or digits in the current record (to reread or rewrite data). You cannot use **TL** to position outside the current record.
TR*n*	Tabs right "n" characters or digits in the current record (to skip data).
/ (slash)	On input, skips the rest of the current record and reads the next record. Two or more slashes skip two or more records. On output, one slash terminates the current record and writes the next record (line). Two slashes output a blank line.
' ' (single quotes)	Writes a character or character string literally. For example, `write (*,*) 'Hello'` writes the character string **Hello** to standard output.
: (colon)	Ends formatting if there are no more items in the *iolist*; ignored if there are more items in the *iolist*.
$ (dollar sign)	By default, a **write** statement always causes a carriage return and linefeed; however, if you use the $ option, the **write** suppresses the terminating carriage return and linefeed. Note that this option only has an effect at the end of a **format** statement.

GENERAL FORMAT RULES

On input, FORTRAN ignores leading blanks in numeric values. It also ignores embedded or trailing blanks unless you use the **BZ** edit descriptor, which directs the compiler to interpret embedded blanks as zeros for the rest of the format specification. The **BN** descriptor restores normal blank handling in a format specification.

By default, FORTRAN does not output plus signs (+) before positive values. The **SP** edit descriptor forces plus sign output for positive values; **S** or **SS** restores normal sign handling. Like **BZ** and **BN**, once **S**, **SS**, or **SP** are set, they affect the rest of the format specification.

Format Reversion

FORTRAN always attempts to format all entities in an *iolist*. If it encounters a slash (/) in the format specification, FORTRAN reads the next record, using the repeatable edit descriptors to the right of the slash for the remaining *iolist* entities.

If the repeatable edit descriptors in any format specification have been used up and there are more *iolist* entities, FORTRAN performs *format reversion*, as follows:

- If there are nested parentheses in the format specification, FORTRAN reverts to the format terminated by the last preceding right parenthesis.
- If there are no nested parentheses, FORTRAN reverts to the beginning of the format specification.

For example, in the following simple program, the **print** statement lists two *iolist* entities: **i** and **matrix(i)**. The **format** statement has a description for both, so no format reversion is required.

```
      program reversion_example
      integer*2 i, matrix(6)

      do 10, i=1,6
          matrix(i)=i*i
          print 20, i, matrix(i)
10    continue

20    format('Index equals ', I2, 2X, 'Value equals ', I2)
      end
```

This is the way the output looks:

```
Index equals  1  Value equals  1
Index equals  2  Value equals  4
Index equals  3  Value equals  9
Index equals  4  Value equals 16
Index equals  5  Value equals 25
Index equals  6  Value equals 36
```

But the output changes significantly if the **format** statement is altered so that it reads:

```
20    format('Num equals ', I2)
```

Now there's only a description for the first *iolist* entity—in this case, the variable **i**. So FORTRAN formats **i**, performs format reversion, and formats **matrix(i)** using the description it just used for **i**. This time the result looks like this:

```
Num equals  1
Num equals  1
Num equals  2
Num equals  4
Num equals  3
Num equals  9
Num equals  4
Num equals 16
Num equals  5
Num equals 25
Num equals  6
Num equals 36
```

When FORTRAN reverts to a format that includes a repeat count, it uses the repeat count. The reused portion of the format specification must contain at least one repeatable edit descriptor.

I (INTEGER) EDITING

The repeatable edit descriptor I edits data into integer format, and has the following syntax:

[*count*] I [*width.min*]

count — Optional repeat count: an unsigned positive integer. Directs the compiler to reuse this edit descriptor *count* times.

width — Required field width: an unsigned positive integer indicating the size of the integer variable to be read or written. Signs and blanks count as characters. On input, blanks have no significance unless you specify **BZ**, which causes FORTRAN to treat embedded blanks as zeros. On output, blanks have no value significance. On output, if the actual number of characters exceeds *width*, FORTRAN writes a field of asterisks. If the number of actual characters is less than *width*, FORTRAN pads with leading blanks to equal *width*.

.min An optional positive integer indicating the minimum number of characters to be output. If necessary, FORTRAN pads to *min* with leading zeros.

Example of I (Integer) Editing

```
      program i_fmt_example

      integer*2 first, second, third
      integer*4 sum, with_blanks

*  Note that the first format statement includes a repeat count of 3.

      print *, 'Enter three three-digit integers separated by spaces:
     '
      read (*, 10) first, second, third
10    format (3I4)

      sum = first + second + third
      write (*, 20) sum
20    format ('They add up to: ', I4)

*  This section shows what happens when you use the BZ edit
*  descriptor. The user enters a number with embedded blanks, and
*  because the format statement at line 30 includes BZ, those
*  embedded blanks are treated as zeros.  The output shows this.
      print 25
25    format (/, 'Now, enter a 7-digit integer with embedded blanks:')
      read (*, 30) with_blanks
30    format (BZ,I7)
      write (*, 40) with_blanks
40    format (I7)
      end
```

Using This Example

This program is available online and is named **i_fmt_example**. Following is a sample run of the program:

```
Enter three three-digit integers separated by spaces:
[illegible]
They add up to: 1400

Now, enter a 7-digit integer with embedded blanks:
[illegible]
3006502
```

F, E, D, OR G (REAL, DOUBLE-PRECISION OR COMPLEX) EDITING

The repeatable edit descriptor **F** reads or writes data in real or double-precision format. To format complex data, you can use two **F** descriptors, one for the real part and the other for the imaginary part. (**E**, **D**, or **G** descriptors are more often used for this purpose.)

When your program accepts input for a real or double-precision number, the value does not have to contain an actual decimal point. If it does, the compiler honors the actual decimal position, rather than the decimal position that the edit descriptor specifies. A floating-point variable edited with **F** may or may not include an explicit exponent (preceded by the uppercase letter E) on input; the compiler always writes it without an explicit exponent.

The repeatable edit descriptors **E**, **D**, and **G** read or write data in real or double-precision format. **E** and **D**, which are functionally equivalent, read data with or without an explicit exponent, but write data with an explicit exponent.

On input, **G** editing is identical to **F**, **E**, and **D**. On output, **G** editing is the same as either **F** or **E**, depending on the magnitude of the data. If the exponent is between 0 and the number of digits to the right of the decimal point (**dec**), **F** mode is used. In that case, FORTRAN decreases the field width by four (width−4); that is, it writes the digit in the first width−4 columns, and follows it with four blanks. If the exponent is larger than the value of **dec**, **E** mode is used.

The **F**, **E**, **D**, and **G** edit descriptors have the following syntax:

[*k***P**] [*count*]**F***width.dec*

[*k***P**] [*count*]**E***width.dec*[**E***exp*]

[*k***P**] [*count*]**D***width.dec*

[*k***P**] [*count*]**G***width.dec*[**E***exp*]

*k***P** Optional scale factor: a signed or unsigned integer which moves the decimal point *k* places to the right or left.

A scale factor can appear anywhere in the list of edit descriptors but must precede the edit descriptors (**F**, **E**, **D**, and **G**) to which you want it to apply. If you do not specify a scale factor, it defaults to 0 at the beginning of each input/output statement and applies to all subsequently interpreted edit descriptors until a specified scale factor is encountered, and then that scale factor is established.

On both input and output, the value's external representation equals its internal representation multiplied by 10***k*. However, the scale factor has no effect if the value has an exponent.

On output with **E** or **D** editing, FORTRAN multiplies the value by "10***k*", then subtracts *k* from the exponent, which changes the value's representation (not the value itself). *k* must be greater than *-dec* (the decimal field width), and less than *dec*+1. If *k* is greater than 0 and less than *dec*+2, FORTRAN outputs *k* digits to the left of the decimal point, and *dec*-*k*+1 digits to the right of the decimal point. If *k* is less than 0 and greater than *-dec*, FORTRAN outputs **abs**(*k*) — the absolute value of *k* — with leading zeros to the left of the decimal point and *dec*-**abs**(*k*) digits to the right of the decimal point.

On output with **G** editing, FORTRAN uses either **F** or **E** editing, depending on the absolute value of the output, which it evaluates before checking the scale factor. If the absolute value of the output falls between .1 and 10***dec*, inclusive, FORTRAN ignores the scale factor and outputs the value in **F** format. If the value's exponent is less than 0 or greater than *dec*, FORTRAN outputs the value in **E** format, using the scale factor to determine the number of leading zeros after the decimal point.

count Optional repeat count: an unsigned positive integer. Directs the compiler to reuse this edit descriptor *count* times.

width — Required total field width: an unsigned positive integer indicating the total size of the value to be read or written, including the actual decimal point (if there is one) and all digits to the right of the decimal point. On output, if the actual number of characters exceeds *width*, FORTRAN writes a field of asterisks. If the number of actual characters is less than *width*, FORTRAN pads with leading blanks to equal *width*. Signs and blanks are handled just as they are in **I** (integer) editing.

.dec — Required decimal field width: an unsigned positive integer indicating the number of digits to the right of the decimal point. On output, if the number of digits after the decimal point in the variable exceeds *dec*, FORTRAN rounds the fraction to the *dec* place. For example, if the value 3.14159 were to be written using the format specification F5.3, the value would be rounded to 3.142.

E*exp* — Optional explicit exponent: an unsigned integer indicating the number of digits of the exponent. Output only; ignored on input. If you don't specify *exp*, the exponent value cannot exceed 999. An exponent larger than 99 occupies 4 character positions, including sign and three digits. If *exp* is specified, the exponent occupies *exp*+2 character positions, including the **E**, a sign, and *exp* exponent digits.

Example of F Edit Descriptor

```
      program f_fmt_example
      real*4 a, b, c, d, sign

*  The format statement requires that input values look like "nn.nnn"
*  with a space after each value.  However, you may substitute blanks
*  for any leading zeros.

      print *, 'Enter four real numbers separated by spaces'
      read (*,10) a, b, c, d
10    format (4(F6.3, 1X))

*  Write with a scale factor of 2, so the decimal point is moved
*  two places to the right.  This corresponds to multiplying by 100.

      write (*,15) a, b, c, d
15    format (4(2PF8.3, 1X))

*  The format statement on line 20 includes examples of the slash,
*  apostrophe, and dollar sign edit descriptors.

      print 20
20    format (/, 'Now, enter another real number: ', $)
      read (*,25) sign
25    format (F8.2)
```

```
*  This format statement demonstrates the SP sign control descriptor.
*  The output field has been expanded one place to accommodate the
*  sign that SP produces.

      write (*, 30) sign
30    format (SP, F9.2)

      end
```

Using This Example

This program is available online and is named **f_fmt_example**. Following is a sample run of the program. Note that although you can substitute blanks for the leading zeros on the first input line, we show those zeros for clarity.

```
Enter four real numbers separated by spaces
03.333 44.444 06.876 12.305
 333.300 4444.400  687.600 1230.500

Now, enter another real number: 20306.12
+20306.12
```

Example of E and G Edit Descriptors

```
      program eg_fmt_example
      real*8 x, y
      data x,y/-7655, -7.655E+04/

      print *, 'The values of x and y with E editing are:'
      print 10, x, y
10    format ('x = ', E11.4/, 'y = ', E11.4)

*  The G edit descriptor says to use either F or E editing, depending
*  on the absolute value of the output.  In this case, x is output
*  with F editing and y with E editing.  FORTRAN uses E formatting
*  for y because its exponent is greater than the number of
*  digits after the decimal point (that is, 3).

      print *, 'The values of x and y with G editing are:'
      print 20, x, y
20    format ('x = ', G11.4/, 'y = ', G11.4)


*  This shows how to use the scale factor to print x in scientific
*  notation.  In that notation, the mantissa is in the range 1.0
```

```
*    through 9.999, rather than 0.1 through 0.999.

      print *, 'The value of x written in scientific notation is:'
      print 30, x
30    format ('x = ', 1PE11.4)

      end
```

Using This Example

This program is available online and is named **eg_fmt_example**. If you execute the program, you get this output.

```
 The values of x and y with E editing are:
x = -0.7655E+04
y = -0.7655E+05
 The values of x and y with G editing are:
x =  -7655.
y = -0.7655E+05
 The value of x written in scientific notation is:
x = -7.6550E+03
```

L (LOGICAL) EDITING

The repeatable edit descriptor **L** reads or writes logical values (**.true.** and **.false.**). The variable must be of type **logical**, and its first nonblank characters must be

for value of **.true.**	for value of **.false.**
T	F
.T	.F
t	f
.t	.f

FORTRAN ignores all subsequent characters. This means, for example, that if you enter the value **fine** for a **logical** variable, it is assigned **.false.** because **fine** begins with an 'f'. Likewise, the value **tiny** results in an assignment of **.true.**

The **L** edit descriptor has the following syntax:

[*count*] **L***width*

count — Optional repeat count: an unsigned positive integer. Directs the compiler to reuse this edit descriptor *count* times.

width — Required total field width: an unsigned positive integer indicating the total number of characters of the value to be read or written. On output, if the value of the variable is **.true.**, FORTRAN writes *width*–1 blanks, followed by **T**. If the variable's value is **.false.**, FORTRAN writes *width*–1 blanks, followed by **F**.

Example of L Edit Descriptor


```
      program l_fmt_example
      logical did, did_not
      print 10
10    format ('Enter the values for the logical variables:')
      read (*,20) did, did_not
20    format (L4,1X,L3)
      print 25
25    format ('Now they look like this: ')
      print 30, did, did_not
30    format (L4,L3)
      print 40
40    format ('1234123          - ruler to help you see spacing')
      end
```

Using This Example

This program is available online and is named **l_fmt_example**. Following is a sample run of the program.

```
Enter the values for the logical variables:
true foo
Now they look like this:
   T  F
1234123          - ruler to help you see spacing
```

A, ' ', AND H (CHARACTER) EDITING

You generally use the A edit descriptor to read or write character variables, although in Domain FORTRAN you can edit variables of any type with A. For example, the statements

```
        integer*4     int_num
        write (*, 20) int_num
   20   format (A3)
```

write the leftmost three bytes (characters) of the **integer*4** variable **int_num** to standard output.

Domain FORTRAN also supports editing with single quotation marks (using the A—for apostrophe—edit descriptor) and H (Hollerith) editing for character strings. However, you can use single quotes and **H** editing for character string output only.

The A edit descriptor has the following syntax:

[*count*]A[*width*]

[illegible]

count — Optional repeat count: an unsigned positive integer. Directs the compiler to reuse this edit descriptor *count* times.

width — Optional total field width: an unsigned positive integer indicating the total number of characters of the value to be read or written. If you omit *width* on input, FORTRAN reads the number of characters previously declared for the variable. For example, given

```
character*10 first_name
read (*, '(A)') first_name
```

FORTRAN reads all 10 characters in **first_name**, even though the format specification A has no field width.

If *width* is less than the number of characters that can be assigned to the variable, *width* characters, left-justified, are assigned to it, and trailing blanks are added to fill the variable. If *width* is greater than the number of characters, FORTRAN reads the rightmost *width* characters. The leftmost excess characters are ignored.

If you omit *width* on output, FORTRAN writes all the characters in the variable. If *width* exceeds the number of characters, FORTRAN right-justifies the value and pads the left with leading blanks to equal the value of *width*. If *width* is less than the number of characters, FORTRAN writes the leftmost *width* characters.

The **H** edit descriptor has the following syntax:

[illegible] — [illegible] the number of characters, including blanks, in [illegible]

[illegible] the string of characters you want printed out.

Example of A and H Edit Descriptors

```
      program ah_fmt_example

      character*26 line
      character*10 first_name
      character*1  middle_initial
      character*15 last_name, word

      open(4, file='names_data', recl=26, status='readonly')

*  This section reads names from a file and uses the A edit descriptor
*  to format them for output.

      print *, 'Here are the names'       {Single quote editing used  }
5     read(4, '(A26)', end=100) line      {in these lines.           }
      first_name(1:10) = line(1:10)
      middle_initial = line(11:11)
      last_name(1:15) = line(12:26)
      write (*,10) first_name, middle_initial, last_name
10    format (A,1X,A,1X,A)                {Edit descriptor A used    }
      go to 5                             {without width specifiers.}

100   close(4)
*  This section allows for a word of up to 15 characters to be
*  entered, but when the word is written out, the width specified
*  is only 10 characters.  It shows what happens when the width
*  specification is less than the length of the variable's value.
      write (*,110)
110   format (16HEnter any word: ,$)     {Notice Hollerith editing.}
      read (*,120) word
120   format (A15)
      write (*,130) word
130   format (A10)
      end
```

Using This Example

This program is available online and is named **ah_fmt_example**. Following is a sample run of the program:

```
 Here are the names
Stewart    M Franklin
Kayla      J Brady
Pierre     Y Giroux
Maddie     A Hayes
Sterling   R Gillette
```

```
Ilsa       L Lazlo
Enter any word:
Encycloped
```

Z (HEXADECIMAL) EDITING

You can use the repeatable edit descriptor **Z** to edit data into hexadecimal format. It can be used with data of any type in Domain FORTRAN. **Z** has the following syntax:

[*count*] **Z***width* [*.min*]

count Optional repeat count: an unsigned positive integer. Directs the compiler to reuse this edit descriptor *count* times.

width Required total field width: an unsigned positive integer indicating the total number of characters of the value to be read or written. The field and the corresponding data type are right justified on both input and output.

.min An optional positive integer indicating the minimum number of characters to be output. If necessary, FORTRAN pads to *min* with leading zeros.

Example of Z Edit Descriptor

```
*  This program takes a base-10 number that a user enters, and prints
*  it out in hexadecimal format.

      program z_fmt_example

      integer*4 your_num
      character answer
      logical   again
      again = .true.

      do while (again)
          print 10
10        format ('Enter a base 10 (decimal) integer: ',$)
          read (*, '(I4)') your_num
          print 20, your_num
20        format ('The number in base 16 (hexadecimal) is: ', Z4)
          print 30
30        format (/, 'Again? (Y or N) ', $)
          read (*, '(A1)') answer
          if ((answer .eq. 'n') .or. (answer .eq. 'N')) again =.false.
      enddo

      end
```

Using This Example

This program is available online and is named **z_fmt_example.** Following is a sample run of the program:

```
Enter a base 10 (decimal) integer: 13
The number in base 16 (hexadecimal) is:    D

Again? (Y or N) y
Enter a base 10 (decimal) integer: 500
The number in base 16 (hexadecimal) is:  1F4

Again? (Y or N) n
```

O (OCTAL) EDITING

You can use the repeatable edit descriptor **O** to edit data into octal format. It can be used with data of any type in Domain FORTRAN. **O** has the following syntax:

[*count*] O*width* [*.min*]

count	Optional repeat count: an unsigned positive integer. Directs the compiler to reuse this edit descriptor *count* times.
width	Required total field width: an unsigned positive integer indicating the total number of characters of the value to be read or written. The field and the corresponding data type are right justified on both input and output.
.min	An optional positive integer indicating the minimum number of characters to be output. If necessary, FORTRAN pads to *min* with leading zeros.

Example of O Edit Descriptor

```
*  This program takes a base-10 number that a user enters, and prints
*  it out in octal format.

      program o_fmt_example

      integer*4 your_num
      character answer
      logical   again

      again = .true.
      do while (again)
          print 10
10        format ('Enter a base 10 (decimal) integer: ',$)
          read (*, '(I4)') your_num
          print 20, your_num
20        format ('The number in base 8 (octal) is: ', O4)
          print 30
30        format (/, 'Again? (Y or N) ', $)
          read (*, '(A1)') answer
          if ((answer .eq. 'n') .or. (answer .eq. 'N')) again =
.false.
      enddo

      end
```

Using This Example

This program is available online and is named **o_fmt_example.** Following is a sample run of the program:

```
Enter a base 10 (decimal) integer: 27
The number in base 8 (octal) is:   33

Again? (Y or N) y
Enter a base 10 (decimal) integer: 255
The number in base 8 (octal) is:   377

Again? (Y or N) n
```

FORMATTING WITH CHARACTER EXPRESSIONS

Although it is always permissible to specify the label of a **format** statement in a **read** or **write** statement, this is not a requirement. You can use a character constant or string within the **read** or **write** statement itself to specify the format. If you do so however, you must enclose the format in parentheses within single quotes. For example:

```
integer age, year_born
   . . .
read (5, '(2I4)') age, year_born
```

You also can define a format specification as an expression, variable, or array of type **character**, and then use that directly in an I/O statement. The example following shows this type of format specification.

Example of Formatting Characters

```
*  This program uses formatting with a character variable and with a
*  character array.

      program char_fmt_example

      character*7 form       {Declare character variable.}
      character*1 other_form(7)     {Declare character array.    }

      real*4  first, second, third
      data    first, second, third /2.34, 76.18, 95.93/

* Assign values, which together make a format, to the array other_form
      data other_form /'(', '3', 'F', '7', '.', '3', ')'/

      form = '(3F6.2)'       {Assign format specification to variable}

*  Use character variable and array in format specifications.
      write (*, form) first, second, third
      write (*, other_form) first, second, third

      end
```

Using This Example

This program is available online and is named **char_fmt_example**. If you execute the program, you get the following output.

```
2.34 76.18 95.93
2.340 76.180 95.930
```

FORMATTING WITH ASSIGNED VARIABLES

Another way to specify a format is to use the **assign** statement to associate an **integer*4** variable with the statement label of a **format** statement. You can then use the variable name in place of the **format** label. Note, however, that using an **assign** statement with an **integer*4** variable disables optimization of the variable.

Example of Formatting with Assigned Variables

See the listing for **assign** in this encyclopedia for more information and an example of this format specification method.

LIST-DIRECTED FORMATTING

List-directed formatting signals the compiler to format *iolist* variables solely according to their data types. It is typically used for I/O to and from standard input and standard output files.

List-directed formatting uses the asterisk (*) as a format identifier instead of using edit descriptors. For example, in the following

```
real*8 a, b
    .
    .
    .
read *, a, b
```

the **read** statement formats and stores **a** and **b** as **real*8** values.

If you're reading **integer, real, double precision, complex,** or **logical** values with list-directed formatting, each value must have the same data type as its corresponding *iolist* entity. You must separate values in the input record with one or more blanks or a comma. A slash marks the end of the data, but the slash is optional.

For example, if your program includes this

```
integer high, low
    . . .
read *, high, low
```

your input can be in either of these forms:

```
                                 {input values separated        }
234 89                           {                   - by a blank}
234,89                           {                   - by a comma}
```

If the value is **complex**, both its real and its imaginary parts must appear in the input record, separated by a comma and enclosed in parentheses. For example, this is a valid **complex** input value:

```
(3.E2,5)
```

You cannot include blanks between the real and the imaginary parts of **complex** values in the input record. Complex values are output in the same format.

With list-directed formatting, FORTRAN outputs logical values as either **T** (for **.true.**) or **F** (for **.false.**).

When you input **character** data using list-directed formatting, *you must explicitly enclose* the data in single quotes. If you don't, your program won't work correctly. For instance, this input is correct:

```
'this character string has single quotes around it'
```

Note that the requirement for single quotes around **character** data applies only to input using list-directed formatting. If you use the A edit descriptor to designate the format of **character** data, you do not have to enclose the data in single quotes. See the example of the A edit descriptor earlier in this section for more information.

Example of List-Directed Formatting

```
*   This shows how to use list-directed formatting in print, read, and
*   write statements with integer, logical, complex, and character
*   data types.

      program list_fmt_example

      integer*4     high, low
      character*15  a_word
      logical       yes, no
      complex       complicated
      data  yes, no /.true., .false./, complicated/(6.E3,4.342)/

      print *, 'Enter two integers:'
      read *, high, low
      print *, 'The numbers are:'
      write (*,*) high, low

      print *, 'Enter a word - don''t forget the single quotes:'
      read *, a_word
      print *, 'Your word is: '
      write (*,*) a_word

      print *, 'The logical variables have these values: ', yes, no
      print *, 'The complex variable equals: ', complicated

      end
```

Using This Example

This program is available online and is named **list_fmt_example**. Following is a sample run of the program:

```
Enter two integers:

The numbers are:
24059382 4932
Enter a word - don't forget the single quotes:

Your word is:
aardvark
The logical variables have these values:  T F
The complex variable equals:  (6000.000,4.342000)
```

go to Transfers control to another statement in a program unit.

FORMATS

go to *label1* [. . . ,*labelN*] {unconditional **go to**}

go to *intvar* [,] (*label1* [. . . ,*labelN*]) {assigned **go to**}

go to (*label1* [. . . ,*labelN*]) [,] *intvar* {computed **go to**}

ARGUMENTS

label Label of the executable statement to which you want to transfer control. If you list multiple labels, separate them with commas.

intvar An integer variable or integer expression. In a computed **go to**, control is transferred to *label1, label2, label3*,..., depending on whether *intvar* equals 1, 2, 3, etc. Execution of the assigned **go to** transfers control to the statement associated with *intvar*.

, The comma is optional between *intvar* and the list of statement labels in an assigned **go to**, and between the list of statement labels and *intvar* in a computed **go to**.

DESCRIPTION

A **go to** statement usually breaks the normal sequence of program execution and transfers control to another labeled executable statement. The target statement may be either before or after the **go to**, but it must be in the same program unit.

> **NOTE:** Both **go to** and **goto** are acceptable.

The target of a **go to** must be an executable statement; it cannot be a **format** statement.

Because **go to** usually breaks the normal sequence of program execution, a program with many **go tos** usually is difficult to follow and therefore difficult to maintain. Also, many **go tos** make it difficult for the compiler optimizer to generate efficient code, and so your compilation or run time or both may be unnecessarily long. Because of these drawbacks, we recommend that you use **go to** sparingly.

The following subsections describe the three kinds of **go to** statements:

- Unconditional
- Assigned
- Computed

Unconditional Go To

An unconditional **go to** simply transfers control from the current statement to the one at *label*. The statement can be as simple as this:

```
      go to 50
          .
          .
          .
 50   {executable statement}
```

The unconditional **go to** can also appear in a logical **if** statement, as follows:

```
      if (low .gt. high) goto 20
```

Assigned Go To

The assigned **go to** is just like the unconditional **go to**, except that instead of specifying a statement label, you specify an **integer*4** variable previously assigned to a statement label via the **assign** statement. For example:

```
      integer*4  int
      assign 50 to int
          .  .  .
      goto int
          .
          .
          .
 50   {executable statement}
```

As with other **go to** statements, the assigned **go to** statement and the statement it jumps to must be in the same program unit.

For more information and further examples of this type of **go to**, see the listing for **assign** in this encyclopedia.

Computed Go To

The computed **go to** provides two or more alternate paths; one path is executed depending on the value of the computed **go to**'s integer expression. For example:

```
integer*4 path
    . . .
go to (20, 40, 60, 80) path
```

In this statement, the destination chosen depends on the value of **path**. It works this way:

If path equals	Jump to statement
1	20
2	40
3	60
4	80

If **path** is less than 1 or greater than 4, the **go to** is not executed. Instead, execution resumes with the statement following the **go to**.

EXAMPLE

```
*   This program uses both unconditional and computed goto's to
*   determine what message should be printed when a user enters
*   a number.  This program could easily have been written without
*   goto's, and we recommend that you use as few goto's as possible in
*   your own programs.

       program goto_example
       intrinsic    mod
       integer*4    int_num, i, even_odd
       character*1  answer

10     i = 2
       print 20
20     format ('Enter an integer: ', $)
       read *, int_num

        even_odd = mod (int_num,2)     {Modulus division.}
       if (even_odd .ne. 0) i = 1
       goto (30,40) i                 {Computed goto.}

30     print *, int_num, ' is an odd number'
       goto 50
40     print *, int_num, ' is an even number'
```

```
50     print 60
60     format ('Again? (Y or N) ', $)
       read (*, '(a1)') answer
       if ((answer .eq. 'y') .or. (answer .eq. 'Y')) goto 10

       end
```

USING THIS EXAMPLE

This program is available online and is named **goto_example**. Following is a sample run of the program:

```
Enter an integer: 56
 56 is an even number
Again? (Y or N) y
Enter an integer: 113
 113 is an odd number
Again? (Y or N) n
```

if Tests one or more conditions and executes one or more statements, depending on the outcome of the tests.

FORMATS

```
if (exp) stmt                                  {logical if}

if (arith_exp) label1, label2, label3          {arithmetic if}

if (exp1) then                                 {block if}
     [stmt1. . .]
[else if (exp2) then
     stmt2. . .] . . .
[else
     stmtN. . .]
end if
```

ARGUMENTS

exp, exp1, exp2	Logical expressions; serve as assertions for the logical or block **if**.
stmt	Any executable FORTRAN statement(s); executed if *exp* is true.
arith_exp	An expression of type **integer, real,** or **double precision**; serves as the assertion for an arithmetic **if.**
label1	Label of the statement that gains control if *arith_exp* is less than zero.
label2	Label of the statement that gains control if *arith_exp* equals zero.
label3	Label of the statement that gains control if *arith_exp* is greater than zero.
stmt1	Statement(s) executed if *exp1* in the block **if** is true.
stmt2	Statement(s) executed if *exp2* in the block **if** is true.
stmtN	Statement(s) executed if all previous expressions in the block **if** are false.

DESCRIPTION

The following subsections describe the three kinds of **if** statements:

- Logical
- Arithmetic
- Block

Logical If

A logical **if** statement specifies that a given statement (*stmt*) should be executed only if the logical expression (*exp*) is true. If *exp* is false, *stmt* does not get executed.

After the **if** statement executes, control passes to the next executable statement in the program unit, unless *exp* is true and the *stmt* directs control to another part of the program.

Notice that a logical **if** does not include the keyword **then**. For example:

```
if ((answer .eq. 'n') .or. (answer .eq. 'N')) done = .true.
```

A logical **if**'s *stmt* can be any executable statement. In the preceding example, *stmt* is an assignment statement; in the following, it is an I/O **print** statement:

```
if (finished) print *, 'Done'   {Assume 'finished' is a logical}
```

Arithmetic If

The arithmetic **if** directs the compiler to select one of three execution paths, depending on the value of *arith_exp*. It is similar to a computed **go to** statement, except that control is based on whether *arith_exp* is less than, equal to, or greater than zero. For example:

```
if (int) 20, 40, 60
```

In this statement, the destination chosen depends on the value of **int**. It works this way:

If int	**Jump to statement**
less than 0	20
equals 0	40
greater than 0	60

The *arith_exp* in this type of **if** statement must be of type **integer**, **real**, or **double precision**.

The arithmetic **if** and all of the statements specified by *label1*, *label2*, and *label3* must be in the same program unit. Also, the target statements must be executable statements; you cannot include the label of a **format** or other nonexecutable statement in the list. However, the labels listed don't have to be those of three separate statements. Any two of the labels may be the same. For example:

```
if (result - 1.0) 15, 15, 20
```

In this case, if the expression **(result - 1.0)** evaluates to a negative number or to 0, control goes to statement 15. If **(result - 1.0)** evaluates to a positive number, control goes to statement 20.

Block If

The block **if** probably is the most frequently used of the three types of **if** statements. A block **if** must begin with an **if** (*exp*) **then** statement and end with an **end if** statement.

It may contain one or more **else if** . . . **then** statements, but only one **else** statement. The keyword **then** must be on the same line as the **if** or **else if**. The statements (*stmt1*, *stmt2*, *stmtN*) must be on separate lines.

When executing a block **if**, FORTRAN begins by evaluating the expression, *exp1*. If it is true, FORTRAN executes *stmt1*, but if it is false, *exp2* in the **else if** portion of the statement is evaluated. If that is true, *stmt2* is executed. If it is false, FORTRAN evaluates any other **else if** expressions present one by one. If all are false, and an **else** clause is present, *stmtN* in the **else** portion is executed.

The following is a simple example of a block **if** statement:

```
if (num1 .lt. num2) then
    bignum = num2
else if (num1 .gt. num2) then
    bignum = num1
else                                        {num1 equals num2}
    print *, 'The numbers are equal'
endif
```

It is important to remember that at most FORTRAN executes only one statement block in an **if...then/else if...then/else** construction. Several expressions may be true, but the run-time system only executes the statement(s) associated with the first true expression. It doesn't look at subsequent expressions.

For example, consider the following code fragment:

```
*   Print weather forecast based on certain conditions.

        logical rain, humid, thunderstorms
           . . .
        if (rain) then
            print *, 'Rain is expected on and off today.'
        else if (humid) then
            print *, 'The humidity will be high today.'
        else if (thunderstorms) then
            print *, 'Chance of afternoon and evening thunderstorms.'
        endif
```

During the summer in many areas, all of the above conditions could be true, but the run-time system would only get as far as the first true one and would never test any remaining expression(s).

You can nest block **if** statements—that is, the *stmt* in an **if** or **else if** can itself be a block **if** statement—but if you do, be sure to include an **end if** statement for every **if then**. For example:

```
        if (year .eq. 1986) then
          if ((month .eq. 3) .or. (month .eq. 4)) then
            print *, 'This is the best time to see Halley''s Comet.'
          end if             {close inner if}
        else
            print *, 'Halley''s Comet isn''t visiting this year.'
        end if               {close outer if}
```

Without the inner **end if**, the **else** would be associated with the inner **if** statement. This would mean the wrong-year message would be printed for any month in 1986 except March and April and (like the comet) wouldn't appear at all for years other than 1986. You also would get a compile error if you only had one **end if** because FORTRAN requires that each block **if** have an **end if**.

EXAMPLE

```
*  This program demonstrates a block if without else if's, a block
*  if with multiple else if's, and a logical if.

      program if_example

      integer*2   age, of_age
      character*1 answer

      print 10
10    format ('Enter an age: ',$)
      read *, age

*  This block if statement does not contain an 'else if' portion, but
*  demonstrates multiple statements within the 'else' part.

      if (age .gt. 17) then
          print *, 'You''re an adult.'
      else
          of_age = 18 - age
          write (*, 15) of_age
15        format ('You have ', I2, ' years before you''re an adult.')
      endif              {close if statement}

      print *, ' '
      print *, 'This part helps you decide whether to jog today.'
      print *, 'What is the weather like?'
      print *, '         rainy  = r'
      print *, '         cold   = c'
      print *, '         muggy  = m'
      print *, '         hot    = h'
      print *, '         nice   = n'
      print 20
20    format (' Enter one of the choices: ', $)
      read (*, '(a1)') answer

*  This block if statement shows that you can have multiple 'else if'
*  portions.  Notice that an 'end if' is always required at the end
*  of a block if statement.

      if (answer .eq. 'r') then
          print *, 'It''s too wet to jog today. Don''t bother.'
      else if (answer .eq. 'c') then
          print *, 'You''ll freeze if you jog today. Stay indoors.'
      else if (answer .eq. 'm') then
          print *, 'It''s no fun to run in high humidity. Skip it.'
      else if (answer .eq. 'h') then
          print *, 'You''ll sweat too much if you try to jog today.'
          print *, 'So don''t.'
```

```
      else if (answer .eq. 'n') then
         print *,'You don''t have any excuses. You''d better go run.'
      else
         print *, 'You didn''t give a valid answer.'
      end if                  {close if statement}


*  Example of a logical if. It assumes that the user entered one of
*  the valid jogging responses.

      if (answer .ne. 'n') print 25
25    format (/, ' Of course, excuses don''t help your fitness any.')

      end
```

USING THIS EXAMPLE

This program is available online and is named **if_example**. Following is a sample run of the program:

```
Enter an age:
You have  3 years before you're an adult.

 This part helps you decide whether to jog today.
 What is the weather like?
         rainy  = r
         cold   = c
         muggy  = m
         hot    = h
         nice   = n
 Enter one of the choices:
 It's too wet to jog today.  Don't bother.

 Of course, excuses don't help your fitness any.
```

implicit Overrides the default typing rules.

FORMAT

implicit *data_type1*(*range*1 [*,range2*] . . .) [*,data_type2*(*range*1 [*,range2*] . . .)] . . .

ARGUMENTS

data_type One of the predeclared data types; namely, **byte**, **integer**, **integer*2**, **integer*4**, **real**, **real*4**, **real*8**, **double precision**, **logical**, **logical*1**, **logical*2**, **logical*4**, **complex**, **complex*8**, **complex*16**, **double complex**, or **character**.

range A single letter or a forward range of letters (for example, i–n) to be associated with *data_type*.

DESCRIPTION

The **implicit** statement assigns a particular data type to all variables whose names begin with any of the letters in a specified range.

FORTRAN's default naming conventions only cover integers and real numbers, but you can define your own naming conventions with the **implicit** statement. For example, you can use the **implicit** statement to implicitly declare all variables that begin with the letters A, B, or C to be **logical** variables.

The following are valid **implicit** statements:

```
* All undeclared variables beginning with 'B' are integers.

      IMPLICIT INTEGER(B)

* All undeclared variables beginning  with 'X','Y', or 'Z' are logicals.

      IMPLICIT LOGICAL(X-Z)

* Undeclared variables beginning with 'a', 'b', 's', 't', 'u', or 'v'
* are REAL*8s.

      IMPLICIT REAL*8(A-B,S-V)
```

Because **implicit** requires a forward range of letters, a declaration like the following is invalid:

```
     IMPLICIT DOUBLE PRECISION(C-A)     { WRONG! }
```

Chapter 4. Code

The **implicit** statement must precede all other specification statements (except **parameter**) in the program unit. That is, if you're going to define any **implicit** ranges, you must do so before you start defining explicit variables. However, if you do make an **implicit** declaration, any contradictory explicit data type declarations that follow override the **implicit** statement.

For example, suppose your program includes the following:

```
IMPLICIT  LOGICAL(N-T)
REAL      TOTAL, TEMPERATURE
INTEGER*2 SIZE
```

The **implicit** statement in this example declares all variables beginning with the letters N through T to be **logicals**. But the explicit declarations that follow override the **implicit** statement. So **total** and **temperature** are **reals**, and **size** is an **integer*2.**

You cannot use the **implicit** statement to define the data types of FORTRAN intrinsic functions.

Be careful when using the **implicit** statement. In a long program, you might not notice an **implicit** declaration, and therefore might use a variable thinking it's one data type when the **implicit** declaration makes it another. For example, suppose you use **implicit** to declare that all variables beginning with X, Y, or Z are **characters**. But then you use an undeclared variable **znum** in an arithmetic statement. Under the usual naming conventions, **znum** would be a **real**, which is what you want it to be. But the **implicit** statement makes it a **character**, and now your program has a bug. Explicit variable declaration lets you avoid this problem. (You can use the **implicit none** statement to enforce explicit variable declaration; refer to the description of this statement in this encyclopedia.)

EXAMPLE

```
* This program prints a feet-to-meters conversion table and
* uses the implicit statement to override FORTRAN's default
* naming conventions so that undeclared variables beginning
* with the letter 'f' (for example, 'feet') are typed as
* reals and variables beginning with the letter 'm' (for
* example, 'meters') are typed as integers.

      PROGRAM IMPLICIT_EXAMPLE

      IMPLICIT INTEGER(F), REAL(M)
      REAL CONVERSION_FACTOR
      PARAMETER(CONVERSION_FACTOR = 3.28)

      PRINT *, 'Feet     |     Meters'
      PRINT *, '---------+----------'
      DO 10 FEET = 1, 10
           METERS = FEET / CONVERSION_FACTOR
           PRINT 20, FEET, '|', METERS
   10 CONTINUE
   20 FORMAT (' ', I3, 5X, A, 5X, F5.3)

      STOP
      END
```

USING THIS EXAMPLE

This program is available online and is named **implicit_example**. Following is a sample run of the program:

```
Feet     |     Meters
---------+----------
  1      |     0.305
  2      |     0.610
  3      |     0.915
  4      |     1.220
  5      |     1.524
  6      |     1.829
  7      |     2.134
  8      |     2.439
  9      |     2.744
 10      |     3.049
```

FORMAT

implicit none

ARGUMENTS

None.

DESCRIPTION

The **implicit none** statement causes the compiler to issue warning messages whenever it finds undeclared variables within the same program unit in which the statement appears. The **implicit none** statement does not override FORTRAN's default naming conventions and does not disable implied data typing. It has the same effect as the **-type** compiler option (see Section 6.5.32), except that the statement's scope is limited to the program unit.

When the **implicit none** statement is used, no other **implicit** statements can appear in the same program unit.

EXAMPLE

```
* This program illustrates the implicit none statement.  When
* compiling the program, the compiler will issue a warning message
* for the undeclared variable i.  Nevertheless, the program will
* compile and execute correctly.  The program computes the
* triangular_number of a number entered at the keyboard.

      PROGRAM IMPLICIT_NONE_EXAMPLE

      IMPLICIT NONE
      INTEGER TRIANGULAR_NUMBER, NUMBER
      DATA TRIANGULAR_NUMBER /0/

      PRINT *, 'Enter a number:'
      READ *, NUMBER
      DO 10 I = 1, NUMBER
           TRIANGULAR_NUMBER = TRIANGULAR_NUMBER + I
   10 CONTINUE
      PRINT *, 'THE TRIANGULAR NUMBER IS', TRIANGULAR_NUMBER

      STOP
      END
```

USING THIS EXAMPLE

This program is available online and is named **implicit_none_example**. When you compile it, the compiler issues the following warning:

```
**** Warning #80 on Line 21: identifier not explicitly typed i
```

Following is a sample run of the program.

```
 Enter a number:
100
 The triangular number is 5050
```

include — Tells the compiler to replace the statement with an external file. (Extension)

FORMAT

include *'filename'*

ARGUMENT

filename — The pathname of the file you want to include.

DESCRIPTION

The **include** statement inserts the file *filename* into the source file in place of the **include** statement. Enclose *filename* in single quotes.

The standard usage is to place **common** blocks in a file, and **include** that file whenever a copy is needed. For example:

```
subroutine x
  include 'commonblocks'
  ...
  end
subroutine y
  include 'commonblocks'
  ...
  end
```

Filenames must be case-correct.

Note that the **include** statement has exactly the same effect as the **%include** compiler directive. (Refer to the listing for "Compiler Directives" in this encyclopedia for details about the **%include** directive.)

inquire Reports the attributes of a unit or file.

FORMAT

inquire([**unit =**] *unitid* | **file =** *filename*

[
,access = *access_method*
,blank = *blank_val*
,direct = *dir*
,err = *label*
,exist = *existence*
,form = *form*
,formatted = *fmtd*
,iostat = *sfield*
,name = *pathname*
,named = *nmd*
,nextrec = *nextrec*
,number = *num*
,opened = *opened*
,recl = *reclen*
,sequential = *seq*
,strid = *stream_id*
,unformatted = *unfmt*
]

)

ARGUMENTS

unit = *unitid* — *Unitid*: the ID number that a previous **open** assigned to this unit; must be an **integer** constant, variable, or expression; required for **inquire**s by unit number.

file = *filename* — *Filename*: required for **inquire**s by *filename*.

For a description of the optional arguments and more information on these required ones, see the listing for "I/O Attributes" in this encyclopedia.

DESCRIPTION

Inquire lets you check for the existence of a particular file or unit, and determine one or more attributes of an I/O connection.

As the format indicates, there are two kinds of **inquire** statements: **inquire** by unit number, for which you supply the unit ID, and **inquire** by file, for which you supply the filename of the file in question. The **unit** = *unitid* and **file** = *filename* arguments are mutually exclusive: you cannot use both in the same **inquire**. The optional arguments, which specify the attributes you're interested in, are the same for both kinds of **inquire** statements.

FORTRAN returns the attribute information to the variable, array, or substring you associate with each optional argument. Accordingly, you must declare appropriate data types for each variable, array or substring before issuing the **inquire**.

Some of the information **inquire** returns makes sense only if the specified file exists and is open, or if the file is connected to a unit. Thus, the **opened** attribute, which equals **.true.** if the unit/file connection is open and **.false.** if it is not, is the key to most inquiries.

The following attributes are undefined if the specified unit is not connected to a file, or if the specified file does not exist:

- **access**
- **blank**
- **direct**
- **form**
- **name**
- **named**
- **nextrec**
- **number**
- **recl**
- **sequential**

In addition, the **formatted** and **unformatted** attributes may be unknown if the specified unit is not connected to a file, or if the specified file does not exist.

You can use **inquire** for a variety of reasons. An **inquire** by filename lets you determine whether a file already is open and if so, the way in which it was opened. This is valuable if your code accesses a file that could have been opened in several different ways. Based on the information **inquire** returns, your code can take the appropriate action.

The same is true of an **inquire** by unit. If a unit could have been opened in a variety of ways, an **inquire** tells you exactly the way it was opened this time. Also, the **inquire** allows you to determine the stream ID of the unit. This is important if you plan to make a streams call on that unit.

EXAMPLE

```
*  This program reads a user-entered filename, opens the file, issues
*  an inquire by unit id, and prints the file's attributes.

      program inquire_example

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/fio.ins.ftn'
%include '/sys/ins/error.ins.ftn'

      character*80 file_name
      character*4  blanks
      character*8  dir, seq
      character*12 file_form
      logical      nmd, opened_stat
      integer*4    open_status, inq_status

*  Ask the user to supply the filename.
      print *, 'Enter the name of a file, enclosed in single quotes,'
      print *, 'about which you want to inquire.  For example, you '
      print *, 'might type: ''this_name_in_quotes'''
      read *, file_name
*  Open the file and issue the inquire.
      open (unit=10, file=file_name, iostat=open_status)
      if (open_status .ne. 0) then
      print 5
5     format (' Error status on open = ', $)
      call error_$print(open_status)
      endif
      inquire (10, blank=blanks, direct=dir, iostat=inq_status,
     +          named=nmd, name=file_name, opened=opened_stat,
     +          sequential=seq, form=file_form)

*  Print the file's attributes.
      print 10, file_name
10    format (/, ' File: ', A)
      print *, 'has the following characteristics:'
      print 20, blanks
20    format (/, ' Blank = ', A)
      print *, 'Direct = ', dir
      print 30
30    format (' Error status on inquire = ', $)
      call error_$print(inq_status)
      print *, 'Named = ', nmd
      print *, 'Full name = ', file_name
      print *, 'Opened = ', opened_stat
      print *, 'Sequential = ', seq
      print *, 'Form = ', file_form
```

```
close (10)

end
```

USING THIS EXAMPLE

This program is available online and is named **inquire_example**. Following is a sample run of the program:

```
 Enter the name of a file, enclosed in single quotes,
 about which you want to inquire.  For example, you
 might type: 'this_name_in_quotes'
'names_data'
 File: //my_node/my_dir/names_data

 has the following characteristics:

 Blank = NULL
 Direct = UNKNOWN
 Error status on inquire = status 0 (OS)
 Named =  T
 Full name = //my_node/my_dir/names_data

 Opened =  T
 Sequential = YES
 Form = FORMATTED
```

intrinsic Identifies a specific or generic intrinsic function.

FORMAT

intrinsic *function1* [. . ., *functionN*]

ARGUMENT

function	The name of a FORTRAN intrinsic function. (See Appendix C for a complete list.) If you specify multiple functions, separate them with commas.

DESCRIPTION

Intrinsic functions are routines built into FORTRAN to perform such procedures as data type conversion, rounding, and trigonometric operations. Appendix C lists all the intrinsic functions.

The **intrinsic** specification statement identifies one or more FORTRAN intrinsic functions. There is no requirement, however, that you specify a function as **intrinsic** before you can use it. The sample program with this listing works just as well if you omit the line:

```
intrinsic sqrt, iabs
```

Whether or not you declare a function as **intrinsic**, you can use such functions as external functions. However, you don't have to declare the data type of an intrinsic function.

FORTRAN divides many of its intrinsic functions into generic classes, where each class consists of two or more specific functions with different data types. For example, the generic class **sqrt** includes the following specific functions: **sqrt**, **dsqrt**, **csqrt**, and **cdsqrt**, which calculate the square roots of **real**, **double precision**, **complex**, and **double complex** numbers, respectively.

You can include expressions with arithmetic operations as arguments to a function. Each argument to a specific function must match the function's data type; the result has that data type as well. The number of arguments allowed for functions differs, depending on their generic type. Appendix C lists the type and argument requirements for intrinsic functions.

EXAMPLE

```
*  This program uses the intrinsic functions sqrt and iabs to find the
*  square roots of two user-entered numbers, and the absolute value of
*  a user-entered integer, respectively.

      program intrinsic_example
      intrinsic sqrt, iabs      {Specify intrinsic functions.}
      real*4 a, b, result1, result2
      integer*4 num, absolute_num

*  Ask user for numbers.
      print *, 'Enter two positive numbers for which you want the
     + square roots computed:'
      read *, a, b
      print *, 'Enter an integer for which you want the absolute value
     + computed:'
      read *, num

      result1 = sqrt(a)         {Invoke the intrinsic functions.}
      result2 = sqrt(b)
      absolute_num = iabs(num)
      write (*,10) a, result1
      write (*,10) b, result2
10    format ('The square root of ', F5.2, ' is ', F7.5)
      write (*,20) num, absolute_num
20    format ('The absolute value of ', I4, ' is ', SP, I5)

      end
```

USING THIS EXAMPLE

This program is available online and is named **intrinsic_example**. Following is a sample run of the program:

```
Enter two positive numbers for which you want the square roots com-
puted:
9 24
 Enter an integer for which you want the absolute value computed:
-89
The square root of  9.00 is 3.00000
The square root of 24.00 is 4.89898
The absolute value of  -89 is   +89
```

I/O attributes

There are several attributes that you can specify for Domain FORTRAN I/O statements. This section describes those attributes in detail. Not all attributes are valid for all I/O statements—for example, you can't specify a record length (**recl**) in the **close** statement—but the I/O statements' individual encyclopedia listings tell which attributes are acceptable when. This section describes all the available attributes for the following I/O statements:

- **backspace**
- **close**
- **endfile**
- **inquire**
- **open**
- **print**
- **read**
- **rewind**
- **write**

Table 4-12 lists the I/O attributes.

Table 4-12. Domain FORTRAN I/O Attributes

Attribute	Action
access = *access_method*	Specifies whether a file is to be opened for sequential or direct access.
blank = *blank_val*	Specifies whether blanks are to be treated as nulls or as zeros.
direct = *dir*	Tells whether direct access is one of the allowed access methods for the file.
end = *label*	Specifies that control transfers to the *label* statement if an end-of-file is reached.
err = *label*	Specifies that control transfers to the *label* statement if the I/O statement fails for some reason.
exist = *existence*	Describes whether the file or unit exists.
file = *filename*	Identifies by name the file to be accessed.
fmt = *fmt*	Specifies the format for the I/O.
formatted = *fmtd*	Describes whether formatted I/O to the specified unit is allowed.
form = *form*	Describes whether the unit or file is connected for formatted or unformatted I/O.
iostat = *sfield*	Holds the value that describes whether the I/O operation succeeded: 0 if the operation completed without error; nonzero if the operation failed.
name = *pathname*	Describes the specified field's full pathname, which is not necessarily the same as that listed in the **file** = *filename* argument.
named = *nmd*	Defines whether the file connected to the specified unit is named.
nextrec = *nextrec*	Holds an integer entity representing the number of the last record read or written plus 1, or just 1 if none of the records in the specified file have been read or written.
nml = *namelist*	Specifies the namelist that is to be used in the I/O statement.
number = *num*	Specifies the unit number of the I/O connection.
opened = *opened*	Specifies whether a unit or file is open.
rec = *rec_num*	Specifies the number of the first record you are reading; for direct-access files only

(Continued)

Table 4-12. Domain FORTRAN I/O Attributes (Cont.)

Attribute	Action
recl = *reclen*	Specifies the record length, in bytes, of the file you are accessing.
sequential = *seq*	Describes whether sequential access is one of the allowed access methods for the file.
status = *status*	Describes whether the file you're accessing exists, and the purpose for which you're accessing it.
strid = *stream_id*	Holds the stream identifier associated with a specified unit number.
unformatted = *unfmtd*	Describes whether unformatted I/O to the specified unit is allowed.
unit = *ifid*	Specifies the internal file ID for I/O operations.
unit = *unitid*	Specifies the external unit to which the file is connected.

The following subsections describe these arguments in detail.

access = *access_method*

The **access** attribute refers to the method by which you're accessing the file or unit for I/O: either sequentially or directly. When used with **open**, *access_method* must equal one of the following character constants:

- 'sequential'
- 'direct'

You must enclose the constant in single quotes. You may also place a character expression to the right of the equal sign, as long as its value, without trailing blanks, equals 'direct' or 'sequential'.

'Sequential' access is the default. You can specify **access** = 'direct' for an existing file only if it has fixed-length records. For **access** = 'sequential', the record type may be either fixed length or variable.

NOTE: If you specify 'direct' when creating a file, we recommend strongly that you also specify the record length using **recl** =. If you do *not* specify the size of the files you create, you are relying on the compiler to set a default record length, and you are risking run–time errors caused by incorrect assumptions about the size of your files.

When **access** is used with the **inquire** statement, it indicates whether the file or unit is being accessed sequentially or directly. In this case, *access_method* is a **character** variable or array element, and FORTRAN returns either the value 'sequential' or 'direct', depending on the access attribute **open** specified.

blank = *blank_val*

The **blank** attribute refers to the way in which you want FORTRAN to handle blanks for an I/O connection: either as nulls or as zeros. (Refer to **BZ** and **BN** in the description of the **format** statement.) When used with the **open** statement, *blank_val* can take the following values:

- 'null'
- 'zero'

'Null' is the default; it tells Domain FORTRAN to treat blanks as nulls. 'zero' says you want blanks handled as zeros. You may also place a character expression to the right of the equals sign, as long as its value, without trailing blanks, equals 'zero' or 'null'.

When you use **blank** with the **inquire** statement, the attribute returns the way in which blanks are handled for the specified I/O connection. In this case, *blank_val* is a **character** variable to which FORTRAN returns either 'null' or 'zero'.

Note that if you use the **form** = 'unformatted' attribute elsewhere in your I/O statement, the **blank** attribute is undefined.

direct = *dir*

This attribute, which you use with the **inquire** statement, describes whether direct access is one of the allowed access methods for the specified file. *dir* is a **character** entity to which FORTRAN returns one of the following values:

- 'yes', if direct access is allowed
- 'no', if direct access is not allowed
- 'unknown', if FORTRAN cannot determine whether direct access is allowed

end = *label*

The **end** attribute specifies the *label* of an executable statement to which FORTRAN will transfer control when it reaches the end-of-file. That executable statement must be in the same program unit as the I/O statement that refers to it. For example, the following fragment says to read a file, and when the end-of-file is reached, transfer to the statement labeled 100. That statement closes the file.

```
      read (4, '(A50)', end=100) in_data
          .
          .
          .
100   close(4)
```

If you use the **iostat** attribute as well as **end** in the I/O statement, FORTRAN returns -1 to **iostat**'s variable when it finds an end-of-file. If you omit **end** from your I/O statement, the program terminates if FORTRAN encounters an end-of-file.

It is illegal to use this attribute in a **read** statement if the statement also includes the **rec=** I/O attribute.

> **NOTE:** If you use the **end** attribute, the code at *label* is not optimized well. As a result, your program may run more slowly. (Refer to Section 6.5.26 for details about optimization.)

err = *label*

Err directs control to a particular executable statement, designated by *label*, if the I/O statement fails for some reason. The labeled statement must be in the same program unit as the I/O statement. Usually the target statement marks the beginning of an error-handling or security-handling section. For example:

```
      open(unit=10, file='state_secrets', recl=100, err=911)
          .
          .
          .
*  Assume 'answer' and 'top_secret' are character variables.

911   print *,'Unable to open the file. Do you have proper clearance?'
      read (*, '(A1)') answer
      if ((answer .eq. 'y') .or. (answer .eq. 'Y')) then
           print *, 'Enter the password'
           read (*, '(A)') top_secret
                  .
                  .
                  .
```

If you also specify **iostat** in the I/O statement, **iostat**'s variable contains a status code describing the error. See the description of **iostat** for more information on deciphering the status code.

> **NOTE:** If you use the **err** attribute, the code at *label* is not optimized well. You should avoid placing *label* in a time-critical part of your program. (Refer to Section 6.5.26 for details about optimization.)

exist = *existence*

The **exist** attribute, which you use with the **inquire** statement, describes whether the specified file or unit exists. *Existence* is a **logical** variable to which FORTRAN returns either **.true.** if the specified unit or file exists, or **.false.** if it does not exist.

file = *filename*

This attribute allows you to specify the name and location of the file on which you want an **open** or **inquire** statement to work. When you use it in an **open** statement, *filename* can be one of the following character expressions:

- A string in the form '^*n*', where *n* is a number from 1 to 9
- A string in the form '**prompt_string*'
- One of these strings: **'-stdin'**, **'-stdout'**, or **'-errout'**
- Any valid pathname enclosed in single quotes

When you use this attribute in an **inquire** statement, *filename* may only be either the first or last option listed above; that is, a string in the form '^*n*' or any valid pathname. Domain FORTRAN ignores trailing blanks in filenames.

NOTE: Beginning with SR10 filenames *must be case correct.*

If you use the '^*n*' form, the *n*th argument on the command line for executing the program is read as *filename*. (The program name itself is considered the zeroth argument on the command line.) The argument can be a pathname, or one of the following: **-stdin** (standard input), **-stdout** (standard output), **-errout** (error output). Specifying one of these latter files on the command line has the same effect as specifying it as the filename in the program.

For example, suppose a program named **trash** includes this **open** statement:

```
open (2, file='^2', status='old')
```

The command line might look like this:

```
$  trash.bin  >garbage.out  garbage.in
```

The command line says to execute the program **trash**, with standard output redirected to **garbage.out**, and **garbage.in** being the filename the **open** statement is to use.

If instead of using the '^*n*' form for filename, you use the '**prompt_string*' form, whatever you specify for *prompt_string* is displayed on standard output and *filename* is read from standard input. *Prompt_string* can contain any printable characters. For instance:

```
open (2, file='*Enter the filename: ', status='unknown')
```

If you omit *prompt_string* and list only an asterisk, you must still enter *filename* from standard input, but no prompt appears on standard output. Remember, this form is valid only with **open**, not with **inquire**.

If you specify **'-stdin'**, **'-stdout'**, or **'-errout'**, FORTRAN connects the unit being opened to the named stream. You cannot use the **close** statement to close a unit connected to one of these steams. (Instead, reopen the unit on a different stream.)

fmt = *fmt*

The **fmt** attribute indicates the format of the data. Its value may be any of the following:

- The label of a **format** statement.
- An **integer*4** variable previously assigned to a **format** statement label via the **assign** statement.
- A character expression that contains a format string. The format string must be enclosed in parentheses and the entire parenthetical expression enclosed in single quotes.
- The name of a character array that contains a format string.
- An asterisk (*) to indicate list-directed formatting. List-directed formatting directs the compiler to format values according to their data types.

If this attribute is the second one in the I/O statement, the phrase **fmt** = is optional. Here are some examples of the **fmt** attribute.

```
* Write to unit 3, using the format specified at statement 10 for
* variable my_val.
      write (3, 10, err=100) my_val

* Read from unit 4, using the format string for variables first, mi,
* and last.  Notice the character expression is enclosed in quoted
* parentheses.
      read(4, '(A10,1X,A1,1X,A15)') first, mi, last

* Read from unit 2, using the format specified at statement 20 for
* variable 'batting_average.' The phrase fmt = is necessary because
* the format attribute is not the second in the list.
      read(2, err=25, fmt=20) batting_average
```

form = *form*

The **form** attribute describes whether a specified unit or file is connected for formatted or unformatted I/O. When you use it with the **open** statement, *form* can take the following values:

- 'formatted' (ASCII)
- 'unformatted' (binary)

You must enclose the constant in single quotes. You may also place a character expression to the right of the equal sign, as long as its value, without trailing blanks, equals 'formatted' or 'unformatted'.

If the file or unit's **access** = 'sequential', the default for *form* is 'formatted'. If the **access** attribute is 'direct', the default **form** is 'unformatted.'

If you are connecting an existing file for formatted I/O, its file type must be either UASC or record-structured, and the ASCII/binary flag in its streams header must be ASCII. If you are connecting an existing file for unformatted I/O, its file type must be record-structured, and the ASCII/binary flag, binary. UASC files contain ASCII text, and are fundamentally unstructured but record-oriented; that is, the newline character delimits records.

By default Domain FORTRAN sets the appropriate file type and streams header flag when you use **open** to create a formatted or unformatted file. UASC is the default file type for all files created by Domain FORTRAN.

When you use **form** with the **inquire** statement, it indicates whether the unit or file is connected for formatted or unformatted I/O. In that case, *form* is a **character** variable or array element to which FORTRAN returns either 'formatted' or 'unformatted', depending on the way the connection was opened.

formatted = *fmtd*

The **formatted** attribute describes whether formatted I/O to the specified unit is allowed. You use the **formatted** attribute with the **inquire** statement. *fmtd* is a **character** entity to which FORTRAN returns one of the following values:

- 'yes', if formatted I/O is allowed
- 'no', if formatted I/O is not allowed
- 'unknown', if FORTRAN cannot determine whether formatted I/O is allowed

iostat = *sfield*

Iostat is a valid attribute for all I/O statements; it directs FORTRAN to return a 32-bit status code to *sfield,* a previously declared **integer*4** variable. *sfield* gets one of the following values:

- 0, if the I/O operation completes successfully
- -1, if the operation results in an end-of-file condition
- Any other integer, if the operation results in an error

In order to find out what the status code means, you must insert the include files **/sys/ins/base.ins.ftn** and **/sys/ins/error.ins.ftn** in your source code, and call the system routine **error_$print** with *sfield*. The routine prints out the message associated with this value of *sfield*. For example:

```
%include /sys/ins/base.ins.ftn
%include /sys/ins/error.ins.ftn  {include the error status codes file}
      integer*4 open_status
         . . .
      open(2, file='records', iostat=open_status, status='old')
      call error_$print(open_status)
```

For more information on **error_$print** and on status codes, see the *Domain/OS Call Reference* and *Programming with Domain/OS Calls.*

name = *pathname*

The **name** attribute, which you use with the **inquire** statement, describes the specified file's full pathname. That full pathname is not necessarily the same as the filename in the **file** = *filename* attribute because *filename* takes into account your working directory and any links there might be.

If you use the **file** attribute to specify a file *n*, where *n* is an unopened file, then the **name** attribute associated with *n* will return only the name of the file and not the full pathname.

pathname is a **character** variable or array element to which FORTRAN returns the pathname. The **name** attribute is undefined if the file is not named; that is, if the **named** attribute (see below) returns a value of **.false**.

named = *nmd*

The **named** attribute gets a value which tells whether the file connected to the specified unit is named. Although most files are named, those designated 'scratch', for example, are not. *nmd* is a **logical** entity to which FORTRAN returns **.true**. if the file is named, or **.false**. if it is unnamed.

Use this attribute with the **inquire** statement.

nextrec = *nextrec*

nextrec is an **integer** entity to which FORTRAN returns one of the following values:

- *last_rec* + 1, where *last_rec* is the number of the last record read or written in the specified file, or
- 1, if the specified file is connected but none of its records have been read or written.

nextrec is undefined if the file is connected for sequential access (**access** = 'sequential'). Use the **nextrec** attribute with the **inquire** statement.

nml = *namelist*

namelist is a synonym for a list of variables and array names. *namelist* must have been defined in a **namelist** statement. For additional information as well as an example, refer to the listing for **namelist** in this encyclopedia.

number = *num*

The **number** attribute returns the unit number of the I/O connection. *num* must be **integer** entity. FORTRAN returns to *num* the unit number to which the specified file is connected. Use this attribute with the **inquire** statement.

opened = *opened*

The **opened** attribute, which gets a value telling whether a unit or file is open, is used with the **inquire** statement, and it is the key to many of that statement's other attributes. That is, if the unit or file is not opened, many other attributes are undefined. *Opened* is a **logical** entity to which FORTRAN returns **.true**. if the file or unit is connected for I/O, or **.false**. if it is not.

rec = *rec_num*

The **rec** attribute holds the number of the first record you're reading or writing. Use this attribute only if the file was opened for direct access (that is, your **open** statement includes **access** = 'direct').

Use this attribute with **read** and **write** statements.

There are several conditions under which you cannot use this attribute. If your code includes a **read** statement with the **end=** I/O attribute, **rec=** cannot also appear. Also, **rec=** is illegal with list-directed formatting, with **namelist** I/O, and when the object of the I/O statement is an internal file.

recl = *reclen*

The **recl** attribute describes the record length, in bytes, of sequential and direct access files (both formatted and unformatted types) for the **open** and **inquire** statements. *reclen* must be an **integer** expression.

For sequential files and new direct access files, *reclen* must equal the file's maximum record length, in bytes. For existing direct access files, *reclen* may be any length less than or equal to the actual record length.

The following **open** specifies a *reclen* of 40 for the records in the file **song_titles**.

```
open (unit=33, file='song_titles', recl=40, status='old')
```

Note that you *must* include a **recl** attribute in the **open** statement for an *existing* file. If you omit it, Domain FORTRAN reports an error in the read statement. If you omit the **recl** attribute when *creating* a file, FORTRAN assumes a 256-byte default. We strongly recommend specifying the record length whenever you create a file, rather than relying on the default. Specifying the record length helps you to minimize run-time errors caused by incorrect assumptions about the size of files.

For the **inquire** statement, the **recl** attribute gets a value equal to the specified file's record length. In this case, *reclen* is an **integer** entity to which FORTRAN returns that record length. Note that this attribute is undefined in an **inquire** statement unless the file was **opened** for direct access (**access** = 'direct'.)

sequential = *seq*

This attribute, which you use with the **inquire** statement, describes whether sequential access is one of the allowed access methods for the specified file. *Seq* is a **character** entity to which FORTRAN returns one of the following values:

- 'yes', if sequential access is allowed
- 'no', if sequential access is not allowed
- 'unknown', if FORTRAN cannot determine whether sequential access is allowed

status = *status*

The **status** attribute lets you describe the way you want a file opened or closed. With it, for example, you can specify that a file you are opening already exists ('old'), or that you want to save the contents of a file you are closing ('keep').

There are a number of valid values for the **status** attribute when you use it in the **open** statement. The ANSI standard defines these values as valid:

- 'old'
- 'new'
- 'scratch'
- 'unknown'

Domain FORTRAN also provides these four additional status types for the **open** statement:

- 'append'
- 'write'
- 'readonly'
- 'overwrite'

'Unknown' is the default in Domain FORTRAN. If the file exists, FORTRAN treats 'unknown' status as 'old'. If the file does not exist at the time of the **open**, FORTRAN treats 'unknown' as 'new'.

Use 'old' if the file already exists, and 'new' if it does not exist (that is, you're creating the file as well as opening it). You must supply a filename—that is, you must include the **file** = *filename* attribute in your **open** statement—if you use 'old', 'new', or 'unknown'.

Use 'new' status with caution. When you specify 'new', FORTRAN assumes that the file does not already exist, and that it should create the file before opening it. Consequently, if the file exists when you try to open it with **status** = 'new', you get a run-time error. (One way to avoid this is to **close** such files with **status** = 'delete', which deletes a file after closing it.)

'Scratch' status tells FORTRAN to delete the file after closing it. 'Scratch' files are unnamed. Therefore, you cannot use both the **file** = *filename* and **status** = 'scratch' attributes in the same **open**. If you do, you get a run-time error.

'Append' status is similar to 'unknown'. If the file does not exist at the time of the **open**, 'append' is identical to 'new'. If the file exists, FORTRAN opens it and sets its position to end-of-file.

'Write' status is also similar to 'unknown', except that the file is kept locked for writing as long as it is open. (Normally, FORTRAN keeps a file locked for writing only while writing is actually occurring.) Use 'write' status for a file if you plan to read it and write to it later. This prevents your program from being locked out by another program trying to read the same file.

'Readonly' status is similar to 'old', except that no output is permitted to the file. When you connect a unit with this status, **write**, **print**, and **endfile** statements cause read-only errors.

'Overwrite' status deletes the file if it exists, and then creates a new file with the attributes specified in the **open**. Because the file is deleted first, there is no problem if the attributes defined in the **open** differ from the file's previous attributes. 'Overwrite' is especially valuable when a node or network failure occurs, since the attributes of a file that is open for writing may not be written to disk after such failures.

When you **open** **'-stdin'**, **'-stdout'**, or **'-errout'**, Domain FORTRAN ignores the **status** specifier.

Following are some examples of **open** statements with different values for the **status** attribute:

```
*  This file already exists.
      open(2, file='gerontology', status='old')

*  Writing to the file is not permitted.
      open(unit=4, file='eyes_only', status='readonly')

*  This file might already exist, but it might not.
      open(3, file='mystery', status='unknown', err=100)
```

The **close** statement has a different list of valid status specifiers. They are:

- 'keep'
- 'delete'

The default **close** status is 'keep' unless you open a file with **status** = 'scratch'. In that one case, the default **close** status is 'delete'.

When specifying one of the character constants for the **status** attribute in either an **open** or **close** statement, you must enclose the constant in single quotes. Note, too, that you may use a character expression in place of the character constants 'old', 'new' or 'keep' as long as its value, without trailing blanks, equals one of the allowed character constants.

strid = *stream_id*

You use the **strid** attribute with an **inquire** or **open** statement.

When you use **strid** with an **inquire** statement, FORTRAN returns the stream id associated with the specified unit number to *stream_id*, which must be an **integer*2** variable. For example:

```
integer*2 id_number
    . . .
inquire(unit=3, strid=id_number)
```

When you use strid with an open statement, FORTRAN attaches a FORTRAN unit to the system stream that you specify with the strid attribute. Thus, you can attach streams opened with the ios_$open system call to a FORTRAN unit and perform FORTRAN I/O to that stream. For example,

```
integer*2 iosid
iosid = ios_$open(pathname, name_length, open_options,status)
open (unit = 3, strid = iosid)
read(3) mystr
```

Note that iosid must be an integer*2 variable.

When you use the open statement to associate a FORTRAN unit number with *stream_id*, make sure that you set the I/O attributes that are appropriate to the file you are opening. In other words, if the file is a binary file, you must specify form='unformatted' in the open statement. If the file is an ASCII file, you can take the default, form='formatted'.

unformatted = *unfmtd*

The **unformatted** attribute describes whether unformatted I/O to the specified unit is allowed; you use it with the **inquire** statement. *Unfmtd* is a **character** entity to which FORTRAN returns one of the following values:

- 'yes', if unformatted I/O is allowed
- 'no', if unformatted I/O is not allowed
- 'unknown', if FORTRAN cannot determine whether unformatted I/O is allowed

unit = *unitid* | *ifid*

The **unit** attribute is the most commonly used of all the I/O attributes because it is a required attribute for all I/O statements except an **inquire** by filename. **Unit** specifies the unit on which the specified I/O statement is to operate.

Unitid is the **integer** constant, variable, or expression that identifies the unit. Domain FORTRAN recognizes unit numbers in the range 0 to 99. You can connect a file to more than one unit at a time. Note, however, that **inquire** statements on files connected to more than one unit may fail.

Units 0, 5, 6, and 7 are the only preconnected units. Unit 0 by default is connected to **errout**. Standard input is associated with unit 5, and standard output is associated with units 6 and 7.

In input and output statements, you can use an asterisk (*) to represent the standard input and output files. In **read** statements, standard input is the default if no unit is specified.

In **print** statements, standard output is the default. Typically, standard input is the keyboard and standard output is the display. There is no need to open preconnected units with the **open** statement.

Ifid is the file's internal file ID and is only valid with the **read** and **write** statements. Choose *ifid* if you want to do a memory to memory data transfer—that is, if you want to tell a **read** or **write** statement the memory location at which to place the specified data. This also is known as an internal **read** or **write**. (Internal reads and writes were done in the past with **encode** and **decode** statements; Domain FORTRAN still supports these older statements.)

Ifid must be a **character** string or variable that names the file. If you specify an internal file, you *must* specify a format (see the **fmt=** I/O attribute earlier in this section), but that specification cannot be for list-directed I/O.

If you specify *ifid*, you cannot also specify *unitid*.

The phrase **unit** = is optional if this attribute is the first one listed in the I/O statement.

Following are some examples of the **unit** attribute:

```
* Read from the file connected to unit 15, using the format specified
* at the statement labeled 10.
      read (15, 10) percentages

*  Close the file connected to unit 30, and if an error occurs, jump
*  to the statement labeled 999.
      close(30, err=999)

*  Open the existing file 'class_data' on unit 4. The phrase unit = is
*  required since the attribute is not the first one listed.
      open(file='class_data', status='old', unit=4)

*  Read the internal file 'inside', using the format at statement 25.
      character*10 inside
      read (inside, 25, iostat=int_num, end=10) file_data
```

namelist Defines a synonym for a list of variables and array names. (Extension)

FORMAT

namelist */name/ var1* [. . . *,varN*]

ARGUMENTS

name The name with which the list is associated.

var The names of one or more variables or arrays separated by commas.

DESCRIPTION

The **namelist** statement defines a synonym (known as a **namelist**) for a list of variables and array names. You can use this synonym in I/O statements, including statements that refer to internal files, in place of all the individual variables.

EXAMPLE

The following example creates and refers to a namelist:

```
*   This program prompts the user to initialize some or all of
*   the values for the variables in a namelist and then prints
*   the values for all of the variables in the namelist.

      program namelist_example

      logical          finished
      integer          i(5), j(10)
      double precision big_num
      character*12     word
      complex          c
*   Create the namelist, my_vars, as a synonym for the
*   previously declared variables.
      namelist /my_vars/ finished, i, j, big_num, word, c

*   Read in and then write out the values for the namelist.
      print *, 'Enter the values for the namelist called my_vars.'
      print *, 'Remember to type a dollar sign followed by the '
      print *, 'name of the namelist--in this example, you should'
      print *, 'type: ''$my_vars <return>''.Next you type a line'
      print *, 'for each variable you wish to initialize. '
      print *, 'For example, the second line you type might be: '
```

```
print *, '''finished = T''.  Don''t use commas in any   '
print *, 'numbers that you enter.  Indicate that you have '
print *, 'no more data by typing a dollar sign ($).'
read (*, nml = my_vars)
write (*, nml = my_vars)
end
```

This example allows a user to enter the values for the variables grouped together in the namelist **my_vars**; it writes those values back out to standard output. But you need to know how to enter those values for the program to work correctly.

At run time, the **read** statement waits for you to enter a dollar sign ($) followed by the name of the namelist (in this case, **my_vars**). The characters **$my_vars** indicate that you are about to enter data.

At this point you can enter values for some, none, or all of the variables (**finished**, **i**, **j**, **big_num**, **word**, **c**) of the namelist. When you're done, you must enter a dollar sign ($) to signal the program that you have finished entering values. For example, suppose you run the program and respond to the **read** statement this way:

```
$ my_vars
finished = T
i = 7,3,4,9,46
j = 9*5,1
$
```

In the data for the 10-element integer array **j**, the asterisk (*) indicates that you are using a repeat count. By specifying **9*5**, you tell the **read** to assign the value 5 to the first nine elements of the array.

USING THIS EXAMPLE

Given the above data, the **write** statement generates the following information. Notice that the **write** shows a value for **big_num** even though you did not assign it a value in the **read** statement. That value equals whatever happened to be in **big_num**'s memory location when the program was invoked.

```
$ my_vars
   finished = T,
   i = 7, 3, 4, 9, 46,
   j = 5, 5, 5, 5, 5, 5, 5, 5, 5, 1,
   big_num = 0.0000000000000000,
   word ='',
   c = (0.0000000,0.0000000) $
```

---

open Connects the specified unit to a file and establishes the connection's I/O attributes.

FORMAT

```
open(unit = unitid
        [
        ,access = 'sequential' | 'direct'
        ,blank = 'null' | 'zero'
        ,err = label
        ,file = filename
        ,form = 'formatted' | 'unformatted'
        ,iostat = sfield
        ,recl = reclen
        ,status = 'old' | 'new' | 'scratch' | 'unknown' |
                  'append' | 'write' | 'readonly' | 'overwrite'
        [illegible] = stream_identifier
        ]
    )
```

ARGUMENT

unitid An **integer** constant, variable, or expression that identifies this unit. Domain FORTRAN allows unit numbers from 0 through 99. Unit 0 is preconnected to error output, unit 5 is preconnected to standard input, and units 6 and 7 are preconnected to standard output.

For more information on this argument, and a description of **open**'s optional arguments, see the "I/O Attributes" listing in this encyclopedia. Also see Section 8.1 for more information about Input/Output Stream (IOS) calls, including stream identifiers.

DESCRIPTION

Open establishes a connection between a unit and a file and establishes attributes for the connection. The unit/file connection established by **open** exists for all program units, not just the one in which the **open** statement appears. The connection remains open until you break it with **close**.

If the file you specify does not already exist, FORTRAN creates it, unless you specify **status** = 'old' or **status** = 'readonly'.

If a given unit is already connected to a file and you **open** that unit for a different file, FORTRAN breaks the original connection and reconnects the unit to the new file.

Table 4–13 shows what the compiler does if you omit the filename argument (**file** = *filename*) and the specified unit is not connected to a file.

Table 4–13. Domain FORTRAN Filenames with **Open** *Statement*

Status Attribute	Method of Compiling	What the Compiler Does
'scratch'	all methods	Creates an unnamed file and connects it to the unit. Unnamed files are always deleted on **close**.
'unknown' 'new' or if the attribute is omitted	**ftn** (without **-uc**)☆	Supplies the name **FOR0*nn*.dat**, where *nn* is the unit id number
'unknown' 'new' or if the attribute is omitted	**ftn** (with **-uc**)☆ **f77**	Supplies the name **fort.*n*** where *n* is the specified unit id number.
☆Refer to Section 6.5.34 for details about the -uc compiler option.		

When you use **strid** with an **open** statement, FORTRAN attaches a FORTRAN unit to the existing stream that you specify with the **strid** attribute. Thus, you can attach streams opened with the **ios_$open** system call to a FORTRAN unit and perform FORTRAN I/O to that stream. Refer to the listing for "I/O Attributes" in this encyclopedia for an example of **strid**=stream_id.

EXAMPLE

```
*  This program demonstrates a variety of opens.
*   NOTE: You must obtain file "names_data" before running this
*   program.  You must store names_data in the same directory as
*   open_example.bin.

      program open_example

      character*26 s_name
      character*50 stuff, user_name
      integer*4    int_num, other_num

*  Ask the user to supply a filename and a blank-embedded number.
      print *, 'Enter the name of a file, enclosed in single quotes,'
      print *, 'into which you want to load numeric data.'
      read *, user_name
      print 10
10    format ('Enter an 8-digit integer with embedded blanks: ', $)
      read (*, 20) int_num
20    format (BZ, I8)

*  This open includes a variable to indicate a user-supplied file
*  name.  It also specifies that the status is unknown -- the file
*  might or might not already exist -- and that blanks on input
*  are to be treated as zeroes.
      open (10, file=user_name, blank='zero',
     +      status='unknown', access='sequential')
      write (10, '(I8)') int_num
      rewind 10
      read (10, '(I8)') other_num
      print *, 'Your file contains: ', other_num

*  These open statements include a literal pathname for the file
*  attribute and specify the record lengths.  Further, the first
*  dictates that its file may only be read, while the second
*  deletes the named file if it already exists.
      open (3, file='names_data', recl=26, status='readonly')
      open (4, file='new_names', recl=50, status='overwrite')




*  Read input file, write its data to output file, and print
*  the contents of the output file.
      read (3, '(A)') s_name
```

```
stuff = s_name
write (4, '(A)') stuff
print *, 'File new_names now contains: ', stuff
close (3)
close (4)
close (10)
end
```

USING THIS EXAMPLE

This program is available online and is named **open_example**. Following is a sample run of the program:

```
 Enter the name of a file, enclosed in single quotes,
 into which you want to load numeric data.
'my_numbers'
Enter an 8-digit integer with embedded blanks:  7 3 1946
 Your file contains:  70301946
 File new_names now contains: Stewart   MFranklin
```

options — Inserts compiler options in source code. (Extension)

FORMAT

options *opt1* [. . . ,*optN*]

ARGUMENT

opt — One or more of the following compiler options:

–cond or **–ncond**	**–indexl** or **–nindexl**
–config	**–info** or **–ninfo**
–dynm or **–save**	**–inline**
–ff	**–l*1** or **–l*2** or **–l*4**
–frnd or **–nfrnd**	**–subchk** or **–nsubchk**
–i*2 or **–i*4**	**–type** or **–ntype**
–idir	**–warn** or **–nwarn**

If *opt* is not one of the options listed above, the compiler issues a warning and ignores the option.

DESCRIPTION

The **options** statement allows you to insert certain compiler options into a source file. The specified options apply to the program unit in which the **options** statement appears, overriding any options specified on the command line. Following the compilation of the program unit, the options revert to those entered on the command line.

The **options** statement must always appear before a **program**, **function**, or **subroutine** statement. If it appears elsewhere, an error occurs.

Suppose that the source file includes the following fragment:

```
options -cond -i*2 -type

subroutine do_nothing(one_int, another_int)

integer one_int, another_int
   .
   .
   .
return
end
```

The **options** statement preceding the subroutine **do_nothing** specifies the options that the compiler is to use when compiling this program unit only, regardless of what options may have appeared on the command line. Specifically, the compiler is to do the following:

- Compile lines marked with the **debug** directive
- Use **integer*2** as the default integer type
- Issue warning messages for variables not explicitly declared

These options apply only to **do_nothing**, not to any program units that may appear before or after it in the source file.

parameter Assigns a symbolic name to a constant.

FORMAT

parameter (*name* = *value* [, . . .])

ARGUMENTS

name The symbolic name of a constant (value). If you assign more than one *name*, separate the *name* = *value* pairs with commas.

value A constant, or an expression that evaluates to a constant. The value of *value* is converted to the same data type as *name*, if necessary. If *value* is an expression, it cannot contain variables, array element references, or function references.

DESCRIPTION

The **parameter** statement assigns a particular symbolic name, *name*, to a constant or constant expression. You can then use that symbolic name just as you would use the constant.

The **parameter** statement's location in a program is important. If you explicitly declare the variable (*name*) to which you are assigning the constant, the declaration *must* precede the **parameter** statement. And in all cases, a **parameter** statement must precede statements that reference the constant it defines. The following example shows the proper order of declaration and **parameter** statements. (In this example and those that follow, the names appear in uppercase letters to help them stand out.)

```
integer*2 i, ARRAY_SIZE        {Explicitly declare variables. }
parameter (ARRAY_SIZE = 50)    {Assign constant to ARRAY_SIZE.}
real*4    log_table(ARRAY_SIZE) {First use of ARRAY_SIZE.     }
    . . .
do i=1,ARRAY_SIZE
       . .
end do
```

As with variable declarations, you should always explicitly declare these *names*.

The **parameter** statement provides a way to make program maintenance easier. Suppose you have a program that includes an array whose current maximum size is 100. You realize, however, that the size could change in the future and you want to simplify future modifications. If you use **parameter** to set the array's size, when that size changes you only need to change one line of code, rather than searching for all occurrences of the maximum size value.

The **parameter** statement also can help you avoid typing mistakes. If you are working on a program that uses the value of pi (π) often, it is possible to mistype 3.14159 at some point. It is much easier to do this:

```
real*4 PI, area, radius
parameter (PI = 3.14159)
   . . .
area = PI * (radius * radius)   {Find the area of a circle.   }
```

The **parameter** statement differs from the **data** statement in that once you use **parameter** to assign a constant value to a *name*, you cannot change that value for the rest of the program unit. After you define a symbolic name with **parameter**, you can use that name in an expression, a **data** statement, or in subsequent **parameter** statements. For example:

```
integer*2 INT_NUM, J
parameter (J=3)
parameter (INT_NUM = J+10)
```

You can use an asterisk as an indefinite length specification for a **character** entity if you later define that entity with a **parameter** statement. For instance:

```
character*(*) BIG_WORD
parameter (BIG_WORD = 'antidisestablishment')
```

In this example, the second asterisk (*) in the **character** type declaration indicates that **BIG_WORD** has an indefinite length. FORTRAN derives **BIG_WORD**'s length (20 characters) from the **parameter** statement.

EXAMPLE

```
*   This program demonstrates how to use an indefinite character length
*   specification with parameter statements, and it assigns a value to
*   a constant named PI so that the value only needs to be typed once.
*   All constant names are capitalized to help you see them easily.

       program parameter_example
       character*(*) ONE, TWO, THREE, FOUR
       parameter (ONE='The ',TWO='quick ', THREE='brown ', FOUR='fox ')

       real*4       PI, radius
       real*8       area, cir
       character*1 answer
       logical      again
       parameter (PI = 3.14159)              {Specify the value of PI.}
       again = .true.

       print *, ONE, TWO, THREE, FOUR
       print *, 'jumps over the lazy white dog.'
       print 5
5      format (/, 'And now, on to more important things.', /)

*   This segment finds the circumference and area of a circle
*   with a user-entered radius.

       do while (again)
           print 10
10         format ('Enter the circle''s radius: ', $)
           read *, radius
           cir = 2 * PI * radius             {Notice use of constant. }
           area = PI * (radius * radius)     {And here it is again.   }
           print 20, cir
20         format ('The circle''s circumference is: ', F10.6)
           print 30, area
30         format ('Its area is: ', F10.6)
           print 40
40         format (/, 'Again? (Y or N) ', $)
           read (*, '(a1)') answer
           if ((answer .eq. 'n') .or. (answer .eq. 'N')) again =.false.
       enddo
       end
```

USING THIS EXAMPLE

This program is available online and is named **parameter_example**. Following is a sample run of the program:

```
 The quick brown fox
 jumps over the lazy white dog.

And now, on to more important things.

Enter the circle's radius: 3.5
The circle's circumference is:  21.991131
Its area is:  38.484478

Again? (Y or N) y
Enter the circle's radius: 7
The circle's circumference is:  43.982262
Its area is: 153.937912

Again? (Y or N) n
```

pause Temporarily stops the program until a user intervenes.

FORMAT

pause [*message*]

ARGUMENT

message A string of digits or a **character** constant enclosed in single quotes.

DESCRIPTION

Whenever a **pause** occurs, the execution of your program is suspended and this message appears on error output:

```
Fortran PAUSE
Type return to continue.
```

The program remains suspended until you press <RETURN> on standard input. If you wish to stop the program's execution completely, type CTRL/Q or the Display Manager command dq

You can expand Domain FORTRAN's built-in message with your own additional information (*message*). For example, you could specify the message 'Press <RETURN>'. If program contains several **pauses**, you might want to insert an integer as the *message* to tell you which **pause** the program has reached.

The **pause** statement is often used in programs that generate multiple graphics images on the screen. If you include a **pause** between the generation of such images, it allows a user the time to view and interpret the image. You might also use **pause** in a program that computes and prints out a lot of data. After 50 lines of output, for example, the program might **pause** to allow the user time to look at the data.

EXAMPLE

```
*  This program computes and prints the square roots of the whole
*  numbers between 1 and 500.  It prints 25 at a time, then pauses
*  to allow the user to look at what has been flashing by.

      program pause_example

      intrinsic sqrt
      integer*2 count
      real*4    num, sqr_root
```

```
      count = 0

*  Loop to compute and print square roots.  The pause is executed
*  after 25 square roots have been printed - except after the
*  final 25, since a pause is not necessary there.

      do 20, num = 1, 500
          sqr_root = sqrt(num)
          write (*,10) num, sqr_root
10        format ('The square root of ', F5.1, ' is ', F8.4)
          count = count + 1
          if ((count .eq. 25) .and. (num .ne. 500.0)) then
              pause '25 more square roots printed'
              count = 0
          endif
20    continue

      end
```

USING THIS EXAMPLE

This program is available online and is named **pause_example**. If you execute the program, you get this output:

```
The square root of   1.0 is   1.0000
The square root of   2.0 is   1.4142
The square root of   3.0 is   1.7321
The square root of   4.0 is   2.0000
The square root of   5.0 is   2.2361
The square root of   6.0 is   2.4495
    .
    .
    .
The square root of  23.0 is   4.7958
The square root of  24.0 is   4.8990
The square root of  25.0 is   5.0000
Fortran PAUSE 25 more square roots printed
Type return to continue
The square root of  26.0 is   5.0990
The square root of  27.0 is   5.1962
The square root of  28.0 is   5.2915
The square root of  29.0 is   5.3852
The square root of  30.0 is   5.4772
    .
    .
    .
```

pointer Enables a program to use pointers returned by programs written in other languages. (Extension)

FORMAT

pointer */pointer_var/based_var_list. . .*

ARGUMENTS

pointer_var An **integer*4** variable to hold the pointer returned by a system call.

based_var_list A list of variables, array, or array elements of any data type. Use commas to separate the entities in the list.

DESCRIPTION

The **pointer** statement is a Domain FORTRAN extension to the ANSI standard. It gives a FORTRAN program access to the pointers returned by programs written in other languages. Thus the FORTRAN program can reference data pointed to by a pointer (to dereference the pointer).

The *pointer_var* must be type **integer*4**, and must be defined as such before you can refer to it within a **pointer** statement. A pointer variable may be local, in **common**, or a dummy argument.

The *based_var_list* can contain one or more variables or arrays of any data type. You can build a structure analogous to a Pascal **record** or a C **struct** with the *based_var_list*. For example, suppose your FORTRAN program contains a call to a Pascal routine that builds a linked list of records containing daily weather data. The Pascal record might be defined like this:

```
type
    date_type = array[1..8] of char;  {For dates in mm/dd/yy format.}
    link = ^weather;
    weather = record
        date : date_type;
        hi_temp, low_temp, rainfall, windspeed : real;
        next : link;
        end;
```

In your FORTRAN program, you would set up a matching structure like this:

```
character*8 date
real*4      hi_temp, low_temp, rainfall, windspeed
integer*4   ptr, next
pointer /ptr/ date, hi_temp, low_temp, rainfall, windspeed, next
```

Domain FORTRAN does not automatically allocate storage for *based* variables. However, you can use the **pointer** statement to dimension an array to be used as a *based* variable (just as you can dimension arrays in **common** statements or type declarations).

You cannot initialize *based* variables with **data** statements, declare them in **common**, or use them as dummy arguments.

When a program written in another language assigns a value to a **pointer** variable, you can reference the pointer variable simply by referencing its based variable(s). You can get to a particular offset of the pointer variable by assigning the based variable to it (*pointer_var* = *based_var*).

For instance, to visit each item in the linked list of daily weather data from the previous example, include the following in your FORTRAN program:

```
do while (ptr .ne. 0)
    {process records}
    ptr = next          {Reset pointer to the next record}
enddo                   {in the list.                    }
```

This fragment demonstrates one other important item for dealing with pointers. When a pointer is set to **NIL** (for Pascal) or **NULL** (for C), FORTRAN sees that as being zero. Assuming you set the pointer field in the last element of a list properly, FORTRAN can find the end of that list simply by testing for a zero condition; that is,

```
do while (ptr .ne. 0)
```

EXAMPLE

Refer to Section 7.4.5 for an example of the **pointer** statement.

print Transfers data to standard output.

FORMAT

print *fmt* [*,iolist*]

ARGUMENTS

fmt Format specification; one of the following:

- The label of a **format** statement.
- An **integer*4** variable previously assigned (via the **assign** statement) to the label of a format statement.
- A character expression or character array.
- An asterisk (*) to indicate list-directed formatting.
- A namelist. If you specify a namelist, you may not include an *iolist* in this **print** statement.

iolist The data to be written to standard output; can consist of the following:

- Variable name(s)
- Array or array element name(s)
- Character substring(s)
- Implied **do** list(s)

DESCRIPTION

The **print** statement directs data to standard output. You typically use **print** to send output to the display, although you can redirect standard output with the shell's redirection notation (**>filename**). For example, if you write a program named **my_prog** and want the results of its **print** statements to go to the file **answers**, you can enter the following at run time:

```
$  my_prog.bin >answers
```

See Chapter 6 for more information on executing Domain FORTRAN programs.

The **print** statement's syntax is slightly simpler than that of the similar **write** statement. That's why it is most often used for data to standard output. For example, compare the two statements' syntaxes for printing a character string:

```
print *, 'Here''s looking at you, kid'

write (*, *) 'Here''s looking at you, kid'
```

EXAMPLE

```
*  This program demonstrates how to use the print statement in
*  a variety of ways.

      program print_example

      integer*2 int_array(10), i
      real*4    hours_worked, hourly_wage, salary  {Declare and}
      real*4    reg_pay, overtime                  {initialize }
      character answer                             {variables. }
      logical   again
      again = .true.

*Print statements with character strings and list-directed formatting.
      print *, 'This part assigns and then prints the values of'
      print *, 'the 10-element array, int_array.'
      do 5, i = 1,10
          int_array(i) = i * 3
5     continue
      print *, 'The array contains: '

*  Print statement with implied do loop and list-directed formatting.
      print *, (int_array(i), i=1,10)

*  Print statement with the label of a format statement.
      print 10
10    format (/, 'This section calculates an employee''s pay.', /)

*  This section computes employees' salaries, including overtime if
*  the number of hours worked is greater than 40.  The loop includes
*  a number of print statements.  Most simply list the label of a
*  format statement, but one includes a label and a variable (salary).

      do while (again)
          print 20
20        format ('Enter the hours worked: ', $)
          read *, hours_worked
          print 25
25        format ('Enter the employee''s hourly wage: ', $)
          read *, hourly_wage
```

```
          if (hours_worked .le. 40.0) then
              salary = hourly_wage * hours_worked
          else                         {the employee worked overtime}
              reg_pay = hourly_wage * 40.0
              overtime = 1.5 * (hourly_wage * (hours_worked - 40.0))
              salary = reg_pay + overtime
          endif                        {close if statement           }
          print 30, salary
30        format ('The salary is ', F7.2)
          print 35
35        format (/, 'Again? (Y or N) ', $)
          read (*, '(a1)') answer
          if ((answer .eq. 'n') .or. (answer .eq. 'N'))
     +       again = .false.
      end do                           {close do/while statement     }

      end
```

USING THIS EXAMPLE

This program is available online and is named **print_example**. Following is a sample run of the program:

```
 This part assigns and then prints the values of
 the 10-element array, int_array.
 The array contains:
 3 6 9 12 15 18 21 24 27 30

This section calculates an employee's pay.

Enter the hours worked:
Enter the employee's hourly wage:
The salary is  699.63

Again? (Y or N)
Enter the hours worked:
Enter the employee's hourly wage:
The salary is  100.00

Again? (Y or N)
```

program Names the main program unit.

FORMAT

program *pgm_name*

ARGUMENT

pgm_name The name you want to assign to the main program unit. It can contain up to 4096 ASCII characters, including underscores (_) and dollar signs ($), but must differ from all subprogram and **common** block names within the program unit.

DESCRIPTION

The **program** statement assigns a symbolic name, *pgm_name*, to a main program unit. You don't have to use the **program** statement. If it is not present, FORTRAN by default names the main program unit **$MAIN** (if you compile with **ftn**) or **main __** (if you compile with **f77** or with the **-uc** option). Note that the last two characters in **main__** are underscore characters.

If you do use **program**, however, it must be the first statement in the main program unit. Note that *pgm_name* does not have to match the main program's filename.

If you compile with **f77**, or using **ftn** with the **-uc** option, the compiler appends an underscore(_) to any subprogram names that do not start or end in an underscore (_) and do not contain a dollar sign ($). The underscore distinguishes the procedure from a C procedure or external variable with the same user-assigned name. Refer to Section 6.5.34 for details about the **-uc** option.

A FORTRAN program can contain only one main program unit—the others must be **function**, **subroutine**, or **block** data subprograms.

EXAMPLE

```
program program_example
print *, 'This very short program demonstrates the use of the'
print *, 'program statement.  Note that this statement is - and'
print *, 'must always be - the first statement in the program.'
end
```

---

read Transfers data from a file to internal storage.

FORMAT

read (*clist*) [*iolist*] {long form}

read *fmt* [,*iolist*] {short form}

ARGUMENTS

(clist) Control list: must be enclosed in parentheses, and can consist of the following:

- **unit** = {*unitid* | *ifid*}. Required for long form **reads**; specifies the external unit to which the file is connected, or the file's internal file ID. The phrase **unit** = is optional if this is the first argument in *clist*.
- **fmt** = *fmt*. Required format specifier. You may use an asterisk (*) to denote list-directed formatting. (List-directed formatting tells the compiler to format values according to their data types.) The phrase **fmt** = is optional if this is the second argument in *clist*.
- **rec** = *rec_num*. Record number, required for reading direct access files. If you use this attribute, you cannot specify that you want to read an internal file or a **namelist**. Also, you can't use the **end**= *label* specifier when this attribute appears.
- **iostat** = *sfield*. Optional I/O status specifier.
- **end** = *label*. Optional end specifier.
- **err** = *label*. Optional error specifier.
- namelist. You can specify the namelist as you would specify a variable name, or you can specify it as **nml** = *namelist*. For example, if you create a namelist called my_vars, you can specify it as my_vars or as *nml* = my_vars. For an example demonstrating a namelist in a *read* statement, refer to the listing for namelist in this encyclopedia.

iolist The data to be transferred; can consist of the following:

- Variable name(s)
- Array or array element name(s)
- Character substring(s)
- Implied **do** list(s)

fmt Format specifier, required for short form **reads**.

For more information about the options associated with a long form **read** (the *clist* options), refer to the listing for "I/O Attributes" in this encyclopedia.

DESCRIPTION

Read transfers data from a file to internal storage, in the format defined by a format specification, or according to the data's type (that is, list-directed formatting).

The *iolist*, which specifies the data you're reading, can consist of variable names, array or array element names, or character substrings. If you include an array name without a subscript, FORTRAN reads all elements of the array and stores them in the order of their array positions.

FORTRAN always tries to read data into all *iolist* entities. When it encounters a slash (/) in the format specification, FORTRAN reads the next record, using the repeatable edit descriptors to the right of the slash for the remaining *iolist* entities.

If the repeatable edit descriptors in the format specification have been used up and there are more *iolist entities*, FORTRAN performs **format reversion**, as follows:

- If there are nested parentheses in the format specification, FORTRAN reverts to the format terminated by the last preceding right parenthesis.
- If there are no nested parentheses, FORTRAN reverts to the beginning of the format specification.

When FORTRAN reverts to a format preceded by a repeat count, it uses the repeat count. The reused portion of the format specification must contain at least one repeatable edit descriptor.

For more information about format reversion and edit descriptors, see the listing for **format** in this encyclopedia.

The *iolist* can also consist of one or more implied **do** loops—abbreviated **do** loops that you typically use to read arrays. For example:

```
read *, first_name, last_name, (grades(i), i=1,6)
```

This statement reads variables **first_name** and **last_name** and six elements of array grades from standard input using list-directed I/O (designated by the asterisk). The implied **do** loop, **(grades(i), i=1,6)** tells the compiler to read in the first six elements of grades, starting with the first element **(i=1)**. Refer to the listing for **do** in this encyclopedia for more information about implied **do** loops.

Each *iolist* item must match its corresponding data item in type. This means, for example, that a **real** value requires a corresponding **real** *iolist* item and an **integer** value requires a corresponding **integer** *iolist* item.

FORTRAN supports two kinds of **read** statements: a short form, for reading from standard input (typically, the keyboard), and a long form.

The short form requires only a format specifier, *fmt*, which can simply be an asterisk (*) for list-directed formatting. The short form's format specifier is exactly like the long form's **fmt** = *fmt* specification. The listing for I/O Attributes in this encyclopedia describes in detail the **fmt** = *fmt* specification and the long form's other *clist* options.

You usually use the long form **read** for reading from files. For example:

```
integer*4 read_status
    . . .
read (4, 20, iostat=read_status, err=999) in_data
```

This statement says to read the file connected to unit 4, using the format specified at the statement labeled 20. The **integer*4** variable **read_status** gets the **read** statement's I/O status; if there is an error, control goes to the statement labeled 999.

EXAMPLE

```
*   This program reads a file and determines whether the left and right
*   parentheses in it are balanced - that is, whether there is an
*   equal number of each.

    program read_example

    character*80 file_name
    character*72 input_line
    integer*4    open_stat
    integer*2    i, left_count, right_count, diff
    data left_count, right_count /0, 0/

*   Ask the user to supply the filename.  Notice that this short form
*   read uses list-directed formatting.

    print *, 'Enter the name of a file, enclosed in single quotes,'
    print *, 'for which you want to check parentheses.'
    read *, file_name

*   Try to open the file.  Stop if it doesn't exist.

    open (unit=10, file=file_name, iostat=open_stat, status='old')
    if (open_stat .ne. 0) stop 'File does not exist'
```

```
*   Use a long form read to read the input file a line at a time.
*   Keep a count of the left and right parentheses encountered.

5       read (10, '(A)', end=100) input_line      {Read input file}
        do i = 1,72                                {line at a time.}
           if (input_line(i:i) .eq. '(') then
              left_count = left_count+1
           else if (input_line(i:i) .eq. ')') then
              right_count = right_count+1
           endif
        enddo
        goto 5

*   Print results.

100     if (left_count .gt. right_count) then
          diff = left_count - right_count
          print 110, diff
110       format(' You have ', I2, ' more left parentheses than right.')
        else if (left_count .lt. right_count) then
          diff = right_count - left_count
          print 120, diff
120       format(' You have ', I2, ' more right parentheses than left.')
        else
           print *, 'Your parentheses are balanced.'
           print 130, left_count
           print 140, right_count
130        format (' You have ', I3, ' left parentheses')
140        format (' and ', I3, ' right parentheses.')
        endif

        close (10)
        end
```

USING THIS EXAMPLE

This program is available online and is named **read_example**. Following is a sample run of the program:

```
 Enter the name of a file, enclosed in single quotes,
 for which you want to check parentheses.
'do_example.ftn'
 Your parentheses are balanced.
 You have  12 left parentheses
 and  12 right parentheses.
```

return Returns control to the calling program unit from a subprogram.

FORMAT

return [*exp*]

ARGUMENT

exp An optional integer constant or expression that resolves to a constant. Use *exp* to indicate an alternate return.

DESCRIPTION

Return, which you may use only in a subroutine or function subprogram, transfers control from the subprogram back to the program unit that called it. (An **end** statement in a subprogram does the same thing as **return.**)

Execution of a **return** statement terminates the subprogram's execution and breaks the association between the subprogram's dummy arguments and the corresponding actual arguments in the calling program unit.

Within a subroutine (*not* a function), you can specify **alternate returns**—directives that pass control to different statements in the calling program unit, depending upon certain conditions. You also can use **return** statements simply to return from multiple entry points in a subprogram. (See the description of the **entry** statement in this encyclopedia.) The following steps describe how to set up alternate returns:

- In the calling program unit, **call** the subroutine, using one or more asterisk/statement–label pairs as actual arguments. The designated statements must be executable statements. For example:

```
call my_sub (x,y,*20,*30)
```

- In the subroutine's **subroutine** statement, use one asterisk as a dummy argument for each asterisk/statement–label pair in the calling program. For example:

```
subroutine my_sub (x1,y1,*,*)
```

- Specify each **return** in the subroutine as follows: **return** *exp*, where *exp* is an integer constant or expression that represents the position of an asterisk in the dummy argument list. That is, if you want to return to the statement associated with the first asterisk in your list, specify **return 1**; if you want to return to the statement associated with the second asterisk, specify **return 2**. Notice that the asterisk count always starts at 1, regardless of how many dummy argument names may precede the first asterisk. In the preceding example, **return 1** returns control to statement 20 in the calling program, and **return 2** returns control to statement 30.

NOTE: If *exp* is less than 1 or greater than the number of asterisks in the dummy argument list, control returns to the statement after **call** in the calling program unit (just as if you used **return** without an argument).

EXAMPLE

```
*This program reads a file of student names and then takes user input
*for five test scores for each student. The subroutine compute_average
*computes each student's average score.  If the average is < 60, the
*student has failed the class, and the subroutine returns to statement
*30 in the main program to print an appropriate message.  If the
*average is >= 60 the subroutine returns to statement 40.

      program return_example

      character*26 s_name
      integer*4    open_stat
      integer*2    i, grades(5)
      real*4       avg
      logical      dummy_true
      dummy_true = .true.

*  Try to open file.  Stop if there's an error.
      open(4, file='names_data', iostat=open_stat,
     +     recl=26, status='readonly')
      if (open_stat .ne. 0) stop 'File open failed'

*  Read each student's name, prompt for test scores, compute
*  average, and print result.
      do while (dummy_true)
          read(4, '(A)', end=100) s_name
          print 10
10        format (/, ' Enter five test scores for student ', $)
          print 20, s_name(1:10), s_name(11:11), s_name(12:26)
20        format (A,1X,A,1X,A)
          read *, (grades(i), i=1,5)

*  Specify alternate return conditions with asterisk/stmt label pairs.
          call compute_average(grades, avg, *30, *40)

30        print *, 'This student will have to repeat the class.'
40        print 50, avg
50        format (' His/her average is: ', F5.2)
      end do

100   close (4)
      end
```

```
************************************************************************
* This subroutine computes a student's average.  It specifies alternate
* returns - if average is < 60, return to statement 30; if not,
* return to statement 40.

      subroutine compute_average(scores, average, *, *)
      integer*2 j, scores(5)
      real*4    total,average

      total = 0
      do 150, j=1,5
          total = total + scores(j)
150   continue
      average = total / 5.0
      if (average .lt. 60.0) then
          return 1                {First alternate return specified. }
      else
          return 2                {Second alternate return specified.}
      endif
      end
```

USING THIS EXAMPLE

This program is available online and is named **return_example.** Following is a sample run of the program:

```
 Enter five test scores for student Stewart    M Franklin
86 75 82 100 87
 His/her average is: 87.00

 Enter five test scores for student Kayla      J Brady
56 78 34 23 40
 This student will have to repeat the class.
 His/her average is: 46.20

 Enter five test scores for student Pierre     Y Giroux
87 [illegible] 56 75 93
 His/her average is: 80.60

 Enter five test scores for student Maddie     A Hayes
87 90 92 100 88
 His/her average is: 91.40

 Enter five test scores for student Sterling   R Gillette
84 62 55 30 40
 This student will have to repeat the class.
 His/her average is: 54.20

 Enter five test scores for student Ilsa       L Lazlo
77 50 93 83 99
 His/her average is: 80.40
```

rewind Positions a sequential file before its first record.

FORMATS

rewind [**unit** =] *unitid* {Short form}

rewind ([**unit** =] *unitid*[**,iostat** = *sfield*]][**,err** = *label*]) {Long form}

ARGUMENTS

unit = *unitid* — *Unitid*: specifies the external unit to which the file you're rewinding is connected. The phrase **unit** = is optional if *unitid* is the first or only argument.

iostat = *sfield* — Optional I/O status specifier. If you use **iostat**, the compiler returns a 32-bit FORTRAN or system status code if the **rewind** fails. *Sfield* must be an **integer*4** variable.

err = *label* — Optional error specifier. *Label* is the label of a statement in the same program unit as the **rewind**. If you use **err** and the **rewind** fails, FORTRAN transfers control to *label*.

For more information on these arguments, see the listing for "I/O Attributes" in this encyclopedia.

DESCRIPTION

Rewind explicitly positions a file opened for sequential access at its initial point (before the file's first record). If the file is already positioned at its starting point, **rewind** has no effect. (The **read** and **write** statements position sequential files implicitly—to the end of the record most recently read or written. **Endfile** writes an end-of-file marker after the last record read or written in a sequential file, and positions the file pointer just after the marker. **Backspace** explicitly positions a file before the record most recently read or written.)

In Domain FORTRAN, you cannot use the **rewind** statement on the error output, standard input, and standard output files (preconnected units 0, 5, 6, and 7).

If you **rewind** a file back to its starting point and then **write** to the file, you lose everything beyond the data you just wrote. The following example program illustrates this point.

EXAMPLE

```
*   This program writes data from an input file to the output file
*   new_data.  It then rewinds and writes new data to the output file.
*   Because of the rewind, the original data is lost.

       program rewind_example

       character*26 s_name
       character*50 stuff, new_stuff
       integer*4    open_stat, rewind_stat
       data new_stuff/'Live long and prosper.'/

*   Open the input and output files.
       open (3, file='names_data', iostat=open_stat,
      +      recl=26, status='readonly')
       if (open_stat .ne. 0) stop 'Error opening file'
       open (4, file='new_data', iostat=open_stat,
      +      recl=50, status='new')
       if (open_stat .ne. 0) stop 'Error opening file'

*   Read input file, write its data to output file, and print
*   the contents of the output file.
       read (3, '(A)') s_name
       stuff = s_name
       write (4, '(A)') stuff
       print *, 'File new_data now contains: ', stuff

*   Rewind to the beginning and write new data to the file.  Notice
*   that this rewind statement returns the I/O status specifier.
       rewind (4, iostat=rewind_stat)
       if (rewind_stat .ne. 0) print *, 'Error on rewind'
       write (4, '(A)') new_stuff

*   Rewind to the beginning, read contents of the file into a different
*   variable to show the file really contains what it's supposed to
*   contain, and print the results.
       rewind 4
       read (4, '(A)') stuff
       print *, 'And now, file new_data contains: ', stuff
       close (3)
       close (4, status='delete')

       end
```

USING THIS EXAMPLE

If you execute the program **rewind_example,** this is the output:

```
File new_data now contains: Stewart   MFranklin
And now, file new_data contains: Live long and prosper.
```

Save Saves the values of a program unit's variables in static storage.

FORMAT

save [*name1* [. . . , *nameN*]]

ARGUMENT

name1, nameN The name(s) of variable(s) and array(s) to be allocated static storage. If you declare multiple *names*, separate them with commas.

DESCRIPTION

Save directs the compiler to allocate static (rather than stack) storage to the variables and arrays of the program unit in which the **save** statement appears. This preserves the values of those variables and arrays from one invocation of the program unit to the next.

> **NOTE:** The compiler cannot perform optimizations on any local variables that appear in a **save** statement, nor on any expressions containing such variables. As a result, your program may run more slowly if you use the **save** statement.

The opposite of static storage is dynamic, or stack, storage. A local variable stored on the stack does not retain its value from one invocation of its program unit to the next. Stack storage means that FORTRAN makes a new copy of the variable each time you invoke the program unit. Since stack size is limited, to avoid addressing faults you should use static storage for large arrays. Note, however, that programs generally execute more slowly if they use static, rather than stack, storage.

Save has two formats. The simple format, a blank **save** statement (no arguments), directs the compiler to save all variables and arrays in the program unit. The list format specifies the variables and arrays you want saved. For example, the following fragment includes an example of the list format **save**:

```
character*10 trees(5), flowers(10), weeds(15)
    . . .
save trees, flowers
```

This **save** specifies that you want to save the values in the 5-element array **trees** and the 10-element array **flowers** between subprogram invocations. However, you don't want to save the values in the 15-element array **weeds**.

If you use the list format, separate the variables or array names with commas. You cannot use a name more than once in a program unit's **save** statements, nor can you use the names of dummy arguments, functions, or variables and arrays in **common**.

By default, Domain FORTRAN allocates static storage to **common** block entities. It also allocates static storage to variables you initialize with **data** statements as well as to any variables associated (in an **equivalence** statement) with other variables initialized by **data** statements or saved by **save** statements. Domain FORTRAN allocates stack storage to all other local variables.

Static storage is not automatically initialized to zero when a program is invoked. If your program assumes that the storage will be zero–filled, you could experience run–time errors. There are two ways around this, however. If you *always* want static storage to be initialized to zero, compile with the **–zero** option (described in Section 6.5.39). On the other hand, if you want to take a more selective approach to what is initialized to zero, you can initialize variables with **data** statements.

The **save** statement is especially useful when compiling a program that was developed on another system. Unlike the Domain FORTRAN compiler, some FORTRAN compilers place all data in static storage. A program developed on such a compiler may depend on data preserving its values from one invocation of a program unit to the next. If you know that the program you are porting was developed on such a compiler, the safest way to proceed is to use the compiler's **–save** option (described in Section 6.5.30) or a blank **save** statement in each program unit, ensuring static storage for all variables.

Unfortunately, preserving data in static storage results in programs running more slowly than if data were allocated storage on the stack. (One reason for the degraded performance is that the optimizer cannot assign variables in static storage to registers.) If you are concerned with program performance, you may want to take the time to analyze the program and identify those variables that must be preserved in static storage, and name only them in the **save** statement.

If you have access to a Series 10000 compiler, you can let the compiler do some of the analysis for you by compiling the program with the **–save** option (or with blank **save** statements in every program unit) and **–info** option (refer to Section 6.5.17). The compiler issues diagnostic messages for all scalar variables that must be allocated static storage in order for the program to execute correctly. You need include only these variables (plus any array variables that you have determined must also go in static storage) in the list of arguments for the **save** statements.

EXAMPLE

```
      program save_example

      integer*2 i

      do i=1,3
          call save_test()
      end do

      end

************************************************************************
* The following subroutine uses a save statement to allocate
* static storage for the variable named first_time. Since first_time's
* value is saved each time the subroutine is called, the
* subroutine yields different results on separate calls.

      subroutine save_test()

      logical first_time
      save    first_time
      data first_time /.true./

      if (first_time) then
       print *, 'This is the first time this routine has been called.'
       first_time = .false.
      else
        print *, 'This routine has been called before.'
      endif

      end
```

USING THIS EXAMPLE

This program is available online and is named **save_example**. If you execute the program, this is the output:

```
 This is the first time this routine has been called.
 This routine has been called before.
 This routine has been called before.
```

Stop Terminates the program's execution.

FORMAT

stop [*message*]

ARGUMENT

message — Optional stop message: a string of integers, or a character constant enclosed in single quotes (for example, 'Error occurred').

DESCRIPTION

Stop terminates the program's execution unconditionally, and optionally writes your stop message (*message)* to error output. Whether you supply your own *message* or not, Domain FORTRAN always sends the string 'Fortran STOP' to error output when it executes a **stop**. Note that a **stop** statement causes a change in the the value of **status_$t**, and programs that terminate with **stop** terminate with a warning severity. (Refer to Section 7.8.2 for more details about **status_$t**.)

Stop differs from **end**. **Stop** terminates the execution of a running program, whereas **end** terminates the execution of a running program *and also* indicates the end of a program unit during translation. Every program unit must end with **end**, whereas the **stop** statement is optional. Thus, in the following example the **stop** statement is unnecessary:

```
program bad_stop
    .
    .
    .
stop            {this statement is unnecessary}
end
```

Although you may use as many **stop** statements as you wish in a program unit, you typically use **stop** to terminate a program after an error. For example, if your program relies on a certain number of input values, you might print an error message and then **stop** if it encounters an end-of-file indicator in the input file.

You can use a string of integers as your *message*. This is useful if your program contains several **stops**, because it allows you to keep track of which **stop** the program reached. For example:

```
logical found_file, has_data, no_data
    .
    .
    .
if ((found_file) .and. (has_data)) then
    {process entries}
else if (no_data) then
    stop 1
else                      {Can't find file.}
    stop 2
endif
```

Since you must provide integers if you use numbers in your **stop**, the following is not correct:

```
stop 3.2                  {Illegal!}
```

EXAMPLE

```
* This program takes a user-supplied filename and tries to open the
* file.  Since the open statement specifies that the file status is
* 'old' (that is, that it already exists), if the file doesn't already
* exist, the program issues a stop.

      program stop_example

      character*80 file_name
      integer*4    open_status
*  Ask the user to supply the filename.
      print *, 'Enter the name of an existing file, enclosed in '
      print *, 'single quotes, that you want to open.'
      read *, file_name

*  Try to open the file.  Stop if it doesn't exist.
      open (unit=10, file=file_name, iostat=open_status, status='old')
      if (open_status .ne. 0) then
          stop 'File does not exist'
      else
          print *, 'The file was opened without error'
      end if
      close (10)
      end
```

USING THIS EXAMPLE

This program is available online and is named **stop_example**. Following is a sample run of the program:

```
 Enter the name of an existing file, enclosed in single
 quotes, that you want to open.
[illegible]
Fortran STOP File does not exist
```

Write Transfers data from internal storage to an output file.

FORMAT

write (*clist*) [*iolist*]

ARGUMENTS

(clist) Control list: must be enclosed in parentheses, and can consist of the following:

- **unit** = {*unitid* | *ifid*}. Required unit specifier; indicates the external unit to which the output file is connected, or the file's internal file ID. The phrase **unit** = is optional if this is the first argument in *clist*.
- **fmt** =*fmt*. Required format specifier. You may use an asterisk (*) to denote list-directed formatting. The phrase **fmt** = is optional if this is the second argument in *clist*.
- **rec** = *rec_num*. Record number, required for reading direct access files. If you use this attribute, you cannot specify that you want to read an internal file or a **namelist**.
- **iostat** = *sfield*. Optional I/O status specifier.
- **err** = *label*. Optional error specifier.
- Namelist(s). You can specify the namelist as you would specify a variable name, or you can specify it as **nml** = *namelist*. For example, if you create a namelist called **my_vars**, you can specify it as **my_vars** or as **nml** = **my_vars**. For an example demonstrating a namelist in a **write** statement, see the listing for **namelist** in this encyclopedia.

iolist The data to be transferred; can consist of the following:

- Variable name(s)
- Array or array element name(s)
- Character substring(s)
- Implied **do** list(s)
- Expression(s) (including constants)

For more information about the *clist* options associated with **write**, refer to the listing for "I/O Attributes" in this encyclopedia.

DESCRIPTION

Write transfers data from internal storage to an output file, according to the control list (*clist*) specifications.

The *iolist* specifies the data, which can consist of variables, arrays or array elements, substring names, or expressions that may include constants. If you include an array name without a subscript, FORTRAN writes all of the array's elements in the order in which they are stored.

FORTRAN always tries to write the data corresponding to all *iolist* entities. When it encounters a slash (/) in the format specification, FORTRAN writes the next record, using the repeatable edit descriptors to the right of the slash for the remaining *iolist* entities. If the repeatable edit descriptors in the format specification have been used up and there are more *iolist* entities, FORTRAN performs *format reversion*, as follows:

- If there are nested parentheses in the format specification, FORTRAN reverts to the format terminated by the last preceding right parenthesis.
- If there are no nested parentheses, FORTRAN reverts to the beginning of the format specification.

When FORTRAN reverts to a parenthesis preceded by a repeat count, it uses the repeat count. The reused portion of the format specification must contain at least one repeatable edit descriptor. For more information about format reversion and edit descriptors, refer to the listing for the **format** in this encyclopedia.

An *iolist* can also contain one or more implied **do** loops, which you typically use to write arrays. For example:

```
      write (unit=2,fmt=25) (grades_data(i), i = 1,10)
 25   format (5I3)
```

This statement writes 10 elements of array **grades_data**, beginning with the first element, to the output file connected to unit 2. It uses the format at statement 25, which says to write five array elements on each line. Refer to the listing for **do** in this encyclopedia for more information about **do** loops.

Control List Arguments

Refer to the listing for "I/O Attributes" in this encyclopedia for more information on the control list (*clist*) arguments.

EXAMPLE

```
*  This program first prints out data it contains, requests
*  new text data from the user, and enters the new data into
*  a file it opened.

      program write_example

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/fio.ins.ftn'
%include '/sys/ins/error.ins.ftn'
      character*7  form
      character*50 in_data
      integer*4    write_stat
      real*4  first, second, third
      data    first, second, third /55.32, 897.01, 9.67/
      form = '(3F7.2)'

*  Write statement using list-directed formatting.
      write (*,*) 'The numbers already in the file are: '

*  Write statement using character variable in format specification.
      write (*, form) first, second, third
      open(unit=2, file='junk', recl=50, status='overwrite')
      write (*,*) 'Enter text you want written to the new file:'
      read (*, '(A)') in_data

*  Write with a unit id specification.  This write says to write to
*  the file connected to unit 2, using the format specification at
*  the statement labeled 10.  It also returns the I/O status
*  specifier into write_stat.
10    format (A50)
      write (2, 10, iostat=write_stat) in_data
      if (write_stat .ne. 0) then
      call error_$print(write_stat)
      endif
      close (2)
      end
```

USING THIS EXAMPLE

This program is available online and is named **write_example**. Following is a sample run of the program:

```
 The numbers already in the file are:
  55.32 897.01    9.67
 Enter text you want written to the new file:
Here's looking at you, kid!
```

Chapter 5

Subprograms

This chapter explains how to declare and call subroutines and functions. It also describes statement function statements, intrinsic functions, and block–data subprograms. The term "subprogram" refers to any of these entities. The chapter concludes with a discussion of subprogram arguments and recursive subprograms.

NOTE: If you compile your program using either the **f77** command or the **ftn** command with the **–uc** option, the compiler appends an underscore(_) to any subprogram name that does not start or end in an underscore and does not contain a dollar sign ($). The underscore distinguishes the subprogram from a C or external variable with the same user–assigned name. Refer to subsection 6.5.34 for more information about the **–uc** option.

5.1 Subroutines

The syntax for a FORTRAN subroutine heading is as follows:

subroutine *name* [([*argument_list*])]

where:

name	is the name of the subroutine.
argument_list	is one or more dummy arguments, each of which corresponds to an actual argument in the calling program unit, or one or more asterisks (*) to indicate alternate return(s).

A FORTRAN subroutine must begin with a **subroutine** statement and end with an **end** statement. It must not contain a **block data, function,** or **program** statement or another **subroutine** statement. It may contain any other statement, including one or more **returns**.

A **call** statement in another program unit (the "calling" program unit) invokes a subroutine and, optionally, passes it one or more actual arguments.

You must include a dummy argument in the **subroutine** statement for each actual argument the calling program unit passes. If there are no dummy arguments, you may place an empty pair of parentheses after the subroutine's name in the **subroutine** statement, although this is optional.

A subroutine's dummy arguments cannot appear in **data, equivalence, parameter, intrinsic, save,** or **common** statements in the subroutine. However, a **common** block may have the same name as a dummy argument.

Within a subroutine, you may declare local variables (in addition to dummy arguments) just as you would declare them in a main program. For instance:

```
subroutine my_sub(dummy1, dummy2)
character*(*) dummy1, dummy2
integer*2     count                  {Count and real_num are}
real*4        real_num               {local variables.      }
```

You can include one or more **entry** statements in a subroutine to define alternate entry points. Whenever the calling program unit calls a particular entry point, the subroutine starts executing from that point. See the listing for **entry** in Chapter 4 for more information.

A **return** or **end** statement terminates a subroutine's execution, breaks the association between the actual and dummy arguments, and returns control to the caller. Execution resumes with the statement immediately after the **call** statement, or, if you use an alternate return, with the statement associated with the alternate return. For more information about alternate returns and a sample program using them, see the listing for **return in** Chapter 4.

By default, FORTRAN allocates dynamic (stack) storage to local variables, which means the variables do not retain their values from one invocation of the program unit to the next. However, you can use the **save** statement to allocate static storage for certain variables or you can compile with the **–save** switch to get the same results. Refer to the listing for **save** in Chapter 4 for information about the **save** statement and to Section 6.5.30 for the **–save** option.

Example

```
*  This program finds what number day (out of 365) a user-supplied date
*  is in a year.  Leap years are ignored.

      program subr_example

      integer*2 month_num, day, tot_days, m(12)
      character answer

      data answer, (m(month_num), month_num=1,12)
     +     /'y', 31,28,31,30,31,30,31,31,30,31,30,31/

      do while ((answer .eq. 'y') .or. (answer .eq. 'Y'))
          call validate(m, month_num, day)
          call compute_days(m, month_num, day, tot_days)
          print 10, tot_days
10        format ('The date you entered is number ',I3,' of the year.')
          print 20
20        format (/, 'Again? (Y or N) ', $)
          read (*, '(A1)') answer
      end do
      end

************************************************************************
* This subroutine asks for a month and day and then checks to see if
* they are valid. If one or both are not, it prints an error message and
* reprompts for input.

      subroutine validate(m, month_num, day)
      integer*2 m(12), month_num, day

      logical bad_month, bad_day
      data bad_month, bad_day /.false., .false./

100   print 110
110   format ('Enter the month and day separated by a space: ', $)
      read (*, *) month_num, day

*  Check that the month and day values entered are in the valid range.

      if ((month_num .lt. 1) .or. (month_num .gt. 12)) bad_month=.true.
      if (day .gt. m(month_num)) bad_day = .true.

      if (bad_month .or. bad_day) then
         print 120
120      format ('You entered an invalid date. Try again.')
         bad_month = .false.
         bad_day = .false.
         goto 100
      end if
      end
```

```
************************************************************************
*  This subroutine takes the user-entered date and computes which
*  number day it is.  Notice that these dummy arguments do not have
*  the same names as the actual arguments.

      subroutine compute_days(months, n, d, total)
      integer*2  months(12), n, d, total, i
      total = 0
      i = 1
      do while (i .lt. n)
          total = total + months(i)
          i = i + 1
      end do
      total = total + d
      end
```

Using This Example

This program is available online and is named **subr_example**. Following is a sample run of the program.

```
Enter the month and day separated by a space: 7 13
The date you entered is number 194 of the year.

Again? (Y or N) y
Enter the month and day separated by a space: 2 30
You entered an invalid date. Try again.
Enter the month and day separated by a space: 2 28
The date you entered is number  59 of the year.

Again? (Y or N) n
```

5.2 Functions

A FORTRAN function must begin with a heading following this syntax

[*data_type*] **function** *name* ([*argument_list*])

where:

data_type — is the data type of the function subprogram: that is, **byte**, **integer**, **integer*2**, **integer*4**, **real**, **real*4**, **real*8**, **double precision**, **complex**, **complex*8**, **double complex**, **complex*16**, **logical**, or **character***len*, where *len* is the length of the character string the function will produce.

name — is the name of the function subprogram.

argument_list — is one or more dummy arguments, each of which corresponds to an actual argument in the calling program unit.

A FORTRAN function must begin with a **function** statement and end with an **end** statement. It must not contain a **block data**, **function**, or **program** statement or another **function** statement. It may contain any other statement, including one or more **returns**.

You must put a pair of parentheses after the function *name*, even if your *argument_list* is empty.

5.2.1 Giving a Function a Data Type

The name you give a FORTRAN function determines what data type the function is *unless* you use the optional *data_type* argument to explicitly define the data type. For example:

- This function is of type **real** because of FORTRAN's default naming conventions.

  ```
  function return_real (real_num, x)
  ```

- This function is explicitly defined to be of type **logical**.

  ```
  logical function truth (yes, no)
  ```

5.2.2 Differences from Subroutines

Function subprograms behave just like subroutine subprograms, with the following exceptions:

- A calling program unit does not invoke a function subprogram with a **call** statement; it simply refers to the function's name, passing whatever actual arguments are necessary to the function's corresponding dummy arguments. The calling program unit must declare the function as an **external** if it uses the function's name as an actual argument.

- During execution of the function, its name acquires a value, which then is passed to the calling program unit. The function name must have a data type for this reason. The calling program unit, in turn, uses the function name just like a variable.

Because the calling program unit uses a function name just like a variable, the program unit needs to know the function's data type. You can let FORTRAN's naming conventions or an **implicit** statement determine the data type, or you can tell the calling unit the data type with an explicit type declaration. For instance:

```
real*8 x, avg_temperature
real*4 daily_temps
    .
    .
    .
x = avg_temperature(daily_temps){Invoke function avg_temperature.}
    .
    .
    .
real*8 function avg_temperature(temps)   {Function heading.}
```

This fragment explicitly declares the function name **avg_temperature** as having the **real*8** data type. It is not enough to state the data type in the **function** heading (although you must do it there, too) and omit the declaration in the calling program unit. If you do omit the first declaration, FORTRAN uses the default naming conventions and assumes that the function is of type **real*4**. Your run-time results will be unpredictable.

5.2.3 Function Body

As with subroutines, a function's dummy arguments cannot appear in **data, equivalence, parameter, intrinsic, save,** or **common** statements within the function. However, a **common** block can have the same name as a dummy argument.

Within a function, you may declare local variables (in addition to dummy arguments) just as you would declare them in a main program. For instance:

```
logical function my_func (dummy1, dummy2)
character*(*) dummy1, dummy2
integer*2     count                     {Count and real_num are}
real*4        real_num                  {local variables.      }
```

Note that the lengths of **dummy1** and **dummy2** are undetermined, as indicated by the second asterisk (*). Refer to Section 5.7.1 for more information on characters as arguments.

You can use one or more **entry** statements in a function subprogram to define alternate entry points. Whenever the calling program refers to that entry point, the function will start executing from that point.

If the function name is of **character** data type, each **entry** name within that function must also be of **character** data type and must have the same length specification as that of the function. If the function name is not of **character** data type, then the **entry** name can be of any data type *except* the **character** data type. However, if you invoke the function by its **entry** name but access the return value that was assigned to the function name, you should be sure that both **entry** name and the function name are of the same data type. For an example of a program that uses **entry** in a function, refer to the listing for **entry** in Chapter 4.

By default, FORTRAN allocates dynamic (stack) storage to local variables, including those used in functions. When variables are stored on the stack, they do not retain their values from one invocation of the program unit to the next. However, you can use the **save** statement or compile with the **-save** option to allocate static storage for certain variables. Refer to the listing for **save** in Chapter 4 for a description of the **save** statement. Section 6.5.30 describes the **-save** option.

Example

```
*  This example uses the integer function scalar_product to compute
*  and print the scalar product of two arrays.

       program function_example

*  Declare variables.  Notice that the name of the function
*  (scalar_product) is declared as an integer*4 variable.  If it
*  were not explicitly declared, FORTRAN would assume the function
*  was supposed to return a real result--regardless of the fact
*  that the function heading states that it is an integer function.

       integer*2   ARRAY_SIZE, i
       parameter  (ARRAY_SIZE = 10)
       integer*4   one_array(ARRAY_SIZE), two_array(ARRAY_SIZE)
       integer*4   scalar_product, result

*  Load data into the two arrays.  The second array gets values
*  equal to the squares of the values in the first array.

       do i = 1, ARRAY_SIZE
           one_array(i) = i
           two_array(i) = i**2
       enddo
       print *, 'The arrays contain the following: '
       write (*,*) (one_array(i), i=1,ARRAY_SIZE)
       write (*,*) (two_array(i), i=1,ARRAY_SIZE)

 *  Invoke the function.
       result = scalar_product(one_array, two_array, ARRAY_SIZE)
       print *, 'The scalar product of the two arrays is: ', result

       end

**********************************************************************
*  This function finds the scalar product of two linear arrays.  A
*  scalar product is the sum of the products of corresponding
*  elements in the two arrays.

*  Declare the function's name and its dummy arguments.

       integer function scalar_product (a, b, SIZE)
       integer*2 SIZE, j
       integer*4 a(SIZE), b(SIZE)

       scalar_product = 0
       do j = 1, SIZE
           scalar_product = scalar_product + (a(j) * b(j))
       enddo
       return
       end
```

Using This Example

This program is available online and is named **function_example**. If you run the program, you get the following results:

```
The arrays contain the following:
1 2 3 4 5 6 7 8 9 10
1 4 9 16 25 36 49 64 81 100
The scalar product of the two arrays is:  3025
```

5.3 Statement Function Statements

A statement function statement is a function that consists of a single unlabeled Domain FORTRAN statement. In effect, it is an expression to which you assign a name, and to which you can pass arguments. It takes this format:

name [*(argument_list)*] = *exp*

where:

name	is the name of the statement function.
argument_list	is one or more dummy arguments, each of which must be variables and each of which corresponds to an actual argument.
exp	is any valid FORTRAN expression. (The expression may contain intrinsic functions, function subprograms, or other statement functions previously defined in the program unit.)

For example, the following valid statement function computes the circumference of a circle:

```
circumference(radius) = 2 * 3.14159 * radius
```

A statement function is a nonexecutable statement. Because of that, its definition must appear before all executable statements in a program unit. Once you've defined the statement function, you can refer to it just as you would a function subprogram as long as the statement function and the reference to it are in the same program unit.

Just like a function name, the name of a statement function acquires a value, and therefore has a particular data type. You can rely either on FORTRAN's default naming conventions or on an **implicit** statement to determine the data type, or you can explicitly define the statement function's data type with a type declaration.

Although the dummy arguments in a statement function's *argument_list* all must be variables, their corresponding actual arguments may be variables, constants, or expressions.

NOTE: If the variable to which you refer in a statement function is declared in the routine in which the statement function appears, the compiler inhibits some optimizations on the referenced variable. As a result, your program may run more slowly.

Example

```
*   This program uses two statement functions that calculate the
*   hypotenuse and area of a right triangle when a user supplies the
*   lengths of two sides.

       program statement_functions_example

       intrinsic sqrt
       real side1, side2, hypot, area  {Includes data type declaration }
                                       {of the two statement functions.}

       character answer
       logical   again
       data      again /.true./

*   Define the two statement functions.

       hypot(a_side, b_side) = sqrt(a_side**2 + b_side**2)
       area(a_side, b_side) = a_side*b_side/2.0

       do while (again)
          print 10
10        format (' Enter lengths for two sides of the triangle: ', $)
          read *, side1, side2

*   Invoke the statement functions and print the results.
          print *, 'The triangle''s hypotenuse is: ',hypot(side1,side2)
          print *, 'And its area is: ', area(side1,side2)

          print 20
20        format (/, ' Again? (Y or N) ', $)
          read (*, '(A)') answer
          if ((answer .eq. 'N') .or. (answer .eq. 'n')) again = .false.
       enddo

       end
```

Using This Example

This program is available online and is named **statement_functions_example.** Following is a sample run of the program.

```
Enter lengths for two sides of the triangle: 6 9
The triangle's hypotenuse is:  10.81665
And its area is:  27.00000

Again? (Y or N) y
Enter lengths for two sides of the triangle: 3 4
The triangle's hypotenuse is:  5.000000
And its area is:  6.000000

Again? (Y or N) N
```

5.4 Intrinsic Functions

Like standard FORTRAN, Domain FORTRAN supports a variety of built-in intrinsic functions. Most of these functions perform standard mathematical operations such as finding sine or cosine or computing a square root. Other functions convert values of one data type to another (for example, convert an integer to a real). Appendix C lists all the available intrinsic functions.

In addition to the standard intrinsic functions, Domain FORTRAN offers the following:

- **int2** and **int4** for forcing a number of any type to an integer*2 or integer*4 value
- The **and**, **or**, **xor**, **not**, **rshft**, and **lshft** functions for manipulating bits in 2-byte and 4-byte integers.
- The **addr** function for returning the address of a variable, array, or subprogram name

When you use an intrinsic function in a program, you can designate it as such with the keyword **intrinsic**. See the listing for **intrinsic** in Chapter 4 for more information and a sample program using intrinsic functions.

5.5 Block–Data Subprograms

A block–data subprogram is a special subprogram that you can use only to assign values to variables in one or more **common** blocks. Its heading takes this format

block data *name*

where

name is the name of this block data subprogram.

The **block data** statement must be the first statement in a block–data subprogram and **end** must be the last. A main program or subroutine or function cannot contain a **block data** statement.

A block–data subprogram cannot contain executable statements, because its only use is to initialize values in **common** blocks. A block data subprogram may contain data type declaration statements, or **implicit**, **parameter**, **dimension**, **common**, **equivalence**, **save**, or **data** statements.

If a variable will not default to the data type you want, you *must* make an explicit type declaration for it within the block–data subprogram. The sample program that follows demonstrates explicit typing.

Do not initialize the same variable twice with a **common** block. Make sure the named **common** blocks cited in the block–data subprogram have the same lengths as they have in other program units; otherwise, unpredictable results will occur.

Example

```
*  This simple program uses a block data subprogram to initialize a
*  variety of variables.  The main program then prints out those values.

       program blockdata_example

*Declare variables and specify those that are to go in the common block.

       integer*2 i
       real*8  pi
       integer*4  nums(10)
       logical the_truth
       character*15 names(5)
       common /jumble/ pi, nums, the_truth, names

       write (*,*) 'The variable pi contains: ', pi
       write (*,*) 'The array nums contains:', (nums(i), i=1,10)
```

```
      write (*,*) 'Variable the_truth equals: ', the_truth
      do i=1,5
         write (*, 10) i, names(i)
10       format (' Entry ', I1, ' of the names array is: ', A15)
      enddo

      end

**********************************************************************
*  This block data subprogram initializes the values in the common
*  block, jumble.

      block data init_vals

      real*8  pi
      integer*4  nums(10)
      logical the_truth
      character*15 names(5)
      common /jumble/ pi, nums, the_truth, names
      data pi, (nums(i), i=1,10), the_truth /3.14159, 10*0, .true./
      data names(1), names(2), names(3), names(4), names(5)
     +     /'Scarlett', 'Rhett', 'Ashley', 'Melanie', 'Aunt Pittypat'/

      end       {All block data subprograms must end}
                {with an 'end' statement.           }
```

Using This Example

This program is available online and is named **blockdata_example**. If you run the program, you get the following output.

```
 The variable pi contains:  3.141590000000000
 The array nums contains: 0 0 0 0 0 0 0 0 0 0
 Variable the_truth equals:  T
 Entry 1 of the names array is: Scarlett
 Entry 2 of the names array is: Rhett
 Entry 3 of the names array is: Ashley
 Entry 4 of the names array is: Melanie
 Entry 5 of the names array is: Aunt Pittypat
```

5.6 Argument Lists

You can declare a subprogram with or without arguments. If you declare a subprogram without arguments, you can only pass it data that is declared in **common** blocks. If you declare a subprogram with arguments, you are specifying the data type of each argument that can be passed to it.

You specify arguments within an argument list. An argument list can have a maximum of 511 arguments and has the following format:

(argument1, . . . argumentN)

After listing argument names in a subprogram heading, you can declare them within the subprogram itself. Indeed, you *must* declare them if, under FORTRAN's naming conventions, they will default to a type other than that of their corresponding arguments in a subprogram call. The following examples show several subprogram headings with a variety of argument types in the ***argument_list*** and with argument data types declared within the subprogram itself:

```
*  Declare a subroutine with no argument list.
       subroutine simple
```

```
 *  Declare a subroutine with an argument list that has two arguments.
       subroutine con(a, b)
       integer*4 a
       real      b          {Since b would default to this data}
                            {type, this explicit declaration is}
                            {not absolutely necessary.         }
```

```
 *  Declare an integer function with two arguments of the same type.
       integer function amigo(x,y)
       logical x, y
```

```
 * Declare a function with an argument list that has three arguments.
       function big(quart, volume, cost)
       integer*2 quart
       real*8    volume
       real*4    cost
```

5.7 Actual Arguments and Dummy Arguments

When a calling routine invokes a subprogram, the routine usually passes values that the subprogram uses to compute results. These values are called **actual arguments.** The arguments listed in a subprogram declaration, such as those shown in the previous section, are called **dummy arguments**.

When you invoke a subprogram and control is transferred to it, actual arguments are associated with dummy arguments. This means that each dummy argument takes the value of its corresponding actual argument. If you assign a new value to a dummy argument within the subprogram, that new value also is assigned to the corresponding actual argument.

Because of the association between actual and dummy arguments, such arguments must agree in data type, order, number, and size. For example, if a subprogram call lists two **real*4**s followed by a **logical** as its actual arguments, the subprogram heading must have two **real*4**s followed by a **logical**—*in that exact order*—listed as its dummy arguments. If the actual arguments differ from the dummy arguments in data type, order, number, or size, unpredictable results will occur.

Corresponding arguments are not required to have the same name, although if they do it may be easier for you to remember what corresponds to what. For example, the following subroutine call and subroutine heading show corresponding arguments with the same and different names:

```
call correspond (int_num, prog_num)
         .
         .
         .
subroutine correspond (int_num, subr_num)
```

Actual arguments can be constants, variables, expressions, arrays, character strings, subprogram names, or alternate return indicators. Dummy arguments can include any of those except constants, expressions, or specific array elements. That means, for instance, that you cannot list an individual array element—for example, **my_array(3)**—as a dummy argument, although you can list it as an actual argument.

5.7.1 Characters as Arguments

If your subprogram heading includes a **character** dummy argument, the argument's length must not be greater than the length of its corresponding actual argument. It can, however, be shorter, or even undetermined. To indicate that a **character** argument's length is undetermined, use an asterisk (*) for the length designation. For instance, if you define a **character** argument this way

```
subroutine sayings(a_proverb)
character*(*) a_proverb
```

its length is set to the length of its corresponding actual argument. So if your code includes the following

```
character*31 murphys_law
character*17 too_fast

murphys_law = 'Anything that can go wrong will'
too_fast = 'Haste makes waste'
call sayings(murphys_law)
         .
         .
         .
call sayings(too_fast)
```

a_proverb is 31 bytes long when **murphys_law** is its corresponding actual argument, and **a_proverb** is 17 bytes long when **too_fast** is its actual argument.

5.7.2 Arrays as Arguments

As with **characters**, a dummy array argument may not be larger than its corresponding actual array. However, the dummy's size may be determined by the values it gets from its actual arguments or from **common** variables. In such a case, the dummy argument is called an **adjustable** array. Consider this sample function that totals up the elements in a **real** array:

```
real function total(matrix, int1, int2)
integer*2 i,j
real      matrix(int1, int2)

total = 0.0
do i = 1, int1
  do j = 1, int2
    total = total + matrix(i,j)
  enddo
enddo
end
```

The values of **int1** and **int2** determine the size of the 2-dimensional array **matrix**. Those values can vary widely, as these sample function calls show:

```
result = total(matrix,2,3)
result = total(matrix,100,50)
result = total(matrix,3,1500)
```

In addition to regular dummy arrays and adjustable dummy arrays, you can also pass an **assumed-size** dummy array. An assumed-size dummy array is one for which the upper boundary of the last dimension is listed as an asterisk(*). For example:

```
subroutine assume(int_array, one_bound)
integer*4 int_array(1:one_bound, 1:*)
```

Domain FORTRAN determines the size of the assumed-size array this way:

- If the dummy array's corresponding actual argument is an array that is *not* of type **character**, the size of the dummy array is the size of the actual argument array.

- If the dummy array's corresponding actual argument is the *n*th element of an array that is not of type **character** and the actual array is defined as being of size *s*, the size of the dummy array is *s* + 1 – *n*. For example:

```
real dewey_decimal_nums(10000)                  {s = 10000}
          .
          .
          .
call find_category(dewey_decimal_nums(2300))    {n = 2300}
          .
          .
          .
subroutine find_category(dec_num)
real dec_num(1:*)
```

  In this case, the size is 10000 + 1 – 2300, or 7701. The first element of the dummy array **dec_num** corresponds to the 2300th element of **dewey_decimal_nums**.

- If the dummy array's corresponding actual argument is a **character** array, element, or element substring, and the **character** entity begins at the *m*th byte of an array that has *n* bytes, the dummy array's size is **int**((*n* + 1 – *m*) / *chars*), where *chars* is the length of an individual element in the dummy array. For example:

```
   character*10 names(15)              {n = 10*15 = 150}
          .
          .
          .
*  This call is to the subroutine sub, and the actual argument is the
*  5th element of the character array names.

   call sub(names(5))                  {m = ((5*10)-10)+1 = 41}
          .
          .
          .
   subroutine sub(x)
   character*10 x(1:*)                 {chars = 10}
```

  In this case, the size is **int**((150 + 1 – 41) / 10), or 11.

5.7.3 Constants as Arguments

When you pass integer constants as actual arguments to a subprogram, you should be aware that each constant is allocated four bytes of storage by default. This is true whether or not the integer value will fit in two bytes. If you want the constant to be stored in two bytes, compile with the **-i*2** switch (described in subsection 6.5.14). However, if the constant will not fit in two bytes, Domain FORTRAN allocates four bytes for it, even if you compile with **-i*2**.

You should never attempt to change the value of a dummy argument whose actual argument is a constant, since this can lead to unpredictable results. For example, the following incorrect example tries to alter the value of the **real*4** constant **normal_temp**, which has been assigned a value of 98.6 in a **parameter** statement.

```
      program goof          {This program is WRONG!}
      real*4 normal_temp
      parameter (normal_temp=98.6)

      call oops (normal_temp)
      print *, normal_temp
      end

      subroutine oops(celsius_normal_temp)
      real*4 celsius_normal_temp
      celsius_normal_temp = 37.0
      end
```

If you try to run a program like this, it terminates with an access violation.

5.8 Recursion—Extension

A recursive subprogram is a subprogram that calls itself. As an extension to standard FORTRAN, Domain FORTRAN supports recursive subprograms. The following example demonstrates a recursive method for calculating factorials:

```
*  This program uses a recursive function, fact, to calculate the
*  factorial of a user-entered positive integer.

      program recursion_example
      integer*2 in_num
      integer*4 fact, result

      print 10
10    format ('Enter an integer between 1 and 10: ', $)
      read (*, '(I2)') in_num

*  Invoke the function and on return, print the result.
      result = fact(in_num)
```

```
      print 20, in_num, result
20    format (I2, ' factorial is ', I8)
      end

************************************************************************
      integer*4 function fact(i)
      integer*2 i
      if (i .le. 1) then
          fact = 1
      else
          fact = i * fact(i-1)          {Function fact calls itself.}
      endif
      end
```

Using This Example

This program is available online and is named **recursion_example**. If you run the program, you get the following output:

```
Enter an integer between 1 and 10: 8
 8 factorial is    40320
```

Chapter 6

Program Development

This chapter describes how to produce an executable file (that is, a finished program) from Domain FORTRAN source code. There are three Domain environments in which you can develop programs: Aegis, SysV, and BSD. Where the development process differs from one environment to another, we describe each environment separately; otherwise, you can assume the process is the same for all three environments. If there are no differences between SysV and BSD, we refer to them as UNIX environments. Figure 6-1 illustrates the general program development process.

Briefly, you create an executable file in the following steps:

1. Compile each file of source code that makes up the program. The compiler creates one object file for each file of source code. If your program consists of only one source file, this step results in an executable file.

2. Link (bind) the object files if necessary. Linking is necessary if your program consists of more than one object file. The linker resolves external references; that is, it connects the different object files and subprograms so that they can communicate with one another. Before linking, you may wish to package related object files into an archive or library file.

In addition to the above two-step process, there are a number of tools that help you to create, debug, and maintain programs. We describe these tools in Section 6.10.

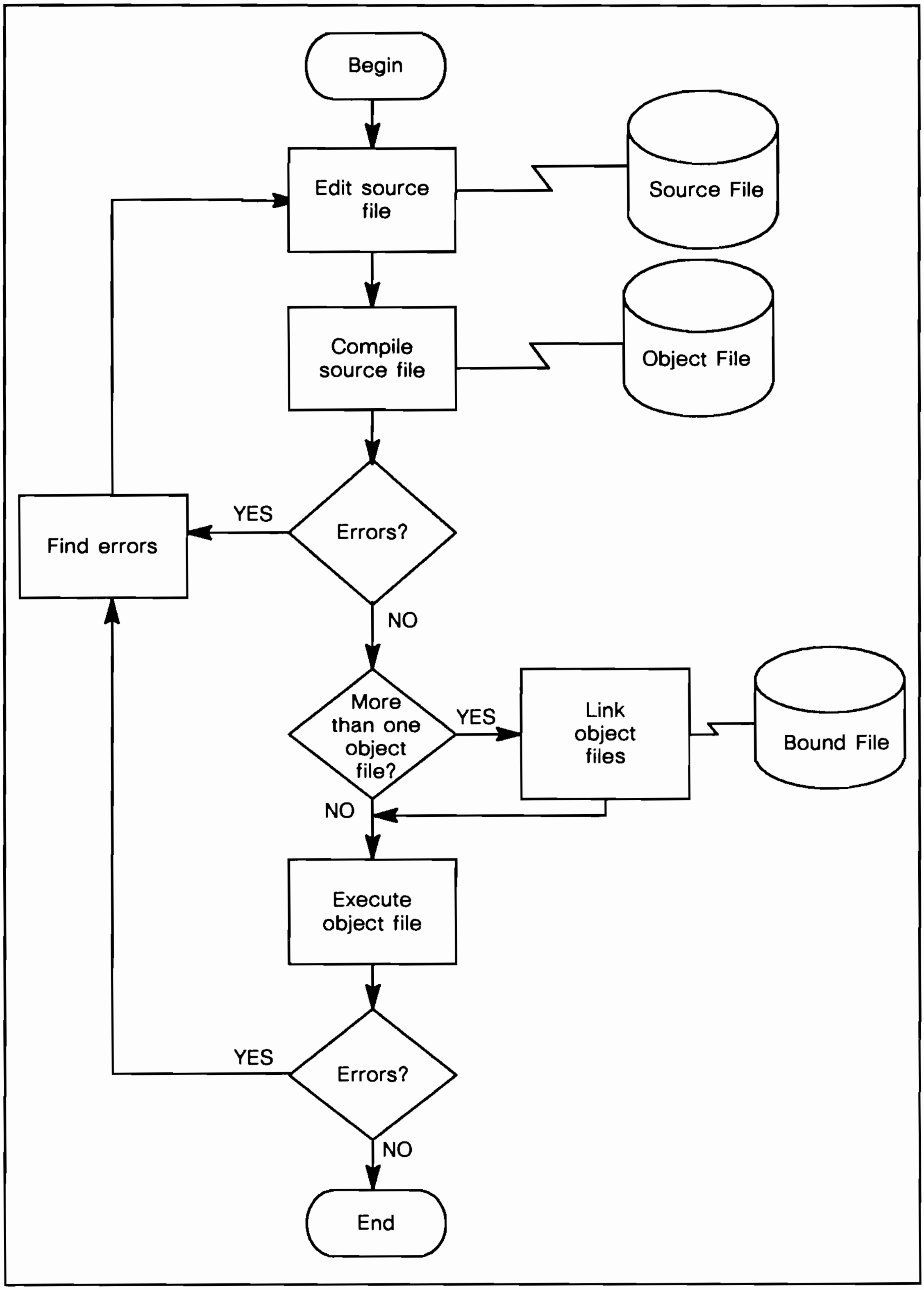

Figure 6-1. Steps in FORTRAN Program Development

6.1 Program Development in a Domain Environment

You invoke the Domain FORTRAN compiler with either one of two commands: **f77** or **ftn**. Typically, you use **f77** to invoke FORTRAN in UNIX environments, and you use **ftn** to invoke FORTRAN in the Aegis environment.

6.1.1 Development Using f77

To create an executable object from one or more files of source code using the **f77** command, you need to compile the source files. The compiler creates one object file for each source code file. The **f77** command allows you to compile more than one source file at a time. Moreover, after it compiles source files, **f77** automatically invokes **ld**, the link editor.

Note that, unlike some object modules produced by **ftn**, all object files produced by **f77** go through a linking stage before they are executable. This is true even if the source file does not refer to external symbols.

6.1.2 Development Using ftn

The **ftn** command compiles only one source file at a time. If your program contains more than one module, then you must link the object files together with the **bind** or **ld** command. These two commands perform similar operations—**bind** invokes the Aegis binder; **ld** is a UNIX link editor. You can use either command to link object modules together.

If your program consists of a single module you do not need to invoke a linker. Object files produced by **ftn** that do not refer to external symbols can be loaded and executed without going through a linking stage.

6.2 Compiler Variants

Apollo produces four variants of the Domain FORTRAN compiler. The four differ in the kind of machine they run on and the kind of machine for which they generate code (target machine). The variants are:

- MC680x0 native compiler
- Series 10000 (*PRISM*) native compiler
- MC680x0-to-*PRISM* cross compiler
- *PRISM*-to-MC680x0 cross compiler

You have at least one of these compilers on your system, but you may be able to use one of the others by means of links to other systems. To find out which one you have, use the **-version** option of the **ftn** command. It will display one of the following codes:

68K	MC680x0 native compiler
PRISM	Series 10000 (*PRISM*) native compiler
68K=>PRISM	MC680x0-to-*PRISM* cross compiler
PRISM=>68K	*PRISM*-to-M680x0 cross compiler

Where the compilers differ from one variant to another, we describe each separately; otherwise, you can assume that all four behave in the same way.

6.3 Compiling

This section tells you how to compile your Domain FORTRAN source code files. Specifically, it describes:

- How to compile using **f77** in a UNIX shell, or **/usr/bin/f77** in an Aegis shell
- How to compile using **ftn** in an Aegis shell, or **/usr/apollo/lib/ftn** in a UNIX shell
- The options available when you use **f77**
- The options available when you use **ftn**

6.3.1 Compiling with f77

In a SysV or BSD shell, you can use the following command to compile one or more files of Domain FORTRAN source code:

$ f77 [*option ...*] *source_pathname* [*source_pathname_n...*]

You can compile using **f77** from an Aegis shell by specifying the full pathname for f77 as follows:

$ /bsd4.3/usr/bin/f77 [*option ...*] *source_pathname* [*source_pathname_n...*]

$ /sys5.3/usr/bin/f77 [*option ...*] *source_pathname* [*source_pathname_n...*]

where *source_pathname* can be either of the following:

- The pathname of a source file you want to compile
- The pathname of an object file that you want to link to the other files you specify

NOTE: If you use the **f77** command in an Aegis shell, you must select either the BSD or SysV environment in order to specify the filenames and options that are applicable. The simplest way to do this is to designate the environment in the full pathname for the **f77** command as shown in the preceding examples. However, you can also indicate your selection by changing the value of the **systype** variable.

Table 6-1 shows the types of files that you can compile using **f77**, the filename suffix associated with each type, sample input and output filenames, as well as what **f77** does with each type of file. *Note that the filename must have a recognized suffix to be processed.* If you want to use **f77** to compile source code in files with names ending with the **.ftn** suffix, you must rename those files.

Table 6-1. Sample Filenames Used with **f77**

Filename Suffix	Type of Source File	Input Filename	Output Filename	What f77 Does with Input File
.f	FORTRAN	sample.f	sample.o	Compiles
.F	FORTRAN	sample.F	sample.o	Processed by C preprocessor; then compiles
.r	Ratfor	sample.r	sample.o	Transformed by Ratfor preprocessor, then compiles
.c	C	sample.c	sample.o	Compiles
.o	Object file	sample.o	a.out	Links

f77 does the following:

- If any of the source files end with a **.F** suffix, **f77** hands off these files to the C preprocessor (cpp) before compiling.
- Compiles all source files (**.f**, **.F**, **.r**, and **.c**) and produces an object file (**.o**) for each source file.

- If **f77** finds a flag that it does not recognize or support, it passes the flag to the link editor.

- Invokes the link editor (**ld**).

  - **ld** checks whether any **.o** files contain an entry point—that is, a program statement. If one or more **.o** files contain an entry point, **ld** attempts to link all the object modules together, including any **.o** files specified on the command line.

  - If **ld** is *successful*, it produces an executable file, which by default it names **a.out**. It also deletes all of the **.o** files produced by this invocation of **f77**.

  - If **ld** is *unsuccessful*, it leaves the **.o** files unchanged. **ld** could be unsuccessful either because of unresolved globals or a missing program statement.

For information about the link editor (**ld**), refer to subsection 6.6.1.

6.3.2 Compiling with ftn

You can compile a file of Domain FORTRAN source code by entering a command of the following format in any Aegis shell:

$ ftn *source_pathname* [*option* ...]

You can compile using **ftn** from a UNIX shell by specifying the full pathname for **ftn** as follows:

$ /usr/apollo/lib/ftn *source_pathname* [*option* ...]

source_pathname is the pathname of the source file you want to compile. You can compile only one source file at a time. In order to simplify your search for FORTRAN source programs, we recommend that *source_pathname* end with a **.ftn** suffix, but you need not include the suffix with the pathname in the compile command line.

Your **ftn** command line can contain one or more of the options listed in Section 6.5. You can use these options with **/usr/apollo/lib/ftn** in a UNIX shell, as well as with **ftn** in an Aegis shell. Note that you cannot abbreviate the options.

For example, consider the following three sample compile command lines, each of which compiles source code file **circles.ftn**:

$

$

$ ftn circles -map -exp -cond -cpu mathlib_sr10

If there are no errors in the source code and the compilation proceeds normally, the compiler creates an object file in your current working directory.

The compiler names the object files it creates according to the following rules:

- If your *source_pathname* ends with **.ftn**, the compiler replaces that suffix with **.bin**.
- If your *source_pathname* does not end with **.ftn**, then by default Domain FORTRAN gives the object file the same pathname as the source pathname, but with the **.bin** suffix.
- If you want the object file to have a nondefault name, use the compiler option **-b** *pathname* or **-bx** *pathname*.

If you want the compiler to create a listing file in addition to an object file, use the **-l** *pathname* option. Listing files contain both the source code and any error, warning or information messages issued by the compiler. Error and warning messages are also reported in **errout**, which by default is the transcript pad. (See the *Aegis Command Reference* to learn how to change this default.)

Table 6-2 shows examples for each of these rules.

Table 6-2. Sample Filenames Used with **ftn**

	Source Code	Command	Object Filename

Source code file named with .ftn suffix	extratext.ftn	$ ftn extratest
		or
		$ ftn extratest.ftn
			extratest.bin

Source code file named with other suffix	test.first	$ ftn test.first	test.first.bin

Source code file named without suffix	test	$ ftn test -b //good/compilers/newtest
		or
		$ ftn test -bx //good/compilers/newtest
			//good/compilers/newtest.bin
			or
			//good/compilers/newtest

6.4 f77 and cpp Command Options

Table 6–3 summarizes the command–line options available with the **f77** command. Some options are supported only in a BSD shell, some only in a SysV shell, and some in both. Also, note that the **–F** option has slightly different meanings in the different shells.

You can use the **f77 –W0** compiler option (see Table 6–3) to access the **ftn** command options that are not otherwise available with the **f77** command. For example, the following command line

$ [illegible]

passes the option **–l** and the argument **foo** to the compiler. See Table 6–3 for information about the **–W0** option.

Table 6–4 lists the C preprocessor (**cpp**) options that you can use when you compile with **f77**. To access a **cpp** option from **f77**, use the **f77 –Wp** option. For example, the following command line:

$ [illegible]

passes the –C option to the C preprocessor. No space is permitted between the comma and the option name. See Table 6–3 for information about the **–Wp** option.

The link editor (**ld**) command and its options are described separately in subsection 6.6.1. For a full description of the **f77**, **cpp**, and **ld** commands and their options, refer to the *BSD Command Reference* or the *SysV Command Reference*.

Table 6-3. **f77** *Command Options*

Option	What the Option Tells the Compiler	BSD	SysV
–c	Suppress the linking phase of the compilation and force an object file to be produced, even if only one program is compiled.	✓	✓
–g	Produce additional symbol table information for the debuggers.	✓	✓
–m	Apply the M4 macro preprocessor to each Ratfor source file before transforming it with the Ratfor processor.	✓	✓
–mp	Optimize and compile program using HP Concurrent FORTRAN (HPCF) product. This option is effective only when compiling programs to run on Series 10000 workstations.	✓	✓
–pg	Produce code that, when executed, creates a **gmon.out** file for use by the **gprof** utility.	✓	
–p	Produce code that, when executed, creates a **mon.out** file for use by the **prof** utility.	✓	
–q	Suppress printing of filenames and program unit names during compilation.	✓	✓
–v	Print the version number of the compiler and the name of each pass.	✓	✓
–w	Suppress all warning messages.	✓	✓
–A run[type],*type*	Cause the compiler to use the run-time semantics of the specified systype (*type*) regardless of the current environment setting. *type* can be any of the following: **bsd4.3** Berkeley 4.3BSD **sys5.3** System V Release 3 **any** program is independent of a particular UNIX system	✓	✓
–A sys[**type**],*type*	Cause the compiler to stamp the object module for execution under the specified version (*type*) of the UNIX system. *type* can be any of the following: **bsd4.3** Berkeley 4.3BSD **sys5.3** System V Release 3 **any** program is independent of a particular UNIX system	✓	✓

(Continued)

*Table 6–3. **f77** Command Options (Cont.)*

Option	What the Option Tells the Compiler	BSD	SysV
–A cpu,*id*	Generate code for the specified workstation type. *id* is usually one of the following: **mathlib_sr10** Generate Series 10000 code if you are compiling on a Series 10000 workstation, or MC680x0 code if you're compiling on an MC680x0 workstation. This is the default. **m68k** Generate code for a MC680x0 workstation **a68k** Generate code for a Series 10000 workstation Refer to Table 6–6 for a list of other possible arguments.	✓	✓
–C	Compile code that checks to see if subscripts are within array bounds. For multidimensional arrays, check only the equivalent linear subscript.	✓	✓
–F	Preprocess Ratfor files into **.f** files and leave those **.f** files on the disk without compiling them.		✓
–F	Apply the C preprocessor to **.F** files, apply the Ratfor preprocessors to **.r** files, put the result in the file with the suffix changed to **.f**, but do not compile.	✓	
–I2 –I4	Make the default integer constants and variables 2 or 4 bytes. The default is **–I4**.		✓
–O*n*	Produce optimized code. n is a single–digit integer in the range 1 – 4, indicating the level of optimization. If n is omitted, 3 is the default. If you omit the –O option altogether, no optimization is performed. .	✓	✓
–R *string*	Use *string* as a Ratfor option in processing **.r** files.	✓	✓
–T1*pathname*	Substitue the compiler at *pathname* for the compiler that **f77** calls by default.	✓	✓

(Continued)

*Table 6–3. **f77** Command Options (Cont.)*

Option	What the Option Tells the Compiler	BSD	SysV
–W0,*arg_list*	Hand off the arguments in *arg_list* to the compiler. Allows you to access **ftn** options when compiling with **f77**. If an option in *arg_list* takes an argument, it must also be separated from the option by a comma; for example, **f77 –W0,–opt,3** *prog_name*.	✓	✓
–Wl,*arg_list*	Hand off the arguments in *arg_list* to the link editor. Allows you to access **ld** options when compiling with **f77**. If an option in *arg_list* takes an argument, it must also be separated from the optionby a comma; for example, **f77 –Wl,–o**,*obj_name* *prog_name*.	✓	✓

Table 6–4. C Preprocessor (**cpp**) *Command Options*

Option	What the Option Tells the Compiler
–Wp,–C	Prevent the preprocessor from stripping comments.
–Wp,–D*name* [=*def*]	Define *name* to the preprocessor, as if by the C preprocessor statement, **#define**. If no definition is given, define *name* as 1. The **–D** option has lower precedence than **–U**; if both are used for the same name, the name will be undefined, regardless of the order in which the options appear. Note that there must be no space between –D and *name*.
–Wp,–H (SysV only)	Print out to **errout** the pathname of every file included during this compilation.
–Wp,–I*dir* (BSD only)	Change the search path for **%include** files with names enclosed in double quotes rather than angle brackets and not beginning with a slash (/). Look first in the directory of the source file in which the **%include** directive occurs, then in directories named in this option, and finally in directories on a standard list. Note that this option does not affect filenames enclosed in angle brackets. It is similar to the **ftn –idir** option, but the search rules are somewhat different. Note that there must be no space between –**I** and *dir*.
–Wp,–U*name*	Remove any initial definition of *name*. Note that there must be no space between –**U** and *name*.

6.5 ftn Command Options

Domain FORTRAN supports a variety of compiler options that you can use on the **ftn** command line. Table 6-5 summarizes the **ftn** compiler options, while the following sections describe **ftn** options in detail.

NOTE: You can use the **Options** statement to insert **ftn** compiler options in your source code. Refer to the listing for **options** in Chapter 4.

Table 6-5. **ftn** *Command Options*

Option	What It Causes the Compiler to Do
☆ -ac	Compile using AC (Absolute Code). This means that the compiler sets the load address at compile time, and therefore your program runs faster. Compare to **-pic** option. This option has no effect when compiling programs to run on Series 10000 workstations.
-alnchk	Display messages about the alignment of data, if used with **-info** option. Default for Series 10000 compilers.
☆ **-b** *pathname* **-nb**	Generate a binary file (that is, an executable object) at *program_name.bin* or *pathname.bin.* Suppress creation of binary file.
-bounds_violation ☆ **-no_bounds_violation**	Specify that program can violate array subscript boundaries during execution. Specify that program does not access array elements beyond the declared size of the array.
-bx *pathname*	Generate a binary file, without the **.bin** suffix.
-cond ☆ **-ncond**	Compile lines with D or d in column 1. Ignore lines with D or d in column 1.
-config *var1. . . varN*	Set conditional compilation variables to true.
-cpu *id*	Specify the workstation type for which the compiler is to generate code. The *id* argument is usually one of the following: **mathlib_sr10**, **mathlib**, or **mathchip**. The default for MC680x0 compilers is **mathlib_sr10**. The only valid argument for the Series 10000 compiler is **a88k**.
☆ denotes a default option	
NOTE: At SR10, the **-align** and **-nalign** options are obsolete.	

(Continued)

Table 6-5. **ftn** *Command Options (Cont.)*

Option	What It Causes the Compiler to Do
☆ **-db**	Generate minimal debugging information. When you debug this program, you can set breakpoints, but you can't examine variables.
-dba	Generate full run-time debug information and turn off all optimization.
-dbs	Generate full run-time debug information. This option has no effect on optimization.
-ndb	Suppress creation of debugging information. The debugger cannot debug such a program.
☆ **-dynm**	Allocate local variables on the stack (dynamic storage). Compare to **-save** option.
-exp	Generate assembly language listing at *program_name.lst*. (This option generates a listing file even if you omit **-l**.)
☆ **-nexp**	Suppress creation of assembly language listing.
-ff	Activate free format: length of source lines may be up to 1024 characters; an ampersand (&) in column 1 specifies a continuation line.
-frnd	Force the compiler to write all floating-point operands to memory so that floating-point comparisons produce the correct results.
☆ **-nfrnd**	Do not force the compiler to write all floating-point operands to memory.
-i*2	Use **integer*2** as the default integer type.
☆ **-i*4**	Use **integer*4** as the default integer type.
-idir *dir1. . . dirN*	Search for an include file in alternate directories.
-indexl	Disable some code optimizations generated for subscript calculations, and cause all array indexing to use 4-byte integer arithmetic.
☆ **-nindexl**	Use all the code optimizations generated for subscript calculations.
-info *n*	Display information messages to the *n*th level. *n* is an integer between 0 and 4. If *n* is omitted, or the entire option is omitted, display information messages to level 2.
☆ **-ninfo**	Suppress information messages.

☆ denotes a default option

(Continued)

Table 6-5. **ftn** *Command Options (Cont.)*

Option	What It Causes the Compiler to Do
-inlib *pathname*	Load *pathname* (a PIC binary file) at run time and resolve global variable references. Thus you can use *pathname* as a library file for many different programs.
-inline *char*	Select *char* as an inline comment designator. The default *char* is "{".
-l *pathname* ☆ **-nl**	Generate a listing file at *program_name.lst* or *pathname.lst*. Suppress creation of listing file.
-l*1 **-l*2** ☆ **-l*4**	Use **logical*1** as the default logical type. Use **logical*2** as the default logical type. Use **logical*4** as the default logical type.
-mp	Optimize and compile program using HP Concurrent FORTRAN (HPCF) product. This option is effective only when compiling programs to run on Series 10000 workstations.
☆ **-msgs** **-nmsgs**	Generate final error and warning count message. Suppress creation of final error and warning count message.
-natural ☆ **-nnatural**	Make natural alignment the default for this compilation. Make word alignment the default for this compilation.
-nclines	Suppress generation of COFF line number tables. This option has no effect when compiling programs to run on Series 10000 workstations.
-opt *n*	Optimize the code in the executable object to the *n*th level. *n* is an optional specifier that must be between 0 and 4. If *n* is omitted, or the entire option is omitted, optimize to level 3.
-overlap *args*	Specify the extent to which the program conforms to the ANSI FORTRAN standard for storage association. *args* can be one or more of the following: ☆ **no_dd** ☆ **no_dc** **exact_dd** **exact_dc** **dd** **dc**

☆ denotes a default option

NOTE: The **-nopt** option is obsolete; use **-opt 0** instead.

(Continued)

Table 6-5. **ftn** *Command Options (Cont.)*

Option	What It Causes the Compiler to Do
-pic	Compile using PIC (Position Independent Code). This means that the compiler uses relative addressing for your data and sets the address at run time. **-ac** is the default when compiling programs to run on MC680x0-based workstations; **-pic** is the default when compiling programs to run on Series 10000 workstations.
☆ **-prasm**	Create expanded listing in Series 10000 assembly-code format, if **-exp** is specified and if Series 10000 code is being generated.
-nprasm	Create expanded listing in an alternate assembly-code format.
-save	Allocate space for local variables in static storage, instead of on the stack. Compare with **-dynm**.
-subchk	Generate extra subscript checking code in the executable object file. This code signals an error if a subscript is outside the declared range for the array.
☆ **-nsubchk**	Suppress subscript checking.
-type	Issue warning messages for variables not explicitly typed.
☆ **-ntype**	Suppress checking for explicit declarations of variables.
-u	Turn on case-sensitivity for identifiers.
-uc	Turn on UNIX compatibility features: appended underscore to subprogram and common block names; UNIX version of default filenames.
-nuc	Turn off UNIX compatibility features
-version	Display version number and variant type of compiler. Note that you do not include a filename on the command line when you specify this option.
☆ **-warn**	Display warning messages.
-nwarn	Suppress warning messages.
-xref	Insert a symbol map and cross reference in the listing file. This generates the listing file at *program_name.lst*, even if you omit **-l**.
☆ **-nxref**	Suppress creation of symbol map and cross-reference listing.

☆ denotes a default option

(Continued)

Table 6-5. **ftn** *Command Options (Cont.)*

Option	What It Causes the Compiler to Do
☆ **-xrs** **-nxrs**	Save registers across a call to an external subprogram. Do not assume that calls to external subprograms have saved the registers.
-zero ☆ **-nzero**	Initialize to zero all common blocks and statically allocated variables. Do not initialize all common blocks and statically allocated variables.
☆ denotes a default option	

6.5.1 -ac and -pic: Memory Addressing

The **-ac** option is the default on MC680x0 systems. On Series 10000 systems, **-pic** is the default, and **-ac** is an invalid option.

The **-ac** option forces the compiler to produce absolute, or fixed-position, code, which generally executes faster than position-independent code (PIC). Absolute code programs are loaded at a fixed address that is determined at compile time. Since the loader does not have to set the load address at run time, your program runs faster.

The **-pic** option forces the compiler to produce PIC code. This means that the compiler uses relative addressing and that the addresses are set at run time, rather than at compile time.

In general, absolute code runs faster than PIC code, so you will not use the **-pic** option often. There are, however, a few situations where you must use the **-pic** option. In particular, you should produce PIC code for all routines that are to be entered into an installed library. In addition, you should produce PIC code for the following:

- Programs that invoke other absolute code programs in-process (for example, with the **pgm_$invoke** system call in **pgm_$wait** mode).
- Programs that are dynamically loaded, such as IOS type managers, GPIO drivers, and shared libraries.

Refer to the *Domain/OS Programming Environment Reference* manual for more information about absolute and position-independent code.

6.5.2 -alnchk: Alignment Messages

The **-alnchk** option is always set for compilers that generate Series 10000 code.

When you use the **-alnchk** option in conjunction with the **-info** option, the compiler displays messages telling you when your data is not naturally aligned. Naturally aligned data always improves the performance of your program on any workstation, but on Series 10000 workstations the improvement is significant. For information on how to naturally align data declared in **common** blocks, refer to the listing for "**Common**" in Chapter 4.

6.5.3 -b and -nb: Binary Output

The **-b** option is the default.

If you use the **-b** option without an argument, and if your source code compiles with no errors, Domain FORTRAN creates an object file with the source pathname and the **.bin** suffix. If you specify a pathname as an argument to **-b**, Domain FORTRAN creates an object file at **pathname.bin.**

If you use the **-nb** option, Domain FORTRAN suppresses the creation of an object file. Consequently, compilation is faster than if you had used the **-b** option. Therefore, **-nb** can be useful when you want to check your source code for grammatical errors, but you don't want to execute it.

Assuming that an error-free Domain FORTRAN source code is stored in file **test.ftn**, here are some sample command lines:

$ ftn test
{Domain FORTRAN creates **test.bin**}

$ ftn test -b
{Domain FORTRAN creates **test.bin**}

$ ftn test -b jest
{Domain FORTRAN creates **jest.bin**}

$ ftn test -B jest.bin
{Domain FORTRAN creates **jest.bin**}

$ ftn test -nb
{Domain FORTRAN doesn't create an object file}

6.5.4 -bounds_violation and -no_bounds_violation: Array Boundary Violation

The **-no_bounds_violation** option is the default.

By default, the compiler assumes that programs do not violate array subscript bounds during execution—that is, it does not attempt to access array elements beyond the declared size of the array. This assumption frees the compiler from certain restraints when optimizing a program.

If your program does violate array subscript bounds during execution, use the **-bounds_violation** option.

6.5.5 -bx: Suppressing .bin Suffix

The **-bx** option causes the compiler to create a binary object file without the suffix **.bin**.

By default, Domain FORTRAN appends the **.bin** suffix to object files it creates. This is the case even if you invoke the compiler with the **-b** option to name the file. Using the **-bx** option, however, causes the compiler to name the object file without the **.bin** suffix. For example, the following command line will create an object file named **test**:

$ ftn my_ftn_program.ftn -bx test

6.5.6 -cond and -ncond: Conditional Compilation

The **-ncond** option is the default.

The **-cond** option invokes conditional compilation. If you compile with **-cond**, Domain FORTRAN treats lines with a D or d in column 1 as source code, and therefore compiles them.

If you compile with **-ncond**, Domain FORTRAN treats the lines of source code with a D or d in column 1 as comments.

You can simulate the action of this option with the **-config** compiler option. For new program development, you should use the **-config** syntax, since this option is considered obsolete.

6.5.7 -config: Conditional Processing

Use the **-config** option to set conditional variables to true. (Refer to the Compiler Directives listing of Chapter 4 for details on the conditional variables.)

You declare these conditional variables with the **%var** compiler directive. By default, their value is false. You can set their value to true with the **%enable** directive (described in Chapter 4) or with the **-config** option. The **-config** option's format is

-config *var1* ... *varN*

where *var* must be a conditional variable declared with **%var**.

For example, consider what happens when you compile the following program both without **-config** and with **-config** (the program, **config_example**, is available online):

```
          program config_example

      integer*2 x, y, z
      data x, y, z /0, 0, 0/

      print *, 'The start of the program.'

%var first, second, third
%if first %then
      x = 5
      print *, x, y, z
%endif

%if second %then
      y = 10
      print *, x, y, z
%endif

%if third %then
      z = 15
      print *, x, y, z
%endif

      print *, 'The end of the program.'

      end
```

First, notice what happens when you compile without -**config**.

```
$ ftn config_example
no errors, no warnings in CONFIG_EXAMPLE, Fortran version n.nn . . .
$ config_example.bin
 The start of the program.
 The end of the program.
```

Now, use the -**config** option to set conditional variables **first** and **third** to true. Here's what happens:

```
$ ftn config_example -config first third
no errors, no warnings in CONFIG_EXAMPLE, Fortran version n.nn . . .
$ config_example.bin
 The start of the program.
 5 0 0
 5 0 15
 The end of the program.
```

To simulate the action of the **-cond** compiler option, precede the section of code you want conditionally compiled with **%if** *config_variable* compiler directive and conclude it with the

%then compiler directive. Then use **-config** to set *config_variable* to true when you want to compile that section of code.

6.5.8 -cpu: Target Workstation Selection

The **-cpu mathlib_sr10** option is the default if your compiler generates MC680x0 code. The **-cpu a88k** option is the default if your compiler generates Series 10000 code.

Use the **-cpu** option to select the target workstations that the compiled program can run on. If you choose an appropriate target workstation, your program may run faster; however, if you choose an inappropriate target workstation, the run-time system will issue an error message telling you that the program cannot execute on this workstation.

You select the code generation mode with the argument that you specify immediately after **-cpu**. Table 6-6 shows the possible arguments and the code generation mode that they select.

NOTE: If you use the **f77** command and wish to include this option on the command line, you must precede it with the **-A** option, as follows:

f77 -A cpu,*id*

where *id* is one of the arguments listed in Table 6-6. There must be no space between the comma and *id*.

Table 6-6. Arguments to the **ftn** *-cpu Option*

Argument	What the Argument Causes the Compiler to Do
☆ **mathlib_sr10**	Generates code for workstations with an MC68020, MC68030, or MC68040 microprocessor and an MC68881 or MC68882 floating-point coprocessor. The **-cpu mathlib_sr10** option is the default if you are compiling for an MC680x0-based system.
mathlib	Generates code for workstations with an MC68040 microprocessor (including the HP Apollo 9000 Series Models 425t and 433s). Also generates code for MC68020- and MC68030-based workstations with an MC68881 or MC68882 floating-point coprocessor. Programs compiled with this argument run only on SR10.3 and later versions of Domain/OS. Use **-cpu mathlib_sr10** for compiling programs that must also run on SR10.0, SR10.1, or SR10.2.
mathchip **3000** **580** **570** **560** **330** **90**	Generates code for workstations with an MC68020 or MC68030 microprocessor and an MC68881 floating-point coprocessor. All seven arguments are synonyms; they all generate exactly the same code. We recommend that you use the **mathchip** argument; the other six arguments become obsolete at a future compiler release.
peb	Generates code for workstations with a Performance Enhancement Board (PEB) (includes the DN100, DN320, DN400, and DN600, when equipped with an optional PEB).
160 **460** **660**	Generates code for a DSP160, DN460, or DN660 workstation. These three arguments are synonyms; they all generate exactly the same code.
fpx	Generates code for DN5xx workstations with an FPX floating-point accelerator unit. Domain FORTRAN allows you to compile programs on the DN570-T and DN570-T without using the **-cpu fpx** option, but such programs will not take maximum advantage of the FPX unit.
fpa1	Generates code for Series 3000, Series 4000, or Series 4500 workstations with an FPA1 floating-point accelerator unit.
any	Generates Series 10000 code, if you are compiling for a Series 10000 workstation, or generic MC680x0 code, if you are compiling for an MC680x0-based workstation.
☆ **a88k**	Generates code for a Series 10000 workstation. The **-cpu a88k** option is the default if you are compiling for a Series 10000 workstation.
m68k	Generates code for any MC680x0-based workstation.
☆ denotes a default option	

The advantage of using the **-cpu** option to select the appropriate code-generation mode is that the compiler generates code that is optimized for the selected processor. Programs so compiled generally run faster, especially if they make frequent use of floating-point operations. Programs that frequently perform 32-bit integer multiplication and division also run better.

The **-cpu mathchip** option generates the best possible code for the following Apollo workstations:

HP Apollo 9000 Series 400 Model 400s

HP Apollo 9000 Series 400 Model 400t

DN4500

DN4000

DN3500

DN3000

DN2500

DN580

DN570

DN560

DN330

DSP90

The default option for MC680x0-based workstations, **-cpu mathlib_sr10**, generates code that runs well on any of the above workstations.

Table 6-7 shows the relative performance of the MC680x0 code generated with different arguments to the **-cpu** option.

Which argument you use with the **-cpu** option can also affect performance of floating-point applications, especially those that execute on MC68040-based workstations. For more information, refer to Appendix D.

For online information about the **-cpu** option and its arguments, use the **cpuhelp** utility. This utility tells you which **-cpu** argument is appropriate for a particular machine and which machines a particular **-cpu** argument will work on. For more information about this utility, type **help cpuhelp** in the Aegis environment or **man cpuhelp** in a UNIX environment.

Table 6-7. Relative Performance of MC680x0 Code with Different -cpu *Arguments*

	Machine Type						
Argument	**DN100/ 400**	**PEB**	**DN160/ 460/660**	**FPX**	**FPA1**	**MC68020/ MC68030**	**MC68040**
mathlib_sr10	N/A	N/A	N/A	fair	⋆	fair	good
mathlib	N/A	N/A	N/A	fair	⋆	good	best
mathchip	N/A	N/A	N/A	fair	⋆	best	fair
peb	N/A	best	N/A	N/A	N/A	N/A	N/A
160, 460, 660	N/A	N/A	best	N/A	N/A	N/A	N/A
fpx	N/A	N/A	N/A	best	N/A	N/A	N/A
fpa1	N/A	N/A	N/A	N/A	best	N/A	N/A
any	best	fair	poor	poor	poor	poor	fair

⋆Selecting this option forces the compiler to select instructions that do not use the FPA1 floating-point accelerator unit. The resulting code runs exactly the same as code generated for an MC68020- or MC68030-based machine.

NOTE: The terms **poor**, **fair**, **good**, and **best** are relative; they compare performance between different **-cpu** arguments on one machine, not between machines. For example, code that is **fair** for an MC68040-based machine runs faster than the **best** code on an MC68020-based machine.

6.5.9 -db, -ndb, -dba, -dbs: Debugger Preparation

The **-db** option is the default.

Use these options to prepare the compiled file for debugging by the Domain Distributed Debugging Environment. The options allow for different levels of debugging access, which are summarized in Table 6-8.

Table 6-8. Debugger Preparation Options

Option	Debugging Access
-ndb	None
-db	Minimum access: Source line numbers
-dbs	Maximum access: Source line numbers and symbols
-dba	Same as **-dbs** but without any code optimization

Domain FORTRAN stores the debugger preparation information within the executable object file, so, in general, the more debugger information you request, the larger your executable object file. Of the three options, **-dba** results in the largest object module, followed by **-dbs** and **-db**.

If you use the **-ndb** option, the compiler does not put any debugger preparation information into the **.bin** file. If you try to debug such a **.bin** file with **debug**, the system reports the following error message:

```
?(dde) The target program has no debugging information.
```

If you use the **-db** option, the compiler puts minimal debugger preparation information into the **.bin** file. This preparation is enough to enter the debugger and set breakpoints, but not enough to access symbols (for example, variables and constants).

If you use the **-dbs** option, the compiler puts full debugger preparation information into the **.bin** file. This preparation allows you to set breakpoints and access symbols. The **-dbs** option has no effect on optimization.

The **-dba** option is identical to the **-dbs** option except that when you use the **-dba** option, the compiler turns off all optimizations, even if you specify **-opt**.

> NOTE: The **-dba** option overrides anything you specify for the **-opt** option. When you specify **-dba**, all optimization is disabled, regardless of what you specified for **-opt** on the command line for the compilation. Refer to Section 6.5.26 for more details about the optimizations that are set with **-dba**.

For more complete details on these four options, see the *Domain Distributed Debugging Environment Reference*.

6.5.10 -dynm: Stack Storage

The **-dynm** option is on by default.

The **-dynm** option requests dynamic (stack) storage for local variables in subprograms. Variables stored on the stack do not retain their values from one invocation of the subprogram to the next. To force static (as opposed to dynamic) storage, compile with the **-save** option.

6.5.11 -exp and -nexp: Expanded Listing File

The **-nexp** option is the default.

If you compile with the **-exp** option, the compiler generates an expanded listing file at *program_name.lst*. This listing file contains a representation of the generated assembly language code interleaved with the source code.

If you compile with the **-nexp** option, the compiler does not generate a listing file (unless you use the **-l** option).

If you are generating Series 10000 code and you use the **-exp** option, the compiler by default generates an expanded listing in Series 10000 assembly code format. If you want to generate this listing in an alternate assembly-language format, use the **-nprasm** option in conjunction with **-exp**. For more information about the **-nprasm** option, refer to Section 6.5.29.

> NOTE: At SR10, you cannot use both **-exp** and **-xref** on the same command line.

6.5.12 -ff: Free Format

Use **-ff** to activate the following features:

- Length of source lines may be up to 1024 characters.
- An ampersand (&) in column 1 specifies a continuation line.

If you do not specify **-ff**, then

- The compiler ignores characters after the 72nd character.
- You must specify continuation lines by using a continuation character in column 6.

6.5.13 -frnd and -nfrnd: Store and Round Floating-Point Numbers

The **-nfrnd** option is the default.

The **-nfrnd** option causes the compiler to generate code that computes floating-point expressions in at least the precision specified by the program. If the compiler detects an opportunity to optimize execution by doing arithmetic in a greater precision, it does so by keeping floating-point operands in registers rather than by storing them back into memory locations. Floating-point operands held in registers retain greater precision and permit faster execution speed than if they were held in memory.

The **-frnd** option causes floating-point numbers to be rounded to the precision specified by the program (either 32-bit single precision or 64-bit double precision) at key points in the program's execution. The result is that programs compiled with **-frnd** yields results more like those obtained on machines using different floating-point representations, but at the expense or more execution time and less precision.

By default, the registers used for floating-point calculations are extended precision (80-bit) registers. Using the **-frnd** option causes the compiler to behave as if it were using 32-bit and 64-bit registers. The truncation of results to the correct register size before comparison decreases execution time but may also decrease accuracy. Therefore, we recommend that you use the **-frnd** option only if you have a special requirement to make your program behave the same as a 32-bit or 64-bit machine.

6.5.14 -i*2 and -i*4: Integer Size

The **-i*4** option is the default.

Domain FORTRAN supports both 2-byte (**integer*2**) and 4-byte (**integer*4**) integers. Since the **-i*4** option is on by default, FORTRAN assumes that all integer constants, implicit integers, and integer variables are four bytes long, unless the source code specifies otherwise. That is, in this fragment

```
integer   int_num
integer*2 small_int
```

int_num by default is considered a 4-byte integer. However, **small_int** is explicitly declared to be an **integer*2** variable, and Domain FORTRAN always uses the explicit declaration, if one exists.

If you compile with the **-i*2** option, the compiler allocates two bytes for each integer. Given the above example, **int_num** would be a 2-byte integer.

6.5.15 -idir: Search Alternate Directories for Include Files

The **-idir** option specifies the directories in which the compiler should search for include files if you specify such files using relative, rather than absolute, pathnames. Absolute pathnames begin with a slash (/), double slash (//), tilde (~), or period (.).

Without the **–idir** option, Domain FORTRAN searches for include files in the current working directory. For example, if your working directory is **//nord/minn** and your program includes this directive

```
%include 'mytypes.ins.ftn'
```

Domain FORTRAN searches for that relative pathname at **//nord/minn/mytypes.ins.ftn**. However, when you use **–idir**, the compiler first searches for the file in your working directory, and if it doesn't find the file, it looks in the directories you list as **–idir** arguments. When it finds the include file, the search ends. This capability is useful if you have include files stored on multiple nodes or in multiple directories on your node.

Remember that as of SR10, pathnames must be case-correct.

For example, consider the following compile command line:

$ ftn test –idir //ouest/hawaii

This command line causes the compiler to search for **mytypes.ins.ftn** at **//ouest/hawaii/mytypes.ins.ftn** if it can't find **//nord/minn/mytypes.ins.ftn**.

Up to 63 pathnames can follow an **–idir** option. Separate each pathname with a space.

6.5.16 –indexl and –nindexl: Array Reference Index

The **–nindexl** option is the default.

Domain FORTRAN normally tries to optimize the code generated for subscript calculations based on array dimension information. The **–indexl** option disables some of this optimization and causes the subscript calculations of all dummy argument arrays and **common** block arrays to be done using 4–byte integer arithmetic and 32–bit indexing. On some machines, this can affect program performance. Arrays with assumed dimensions (for example, **iarray(*)**) are always referenced with 32–bit indexing.

6.5.17 –info and –ninfo: Information Messages

The **–info 2** option is the default.

The **–info** option allows you to tell the compiler which, if any, types of information messages to display. You specify the types of message by means of an **information message level**. The syntax for the **–info** option is:

–info *n*

where *n* is an integer between 0 and 4 that represents the information message level. The **–ninfo** option is equivalent to **–info 0**.

The compiler displays messages for all levels *up to* and *including* the level you specify. In other words, if you specify **-info 2**, the compiler displays all the messages specified by **-info 2** *plus* all the messages specified by **-info 0** and **-info 1**.

Domain FORTRAN provides the following information message levels:

-info 0 At this level, no messages are displayed.

-info 1 Messages at this level pertain to the size and alignment of variables and types.

If you are porting a program from a system that allocates all variables in static storage and you have access to a Series 10000 compiler, you can use **-info 1** (or greater) to determine which scalar variables must be allocated static storage for the program to execute correctly. All other scalar variables can be allocated to registers to optimize performance. Compile the program using the **-info 1** and **-save** options. The compiler will issue diagnostic messages for all scalar variables that might have been allocated to registers but must be allocated static storage in order for the program to execute correctly. Only the listed scalar variables need to be listed in **save** statements. For more information, refer to the listing under **save** in Chapter 4 and to subsection 6.5.30.

-info 2 Messages at this level describe optimizations performed by the compiler.

The **-info 3** and **-info 4** options provide no additional messages beyond **-info 2**.

6.5.18 -inlib: Library Files

Use **-inlib** to tell the compiler to load library files at run time that are not installed at compile time. The **-inlib** option makes code available to an executing program without actually binding the code into the output object file. If you try to access library files that have not been specified with the **-inlib** option, your program will not work as you intended.

The syntax for the **-inlib** option is

-inlib *pathname*

where *pathname* specifies the library file. The file in *pathname* must be a binary file that was created using the **-pic** option.

When you use the **-inlib** option,

1. The compiler puts the pathname of the library in the binary file.

2. The compiler uses global symbols in the library to identify externals which are *not* absolute data or procedure references.

3. At run time, the loader loads the library file, and the externals identified in Step 2 are dynamically linked.

6.5.19 -inline: Inline Comment Designator

Domain FORTRAN allows the use of inline comments—that is, comments that appear at the end of a statement line. The default designator for inline comments is a left brace character ({).

The **-inline** option lets you select your own character as the designator for inline comments.

It has this syntax

-inline '*char*'

where *char* is a single character, which your program will use for inline comments. You must enclose *char* in single quotes. For example, suppose you write a program named **questions.ftn** and use "!" as an inline comment designator. If your code includes something like the following:

```
if (answer .eq. 'Y') done = .true.    !When answer is 'Y', we're done.
```

you should compile as follows:

$ ftn questions -inline '!'

6.5.20 -l and -nl: Listing Files

The **-nl** option is the default.

The **-l** option creates a listing file. The listing file contains the following:

- The source code complete with line numbers. Note that line numbers start at 1 and move up by 1 (even if there is no code at a particular line in the source code). Further note that lines in an include file are numbered separately.
- Compilation statistics.
- A section summary.
- A count of error and warning messages produced during the compilation.

The format for the **-l** option is

-l *pathname*

If you specify a pathname following **-l**, the compiler creates the listing file at *pathname.lst*. If you omit a pathname, the compiler creates the listing file with the same name as the source file. If the source filename includes the **.ftn** suffix, **.lst** replaces it. If the source filename does not include **.ftn**, **.lst** is appended to the end of the name.

If you use the **-xref** option along with **-l** (or in place of it), the listing file also contains a symbol map and cross reference. If you use the **-exp** option along with **-l** (or in place of it), the listing file also contains the generated assembly language code (from **-exp**).

The **-nl** option suppresses the creation of the listing file except if you also specify the **-exp** or **-xref** options.

6.5.21 -l*1, -l*2, and -l*4: Logical Size

The **-l*4** option is the default.

Domain FORTRAN supports 1-byte (**logical*1**), 2-byte (**logical*2**), and 4-byte (**logical*4**) logical data types. By default, FORTRAN assumes that all logical constants, implicit logicals, and logical variables are four bytes long. You can override the default by specifying **l*1** or **l*2** variables in the source code, or by using the **-l*1** and **-l*2** options.

For example, consider the following fragment:

```
logical   switch
logical*2 small_switch
```

By default, the compiler treats **switch** as a 4-byte logical. However, **small_switch** is explicitly declared to be a **logical*2** variable. Domain FORTRAN uses the explicit declaration for **small_switch**.

If you compile the preceding example with the **-l*2** option, the compiler allocates two bytes for **switch** instead of four bytes.

If you compile the preceding fragment with the **-l*1** option, the compiler allocates one byte for **switch** instead of four bytes.

6.5.22 -mp: Processing for Parallel Execution

The **-mp** option is for use with the HP Concurrent FORTRAN (HPCF) product. It enables the compiler to optimize and compile FORTRAN programs for parallel execution on Series 10000 machines equipped with multiple processors. For more information, refer to the *HP Concurrent FORTRAN User's Guide*.

6.5.23 -msgs and -nmsgs: Message Report

The **-msgs** option is the default.

If you use the **-msgs** option, the compiler produces a final compilation report with the following format:

xx errors, *yy* warnings, *zz* informational messages, Fortran 77 compiler
variant Rev *n.n. 68K Rev 10.7(294) yyy/mm/dd hh:mm:ss*

where *xx*, *yy*, and *zz* are either "no" or a number, *n.n.*, is the version number, *yyy/mm/dd* is the date, *hh:mm:ss* is the time, and *variant* is one of the following:

```
68K
PRISM
68K=>PRISM
PRISM=>68K
```

(Refer to Section 6.2 for information about compiler variants.)

If you use **-nmsgs**, the compiler suppresses the final compilation report.

6.5.24 -natural and -nnatural: Natural Alignment in Common

The **-nnatural** option is the default.

The **-natural** option forces the compiler to align each object declared in a **common** block on its natural boundary—that is, at an address that is a multiple of the object's size in bytes. Thus, single-byte objects are naturally aligned at any address (for example, 1000, 1001, and 1002), 2-byte objects start at addresses that are a multiple of two (for example, 1000, 1002, and 1004), 4-byte objects start at addresses that are a multiple of four (for example, 1000, 1004, and 1008), and 8-byte objects start at addresses that are a multiple of eight (for example, 1000, 1008, and 10010). Programs run faster when their data is naturally aligned.

The **-nnatural** option (the default) specifies word-alignment for all objects larger than a byte that are declared in a **common** block.

When compiling a program that consists of several source files that communicate through a **common** block, if you compile one of the source files with the **-natural** option, you must compile the others with the **-natural** option as well. If you don't, the objects in the **common** blocks will be inconsistently aligned and will produce bad data. This problem is compounded by the fact that the compiler cannot detect the inconsistent alignment. Therefore, when compiling a multiple source-file program, use the **-natural** option either on all the modules or on none of them.

For more information on natural alignment and **common** blocks, refer to the listing under "**common**" in Chapter 4. For information on natural alignment and the Series 10000 workstation, refer to the *Series 10000 Programmer's Handbook*.

6.5.25 –nclines: COFF Line Number Tables

By default, the compilers that generate code for the MC680x0–based workstations generate COFF line number tables whenever you compile using the **–dba** or **–dbs** option. However, the Domain Distributed Debugging Environment does not require these tables, nor does the Domain traceback (**tb**) tool. You can use the **–nclines** option to tell the compiler to suppress the generation of these tables.

Since these tables require a lot of disk space, you should use the **–nclines** option if you wish to save space and do not need the COFF line number tables.

> NOTE: This option has no effect on compilers that generate code for Series 10000 workstations.

6.5.26 –opt: Optimized Code

The **–opt 3** option is the default.

Use the **–opt** option to tell the compiler the optimizations to perform on your source program. Select an optimization level to indicate the optimizations you want. The syntax for the **–opt** option is

–opt *n*

where *n* is an integer between 0 and 4 that represents the optimization level.

If you specify **–opt** and omit the optimization level, the level defaults to **–opt 3**. If you omit **–opt** completely, the default option, **–opt 3**, is assumed. The **–nopt** option is equivalent to **–opt 0**. The **–optall** option is equivalent to **–opt 3**. Both **–nopt** and **–optall** are obsolete.

At **–opt 0** the compiler performs very few optimizations. At each higher optimization level, the compiler performs more optimizations. Each higher level of optimization includes all optimizations performed at the lower levels of optimization.

> NOTE: Because the compiler does increasingly more work at successive levels of optimization, it takes longer to compile your program at each successive optimization level. If you are just beginning to develop your program, and you are compiling mainly to find syntax errors, you may want to compile using a low optimization level to reduce the compile time. When you are ready to test the execution of your program, you can compile with a higher optimization level to take advantage of all the optimizations.

The following is a brief description of the optimizations performed at each optimization level. Note that we include **-dba** in the list of optimization levels because it affects optimization. For a more detailed discussion of compiler optimization techniques, consult a general compiler textbook.

- **-dba** represents the lowest possible optimization level. At this level the compiler does even less optimization that at **-opt 0**. The **-dba** option tells the compiler to store machine registers in main memory after every statement. Even with the **-dba** option the compiler still does some optimizations. Specifically, the compiler does the following:

  - Rearranges expressions to minimize the number of registers needed to compute them.

  - Generates faster short-range branch instructions in place of long branches where possible.

  - Computes constant expressions that appear in the source code rather than generating code to compute them.

  Note that the **-dba** option overrides anything you specify for the -**opt** option. In other words, when you specify **-dba**, the **-opt** option is set to **-opt 0**, regardless of what you specify for **-opt** on the command line.

  If you want your code to be optimized, and you want to use the debugger on your program, use the **-dbs** option rather than **-dba**. Refer to subsection 6.5.9 for details about using the **-dbs** and **-dba** options.

- **-opt 0** performs the optimizations listed above. In addition, the compiler

  - Permits values to remain in registers across statements (where it is legal to do so, and if **-dba** is not also set).

  - Replaces a repeated sequence of instructions in generated code by a branch to a set of instructions that is identical.

- **-opt 1** performs the following optimizations:

  - Eliminates limited global **common subexpressions**. A common subexpression is an expression that appears two or more times in the program, with no intervening assignments to any component of the expression. In such cases, the compiler computes the value of the expression only one time and uses the resulting value to replace other occurrences of the expression.

  - Eliminates **dead code**. Dead code is code that cannot be executed because there is no execution path of the program that leads to the code.

  - Transforms integer multiplication by a constant into shift and add instructions rather than using direct multiply (where appropriate).

- Performs simple expression transformations for speed.
- Merges assignment statements where possible.

- **-opt 2** performs the following optimization:

  - Substitutes constants for **reaching definitions** (see details below).

    When you make an assignment to a variable or use the variable as a parameter in a function call, the compiler produces a definition of the variable. If there are no other definitions between the original definition and the use of the variable, then a particular definition of a variable is said to **reach** later uses of the variable.

    If the definition of the variable is an assignment of a constant to the variable, then the compiler can replace uses of the variable that the definition reaches with the constant. As the compiler makes these substitutions, it transforms the expressions into constant expressions that can be evaluated during compilation. Thus there is no need to generate code to compute the value of the expression.

    For example, in the statements

    ```
    A = 3
    C = 2 * A
    ```

    there are no other definitions of the variable A between the original assignment and the use of A in the expression 2*A. Thus the compiler can substitute the value 3 in the expression 2*A. The expression then becomes 2*3, which is computed during compilation. As a result, the program does not perform a multiply when it executes. Instead, it merely assigns the previously computed value 6 to A.

    The compiler evaluates a floating-point value in a constant expression as a double-precision value, even if the original variable is single-precision.

- **-opt 3** is the default. It performs the following optimizations:

  - Inline expansion of all statement functions and all subprograms enclosed between the **%begin_inline** and **%end_inline** compiler directives. You can also use the **%begin_noinline** and **%end_noinline** compiler directives to selectively disable inline expansion of subprograms. Refer to the listing under "Compiler Directives" in Chapter 4 for a full description of the inline expansion directives.

  - Eliminates redundant assignment statements.

Redundant assignment elimination performed at this optimization level may result in informational messages such as the following:

```
**** Informational #929 on Line 9: Assignment eliminated,
     value never used: SMALL
```

Consider the following example:

```
PROGRAM B
INTEGER I, J
READ(5,*) I, J
IF ( I .EQ. 0) THEN
                J = 3
ENDIF
WRITE(6,*) I
END
```

There are no uses of the variable **J** after the assignment **J=3**. Since the value assigned to **J** is not used, the compiler can eliminate the assignment completely without changing the result computed in the program. In fact, once the assignment is eliminated, the **if** portion of the statement isn't needed either, and can be eliminated. If we change the example so that **J** is used after the assignment, as shown below, then the assignment is no longer eliminated.

```
PROGRAM B
INTEGER I, J
READ(5,*) I, J
IF ( I .EQ. 0) THEN
J = 3
ENDIF
WRITE(6,*) I,J
END
```

— Global register allocation.

Global register allocation allows variables that are local to a routine to have their values placed in machine registers for faster access. In many cases, all definitions and uses of a local variable may occur in a register, and the copy of the variable in the computer's main memory is never used or updated. Keeping variables in registers makes your program execute faster.

— Instruction reordering.

Instruction reordering changes the order in which instructions are executed in order to take advantage of possible overlaps in some instruction sequences. For example, some integer instructions can execute at the same time as some floating-point instructions, as long as the integer instructions do not depend upon the result computed by the floating-point instructions.

- Removes invariant expressions from loops.

  A **loop invariant expression** is an expression whose value does not change during the execution of a loop. When the compiler computes invariant expressions outside a loop, it does so only once. Thus the loop executes faster. For example:

```
          DO 10 I=1, 10
                 J = K * L
                 J = I + J
   10     CONTINUE
```

  The expression **K * L** is invariant in the above example. The compiler can safely transform this loop as follows:

```
         TEMP = K * L
         DO 10 I=1, 10
               J = TEMP
               J = I + J
   10    CONTINUE
```

  After the invariant expression is removed from the loop, the program in the example does only one multiply (instead of 10) to make the assignment to **J**.

- Software pipelining.

  Software pipelining is available only on the Series 10000 workstation. It is a technique for optimizing loops by overlapping loop iterations. This technique takes advantage of the Series 10000 workstation's ability to execute multiple instructions concurrently.

  At **-opt 3**, the compiler finds opportunities for concurrent execution of instructions by identifying small, highly iterative loops as candidates for software pipelining. Such loops must have no internal control flow (for example, no subroutine calls) and no recurrence (that is, data that is defined in one iteration must not be used or destroyed in another iteration).

  When debugging a program that the compiler has software pipelined, you'll notice that some instructions from different loop iterations execute at the same time. If you find it difficult to debug such a program, recompile with it the **-dba** option, which turns off all optimizations, including software pipelining.

- Strength reduction.

  Strength reduction is an optimization performed on expressions in loops. The purpose of strength reduction is to reduce execution time in the loop by substituting equivalent faster operations for slower ones.

For example, consider the following loop:

```
          DO 10 I = 1, 10
          J = I * 5
10        CONTINUE
```

In the loop above, **I** is a counter or induction variable, whose value is incremented by a constant each trip through the loop. We can replace multiplication expressions involving induction variables with addition operations, and get faster loop execution. The loop above might be changed to look like this:

```
          T$00001 = 1 * 5    { This operation folds to just 5 }
          DO 10 I =1, 10
          J = T$00001
          T$00001 = T$00001 + 5
10        CONTINUE
```

Strength reduction has replaced the multiplication with an equivalent addition. Notice that a new variable, **T$00001**, has been introduced. This variable is called a **strength reduction temporary variable**. It is used to take the place of **I**, whose value may be needed at a later point in the program. However, if **I** is not used in the loop in expressions that cannot be strength reduced, and it is not used on a later execution path from the loop, all assignments to **I** may be eliminated. Since the assignment elimination is a side effect of strength reduction, no warning message is issued when this occurs. This may make it difficult to examine induction variables when you are debugging your program, but it eliminates the increment and store of the induction variable in the loop. Of course, you could change your source code yourself and achieve the same effect the optimizer produces.

More frequently, strength reduction helps to eliminate hidden multiplication operations in array accesses. For example, whenever you reference an element in an array, say **A(i)**, there is an address calculation that must take place in order to fetch the correct element of **A**. The calculation for the address is as follows:

(base address of A – 4) + (i * element size of A)

Notice that there is a multiply that will appear in the generated code, even though no explicit multiply appears in your source code.

Consider the following loop:

```
          DO 10 I = 1,10
                   A(I) = 0.0
          10       CONTINUE
```

Even though there are no explicit multiplication operations in the source code, the array reference introduces a multiplication operation, and an opportunity for strength reduction, since the array index **I** is an induction variable. We can transform this loop as follows:

```
           T$00001 = (base address of A - 4) + ( 1 * 4 )
           DO 10 I = 1,10
           T$00001^ = 0.0
           T$00001 = T$00001 + 4
10         CONTINUE
```

In this example, **T$00001^** means an indirect reference through the variable **T$00001**, which contains the address of the selected array element. Notice that strength reduction has succeeded in eliminating the multiplication inside the loop for the array reference, and has also moved the addition of the base address of **A** to a point outside the loop. In some cases, the increment of **T$00001** may be accomplished at the same time the array element is stored, by means of auto-increment addressing modes in the machine instructions. Again, all references to **I** may be eliminated if it is legal to do so in the context of the surrounding program.

— Live analysis on local variables.

When the compiler performs **live analysis** of local variables, it determines the areas of a routine where a variable is actively used. For example,

```
J = K
IF (I .EQ. 0) THEN
   I = 2
   J = 3 * J
ELSE
  K =  I * 4
  WRITE(6, *) K
ENDIF
```

In this example, **J** is not used in the **else** clause. Furthermore, **J** is not used on any execution path from the **else** clause to the end of the program, nor on execution paths from the **else** to other parts of the routine. **J** is therefore considered dead from the statement following the **else** to the end of the routine.

Within the **then** clause, there is a use of **J**. Therefore, **J** is considered live within the **then** clause. If there are other uses of **J** that can be reached from the **then** clause, **J** is considered live along the paths that lead to those uses.

Live analysis is important because it allows the compiler to allocate local variables to machine registers for exactly as long as the variable's value is needed. When the variable becomes dead, the register can be used for other variables or expression values. In general, referencing a value in a register is faster than referencing a value in the computer's main memory. Thus, if the compiler allocates registers efficiently, your programs run faster.

- Exhaustive searches for global common subexpressions to eliminate.

  Note that the **-opt 1** and **-opt 2** levels make only limited searches through the code for global common subexpressions.

- **-opt 4** performs the following optimizations:

  - Reduces execution time in loops.

    The compiler examines each loop to determine whether there are any calls to subroutines or functions in the loop. It also determines whether there are **goto** statements in the loop whose target label is outside the loop. If there are no calls or **gotos** that go outside the loop, the compiler attempts to load **common** variables into registers outside the loop and store them back into the **common** block after the loop's execution.

    This optimization produces faster access to variables in **common** blocks, and increases the execution speed of the loop. If your program uses variables that are in **common** blocks within the range of loops, you may benefit from this level of optimization.

    NOTE: If your program uses cleanup handlers to process exceptions raised during program execution, and your cleanup handlers expect to access variables in **common**, you should *not* use this level of optimization. Suppose that an exception, such as division by zero, is raised during execution of a loop where the compiler has placed **common** variables in registers. When control transfers to your cleanup handler to process the exception, your cleanup handler will be unable to access the current values of these **common** variables. See the *Domain/OS Call Reference* and *Programming with Domain/OS Calls* for more information concerning cleanup handlers.

  - Eliminates some simple loops and replaces them with calls to the vector library.

    The vector library calls supported by this optimization are:

vec_$add_constant	**vec_$dmult_add**
vec_$dadd_constant	**vec_$mult_constant**
vec_$add	**vec_$dmult_constant**
vec_$dadd	**vec_$dot**
vec_$sub	**vec_$ddot**
vec_$dsub	**vec_$sum**
vec_$mult_add	**vec_$dsum**

Specifically, for example, a loop such as

```
do i=1, 100
      A(i) = B(i) + x
end do
```

would be translated into the following vector library call:

```
vec_$add_constant(B, 100, x, A)
```

The following restrictions apply to the types of loops that are optimized:

1. There may be only one statement in the loop.
2. The statement in the loop must be an assignment statement.
3. The loop must translate exactly into one of the vector library calls listed above.
4. The data being accessed by the loop must be either **real*4** or **real*8**.
5. Arrays that are referenced in the loop must have only one dimension.
6. Array references on both sides of the assignment statement must refer to the same element. For example,

   ```
   A(i) = A(i) + B(i)
   ```

   will be optimized, but

   ```
   A(i) = A(i+1) + B(i)
   ```

   will not be optimized.

For more information about vector library calls, see the *Domain/OS Call Reference*.

— **-opt 4** also causes the compiler to perform inline expansion of all subprograms expanded with **-opt 3** and some additional subprograms. You can use the **%begin_noinline** and **%end_noinline** compiler directives to selectively disable inline expansion of subprograms. Refer to the listing for "Compiler Directives" in Chapter 4 for a full description of these directives.

If you use the debugger to debug a program that you compiled using **-opt 3** or **-opt 4**, you may be unable to examine the values of some local variables at points in the source code where those variables are not actively in use.

This happens because the compiler assigns the values of variables to machine registers rather than main memory. The optimizer may decide that the main memory location for this variable does not need to be updated, because all uses of the variable in the source program can legally use the value of the variable that is retained in the machine register. In addition, the optimizer may merge some source statements together, or eliminate source statements entirely.

Thus, when you are debugging with these optimizations, you may see what appear to be strange jumps in the control flow of the program. Furthermore, you may be unable to set a breakpoint at a particular source line because the generated code for that source line has been optimized away or merged with the code from another source line.

See the *Domain Distributed Debugging Environment Reference* for more details about the use of the debugger.

When you use full optimization on your FORTRAN program, you should pay special attention to any warning and information messages that are issued during compilation. These messages often point out hidden problems in your program. Sometimes the messages indicate ways that you can make your program more efficient. At other times, the messages may identify subtle bugs that the compiler has detected in your program. Consider the following example:

```
      SUBROUTINE F
      INTEGER I,J,K

      READ(5,*) I,J
      IF ( I .LT. J ) GOTO 10
      K = I
      CALL Z(J,K)
      IF ( K .EQ. I ) THEN
10        K = K + 1
      ENDIF
      END
```

Compilation of this program produces the following warning:

```
**** Informational #929 on Line 9: Assignment eliminated, value
     never used: K
```

The compiler has detected that **K** is not used after the assignment to it at the line labeled 10, and so it eliminated the assignment.

Closer examination of the program, however, shows that the program depends on this assignment—the program assumes that **K** will be statically allocated. Specifically, the program assumes that **I** will be less than **J** the first time the subroutine is called, so **K** will be initialized to the value of **I**. On subsequent calls to the subroutine, the program depends on **K** having the same value on entry to the subroutine as it did on the last invocation of the subroutine. That is, the program assumes that if **I** is not less than **J**, **K** will be incremented. However, Domain FORTRAN dynamically allocates variables that are not in **common** on the run-time stack. Each time the subroutine is invoked, therefore, **K** is reallocated and gets a random value. Thus, unless you compile it with the **-save** option, this program will produce unpredictable results.

The ANSI standard for FORTRAN does not require static allocation of variables that are not in **common**, unless the variables appear in a **save** statement. Therefore, this example, and programs like it, are not standard.

Nonetheless, many FORTRAN programs depend on the assumption that local variables are statically allocated. To obtain consistent behavior from such programs using Domain FORTRAN, we provide the **-save** option. See Section 6.5.30 on **-save** for more information.

Using **-save** turns off most of the optimizations discussed above, regardless of the setting of the **-opt** option. Thus, to obtain the maximum benefit from the optimizations that the compiler provides, you can do one of the following:

- Change your program to ensure that all variables are properly initialized, *or*
- Place variables that require static allocation in **common**, *or*
- List variables that require static allocation in a **save** statement in the routine.

6.5.27 -overlap: Storage Association

The **-overlap** option tells the compiler what assumptions to make about storage associations within the program. According to the ANSI FORTRAN standard, no dummy argument may be modified in its subprogram if it is associated with ("overlaps") any other dummy argument in the same subprogram or with any variable in a common block referenced by the subprogram. For example, if a subprogram is headed

```
SUBROUTINE FOO (A, B)
```

and is referenced by

```
CALL FOO(C, C)
```

then the dummy arguments **A** and **B** become associated with each other and, according to the standard, neither of them may be modified in **FOO** or in any subprogram referenced by **FOO**.

By default, the compiler assumes that the program conforms to the standard—that is, no dummy argument with storage associations is modified in its subprogram. This assumption frees the compiler from certain restraints when optimizing a program.

The **-overlap** option takes one or more of the following arguments, indicating the degree of conformity to the standard:

- **no_dd** (default): No dummy argument that is modified in its subprogram is associated with any other dummy argument to the subprogram. The program conforms to the standard for dummy-to-dummy associations. This setting produces the most optimized code.

- **exact_dd**: If two dummy arguments overlap, then they must overlap by beginning at exactly the same address. Namely, if A and B are arrays and overlap in any way, then the address of the actual parameter passed as A must equal the address of the actual parameter passed as B.

- **dd**: The program does not conform to the standard for dummy–to–dummy associations.

- **no_dc** (default): No dummy argument that is modified in its subprogram is associated with a variable in a common block and no variable in a common block that is modified is associated with a dummy argument. The program conforms to the standard for dummy–to–common associations. This setting produces the most optimized code.

- **exact_dc**: If a dummy argument is associated with a variable in a common block, the address of the actual parameter is identical to the address of the variable in the common block.

- **dc**: The program does not conform to the standard for dummy–to–common associations.

6.5.28 -pic and -ac: Memory Addressing

The -**ac** option is the default.

Refer to Section 6.5.1, which describes both the -**ac** and -**pic** options.

6.5.29 -prasm and -nprasm: Series 10000 Listing

The -**prasm** option is the default if you are using a compiler that generates Series 10000 code and are compiling with the -**exp** option.

The -**prasm** and -**nprasm** options give you control over the format of expanded listings generated when you use the -**exp** option. If you use either option without also specifying -**exp**, they have no effect.

The -**prasm** option tells the compiler to use the Series 10000 assembly language format for the expanded listing. The -**nprasm** option tells the compiler to use an alternate assembly–language format for the expanded listing. Most programmers find this format easier to use when debugging a program.

If you use either option with compilers that generate MC680x0 code, it has no effect.

6.5.30 –save: Static Storage

By default, FORTRAN allocates static storage for local variables cited in **save** or **data** statements (or those equivalenced to such variables), and to variables in **common** areas. All other local variables are stored on the stack, and therefore do not retain their values from one invocation of a subprogram to the next.

The **–save** option forces the compiler to allocate static storage to all variables. Thus, if your program relies on maintaining the values of local variables from one invocation of a subroutine to the next, you should compile with **–save**, or use **save** or **data** statements. (Refer to the lists for "**data**" and "**save**" in Chapter 4.)

Note that the **–save** option turns off most optimizations that would normally be performed. These optimizations are inhibited even if you specify a high optimization level with the **–opt** option. As a result, your program will run slower.

The **–save** option is especially useful when compiling a program that was developed on another system. Unlike Domain FORTRAN, some FORTRAN compilers place all data in static storage, and a program developed on such a compiler may depend on data preserving their values from one invocation of a program unit to the next. If you know that the program you are porting was developed on such a compiler, the safest way to proceed is to use the compiler's **–save** option or a blank **save** statement in each program unit, ensuring static storage for all variables.

Unfortunately, preserving data in static storage results in programs running more slowly than if data were allocated storage on the stack. (One reason for the degraded performance is that the optimizer cannot assign variables in static storage to registers.) If you are concerned with program performance, you may want to take the time to analyze the program and identify the variables that must be preserved in static storage. These are the only variables that must be declared in the **save** statement.

If you have access to a Series 10000 compiler, you can let the compiler do some of the analysis for you by compiling the program with the **–save** option (or with blank **save** statements in every program unit) and **–info** option (refer to Section 6.5.17). The compiler issues diagnostic messages for all scalar variables that must be allocated static storage in order for the program to execute correctly. You need include only these variables (plus any array variables that you have determined must also go in static storage) in the list of arguments for the **save** statements. For information on the **save** statement, refer to the listing for "**save**" in Chapter 4.

6.5.31 –subchk and –nsubchk: Subscript Checking

The **–nsubchk** option is the default.

If you use **–subchk**, the compiler generates additional code at every subscript or substring to check that the subscript or substring is within the declared range of the array. This extra code slows your program's execution speed.

Subscript and substring checking is never done on arrays with assumed dimensions (for example, **iarray(*)**), or on those that have the same upper and lower dimension (for instance, **iarray(1)**).

If you use **-nsubchk**, the compiler does not generate this extra code.

6.5.32 -type and -ntype: Data Type Checking

The **-ntype** option is the default.

When you compile with the **-type** option, Domain FORTRAN issues warning messages if it encounters variable names that have not been explicitly declared with a data type statement. For example, if you compile this program

```
integer*4 sum
sum = 0
do i = 1,10
    sum = sum + i
end do
end
```

with the **-type** option, you get the following message:

```
(00006)         end
**** Warning #80 on Line 6: identifier not explicitly typed I
no errors, 1 warnings in $MAIN, Fortran version n.nn . . .
```

To eliminate the warning, you must explicitly declare the variable **i**.

The **-type** option can help you find keystroke errors. For example, if you are in the habit of explicitly declaring all variables, a compilation with **-type** can show you if you've misspelled a variable name.

6.5.33 -u: Activate Case Sensitivity for Identifiers

When you use the **-u** option, the compiler makes a distinction between uppercase and lowercase characters that you use in identifier names.

For example, suppose you use the **-u** option when you compile the following fragment:

```
logical different_names, DIFFERENT_NAMES
```

The compiler allocates two different variables—one variable is named **different_names**, and the other is named **DIFFERENT_NAMES**.

However, if you do not use the **–u** option, the compiler issues the following warning, because it is case–insensitive for identifiers.

```
(00003)        logical different_names, DIFFERENT_NAMES
**** Warning #85 on Line 3: [nt_names, @DIFFERENT_] redundant type
     declaration
```

Note that the **–u** option does not affect case–sensitivity within strings.

6.5.34 –uc and –nuc: UNIX–compatibility Features

–uc and **–nuc** are complementary options: the **–uc** option is used with **ftn** to turn *on* UNIX–compatibility features, and the **–nuc** option is used with **f77** to turn *off* UNIX compatibility features.

Using **ftn** with the **–uc** option produces the following effects, which are the same as those produced by using **f77** with no options:

- Appending an underscore to subprogram and **common** block names.

  On UNIX systems, the compiler appends an underscore (_) to the name of a **common** block or a FORTRAN procedure that does not start or end with underscore (_) and does not contain a dollar sign ($). The underscore distinguishes the block or procedure from a C procedure or external variable with the same user–assigned name. Furthermore, FORTRAN built–in procedure names have embedded underscores to avoid clashes with user–assigned subroutine names. If you do not use the **–uc** option, it is possible that the names of your subprogram and **common** blocks will be the same as some external names. This is not usually a problem, however, because the linker looks for names in the files you specify before it looks in the global libraries.

- Use of the UNIX version of default filenames.

  When you open a file, the compiler assigns the default filename **fort.***n*, where *n* is the specified unit number. See the listing for "**open**" in Chapter 4 for details about the **open** statement.

- Passing hidden string length values by value rather than by reference.

  Domain FORTRAN passes an implicit string–length argument at the end of any subprogram that explicitly passes a character string. By default, this argument is passed by reference. By specifying the **–uc** option, you can override the default and cause the argument to be passed by value. For information about passing string arguments between FORTRAN and C programs, refer to Section 7.7.3.

Using **f77** with the **–nuc** option produces the following effects, which are the same as those produced by using **ftn** with no options:

- The system does not append an underscore to subprogram and **common** block names.

- If an **open** statement contains the **status** attributes 'new' or 'unknown', the system uses Aegis default filename conventions. The system also uses these conventions if an **open** statement omits these **status** attributes.

- The compiler supplies the name in the format

  FOnnR0.dat

  where *nn* is the logical unit ID number.

- Hidden string-length values are passed by reference rather than by value. This is the default behavior, which you get whether or not you specify the **-nuc** option.

- The system does not interpret escape sequences in strings. For example, the compiler does not interpret '\n' in the string "This is a string.\n" as a newline.

To invoke **f77** with the **-nuc** option, use the **-W0** option, as follows:

$ f77 -c -W0,-nuc test.f

> **NOTE:** You may want to include the -c option (as in the preceding command line) to avoid the unresolved-reference error messages caused by not having UNIX compatibility invoked.

6.5.35 -version: Version of Compiler

If you use the **-version** option, the compiler reports its version number.

When you use the **-version** option, do not include the filename or any other option on the command line. If you do, you will get an error message from the compiler. For example, the following command line shows the correct use of the **-version** option:

$ ftn -version

In response, the compiler issues a message with the format

`Fortran 77 compiler` *variant* `Rev` *n.n.*

where *n.n.* is the version number and *variant* is one of the following:

```
68K
PRISM
68K=>PRISM
PRISM=>68K
```

(Refer to Section 6.2 for information about compiler variants.)

6.5.36 -warn and -nwarn: Warning Messages

The -**warn** option is the default.

If you use -**warn**, the compiler reports all warning messages.

If you use **-nwarn**, the compiler suppresses reporting warning messages (though it does report the total number of warnings that would have been issued).

6.5.37 –xref and –nxref: Symbol Map and Cross References

The **–nxref** option is the default.

The –**xref** option adds a symbol map and cross reference to the listing file. The **–xref** option always generates a listing file at *program_name.lst*, even if you do not explicitly request one with **–l**.

The following shows most of what **–xref** produces when used on the sample program **xref_example**. (Compilation statistics are omitted because they are also produced when you simply use the –l option.)

```
(00001)
(00002) *  This very simple program adds two user-entered integers and prints
(00003) *  the result.
(00004)       program xref_example
(00005)
(00006)       integer*4 first, second, result
(00007)       logical   again
(00008)       again = .true.
(00009)
(00010)       do while (again)
(00011)           print 10
(00012) 10        format (' Enter two integers: ', $)
(00013)           read (*, *) first, second
(00014)           result = first + second
(00015)           print *, 'The answer is: ', result
(00016)           call find_answer(again)
(00017)       enddo
(00018)
(00019)       end
.
.     {Compilation Statistics are omitted}
.
```

Symbol Name	Type	Data Type	Address Offset	Storage	Lines on Which Symbol Occurs			
10	label				00011	00012D		
again	var	l*4	000000	save	00007S	00008M	00010	00016A
find_answer	subr				00016			
first	var	i*4	000004	save	00006S	00013M	00014	
result	var	i*4	00000C	save	00006S	00014M	00015	
second	var	i*4	000008	save	00006S	00013M	00014	

```
(00020)
(00021) *********************************************************************
(00022) *  Subroutine to ask user whether he/she wants to continue.
(00023)       subroutine find_answer(repeat)
(00024)       logical   repeat
(00025)       character answer
(00026)
(00027)       print 100
(00028) 100   format (/, 'Again? (Y or N) ', $)
(00029)       read (*, '(A1)') answer
(00030)       if ((answer .eq. 'n') .or. (answer .eq. 'N')) repeat = .false.
(00031)
(00032)       end
.
.     {Compilation Statistics are omitted}
.
```

Symbol Name	Type	Data Type	Address Offset	Storage	Lines on Which Symbol Occurs		
100	label				00027	00028D	
answer	var	char	FFFFFE	stack	00025S	00029M	00030
find_answer	subr				00023S		
pfm_$cleanup	exfunc				00019S		
repeat	var	1*4	000008	argument	00023S	00024S	00030M

This program is available online and is named **xref_example**.

The **-xref** output begins with a line-by-line listing of the source program's main routine. Compilation statistics (omitted here) follow. Then **-xref** gives information about each of the symbols the program uses. That information is listed under the boldface column headings. The headings appear here to help you understand the information given but do not appear in the actual **-xref** listing.

The columns give the following information:

Symbol Name Lists each variable, array, subroutine, function, and label appearing in this section of the program.

Type Tells what type of symbol this is (for example, **var** for variable, **subr** for subroutine, and **intfunc** for intrinsic function)

Data Type If the symbol is an array or variable, this column lists its data type. (For instance, **l*4** means **logical*4**, **i*4** means **integer*4**, and char means **character**)

Address Offset For the first variable the program accesses, this number is zero. For subsequent variables and arrays, this is a hexadecimal number that tells how many bytes away from the beginning address a variable or array name is stored. If the variable or array is part of a **common** block, the offset is calculated from the beginning of the common block.

Storage Describes the type of storage allotted for this symbol. The possible values are:

save	for variables and arrays local to the program unit that have been given static storage
stack	for variables and arrays given stack (dynamic) storage
argument	for dummy argument names
/common_block_name/	for variables and arrays that are members of *common_block_name*

Lines on Which Symbol Appears — Lists the line numbers on which a symbol appears, and, optionally, one letter describing what happens to the symbol on that line. These line numbers are those that the listing file generates. The letters mean the following occurred on the indicated line:

- **A** the symbol was used as an argument
- **D** the symbol was defined (for example, in a data statement if it was a variable or array)
- **M** the symbol's value was modified
- **S** the symbol was specified, or declared, in a data type declaration statement

After -**xref** lists all the information about the main program unit, it lists the same information for each individual subprogram present. In the example above, there is only one subprogram, but if there were more, all would be listed in the order in which they were referenced in the main program unit.

> NOTE: At SR10, you cannot use both -**exp** and -**xref** on the same command line.

6.5.38 -xrs and -nxrs: Register Saving

The -**xrs** option is the default.

This option controls whether or not the compiler assumes that the contents of registers are saved across a call to an external routine. Some routines do not preserve data registers D2 through D7, address registers A2 through A4, and floating-point registers FP2 through FP7. If you use -**xrs**, the compiler assumes these registers are saved; if you use -**nxrs**, it does not assume these registers are saved.

In either case, the compiler always saves register contents when it enters a routine and restores those contents to the registers when it exits the routine.

This option is used primarily when your program contains calls back to subprograms in an installed library compiled with pre-SR9.5 compilers. In such a case, you should isolate the portion of your new code that calls the older subprograms, and separately compile that new code with the -**nxrs** option.

6.5.39 -zero and -nzero: Initialization to Zero

-**nzero** is the default.

The -**zero** option causes Domain FORTRAN to initialize to zero the following:

- All **common** blocks declared in the program unit, except for those variables and arrays in **common** blocks that are explicitly initialized by data statements

- All statically allocated local variables in the program unit, except those explicitly initialized with data statements

You can use the **-zero** option in conjunction with **-save**, which allocates static storage to all local variables. Note that the **-zero** option does not initialize local variables that have been allocated stack storage. Also note that compiling with **-zero** may increase the time between invocation of the program and execution of its first instruction.

6.6 Linking in a Domain Environment

There are two commands that enable you to link object modules to form an executable image. The link editor (**ld**) is a standard UNIX link editor with some Domain enhancements. The **bind** command is the traditional Aegis binder. You can use either command regardless of whether the modules were compiled with **ftn** or **f77**.

6.6.1 The Link Editor (ld)

Use the link editor, **ld**, to combine several object modules into one executable program. The input object modules can come from the following sources:

- Libraries created by **ar** (a UNIX archiver)
- Libraries created by **lbr** (the Aegis librarian)
- Object modules created by the Domain/C, Domain Pascal, or Domain FORTRAN compilers, or the Domain 680xx assembler.
- Object modules previously created by **ld**.
- Object modules created by **bind** (the Aegis binder).

The primary purpose of **ld** is to resolve external references. If there are any unresolved external references, **ld** will report them. You can also use **nm**, a UNIX utility, to perform a check of resolved and unresolved global symbols.

Note that **ld**'s output can either be executed (assuming that there is a start address) or used as input for a further **ld** run. Domain FORTRAN supports the **ld** options listed in Table 6-9. When you use any of these options on an **f77** command line, the compiler passes them along to the link editor. For syntax details on **ld** and its options, refer to the *BSD Command Reference* and the *SysV Command Reference*.

*Table 6-9. Link Editor (**ld**) **f77** Compiler Options*

Option	What the Option Tells the Compiler
-a	Produce an object file for execution. This is the default. Use **-r** to retain relocation information in the object module. If you specify both **-a** and **-r**, the link editor will retain relocation information for all data except common symbols, which will be allocated.
-l*x*	Search the library named **lib***x***.a**, in addition to the default libraries. Libraries are searched in the order that they appear on the command line. By default, the link editor searches for libraries first in the directory **/lib**, then in **/usr/lib**.
-o *output*	Name the final output file *output*. By default, *output* is **a.out**. If you specify a different name, the system leaves any existing **a.out** file undisturbed. This is similar to the Aegis **-b** option described in Section 6.5.3.
-r	Retain relocation entries in the output object module. Relocation entries must be preserved if the object file will be specified in a future **ld** or **bind** command. **-a** is the default.
-s	Strip line number entries and symbol table information from the output object file. This option is useful if you want to reduce the size of the object module. Note, however, that removing this information from a program makes it impossible to debug the program with a source-level debugger.
-t	Suppress warnings about multiply defined symbols that do not have the same size.
-u *symname*	Enter *symname* as an undefined symbol in the symbol table. This option is useful if you are using the **cc** command to load a library. The symbol table is initially empty and needs an unresolved reference to force **ld** to load the first routine.
-x	Do not preserve local symbols in the output symbol table; enter external and static symbols only. This saves space in the object module, but still enables the link editor to resolve global references.

(Continued)

*Table 6-9. Link Editor (**ld**) **f77** Compiler Options (Cont.)*

Option	What the Option Tells the Compiler
☆ -A *opt*	Identify Domain extensions to **ftn**. *opt* may be **cpu**, **run**[type], or **sys**[type], among others. For more information about the -A option, refer to Table 6-3, which lists all of the **f77** command options, including -A. For a complete list of Domain/OS extensions that can be passed to **ld** with the -A option, refer to the *BSD Command Reference* or the *SysV Command Reference*.
-L*dir*	Change the search path for libraries. By default, the compiler looks for **lib*x*.a** libraries first in the directory **/lib**, then in **/usr/lib**. This option allows you to specify a different directory, *dir*, before searching these standard directories. This is useful if you have different versions of a library and you want to specify which one the link editor should use. Note that this option is only effective if it precedes a **-l** option.
-V	Output a message giving information about the version of **ld** being used.
☆ The -A **cpu** option replaces -M, which is obsolete.	

6.6.2 The bind Command

The format for the bind command is as follows:

$ **bind** *pthnm1* [*...pthnmN*] [*option1* [*...optionN*]]

A *pthnm* must be the pathname of an object file (created by a compiler) or a library file (created by the librarian). Your **bind** command line must contain at least one pathname.

Your **bind** command line can also contain zero or more binder options, the most important of which is **-b**. If you use the **-b** option, the binder generates an executable object file. If you forget to use the **-b** option, the binder won't generate an output object file.

For example, suppose you write a program consisting of three source code files: **test_main.ftn**, **mod1.ftn**, and **mod2.ftn**. To compile the source code, you issue the following three commands:

```
$ ftn test_main
$ ftn mod1
$ ftn mod2
```

The compiler creates **test_main.bin, mod1.bin,** and **mod2.bin**. To create an executable object, bind the three together with a command like the following:

```
$ bind test_main.bin mod1.bin mod2.bin -b bound_file
```

This command creates an executable object file in filename **bound_file**.

> **NOTE:** At SR10 the compiler generates an object format that is an extended version of the COFF (Common Object File Format) standard. The loader retains knowledge of how to load pre-COFF objects; however, it is not possible to bind old and new modules together. Be aware that COFF object files will not run on pre-SR10 workstations.

Refer to the *Domain/OS Programming Environment Reference* for a complete discussion of the binder and its options.

6.7 Archiving and Using Libraries

A **library file** is a special file created by the librarian, consisting of one or more object modules collected together for easy access by the binder. Use the archiver, **ar**, to create and update library files. Once created, a library file can be used as input to the link editor, **ld**. As with most linkers, **ld** will optionally bind only those modules in a library file that resolve an outstanding external reference. For syntax details on **ar** and its options, see the *Domain/OS Programming Environment Reference.*

You can also create library files with the **lbr** utility, which is detailed in the *Domain/OS Programming Environment Reference*.

> **NOTE:** At SR10, the **lbr** utility handles only objects generated by SR10 compilers, SR10 linkers (**bind** or **ld**), or SR10 archivers (**lbr** or **ar**). The **lbr** utility generates library files in the form of UNIX archive (**ar**) files. For compatibility, we provide a version of **lbr** that handles objects created between SR9.5 and SR9.7—the **lbr2ar** command. You can use **lbr2ar** to convert pre-SR10 **lbr** libraries. See the *Domain/OS Programming Environment Reference* for details about using **lbr2ar**.

In addition to library files, the Domain system also supports run-time (dynamic) libraries, which are detailed in the *Domain/OS Programming Environment Reference*.

On some operating systems, you must bind language libraries and system libraries with your own object files. On the Domain system, there is no need to do this as the loader binds them automatically when you execute the program.

6.8 Executing a Program

To execute a program, enter its full pathname (including any suffixes). For example, to execute an object file, enter

$ *executable_file*

The operating system searches for the file according to its usual search rules, then calls the **loader utility**. The loader utility is user transparent. It resolves external symbols in your executable object file with global symbols in the language and system libraries. Then it executes the program.

By default, standard input and standard output for the program are directed to the keyboard and display, respectively. You can redirect standard input and output by using the shell's redirection notation. For example, to redirect standard input when you invoke your program, type

$ *executable_file <file_to read_from*

The < character redirects standard input so that *executable_file* reads data from *file_to read_from*. You can redirect standard output in a similar fashion, for example:

$ *executable_file >file_to_write_to*

This command line uses the character > to redirect standard output for *executable_file* so that it writes data to *file_to_write_to*.

6.9 Debugging Programs in a Domain Environment

The Domain systems support two source-level debuggers—the Domain Distributed Debugging Environment and **dbx**. The following sections describe them briefly.

6.9.1 The Domain Distributed Debugging Environment Utility

The Domain Distributed Debugging Environment is a screen-oriented debugger that provides all the features of other high-level language debuggers. To prepare a file for debugging with the Domain Distributed Debugging Environment, you do not have to do anything special at bind time, but you do have to compile with one of the following options: **-db**, **-dba**, or **-dbs**. The **-db** option provides minimal debugger preparation; **-dba** and **-dbs** provide full debugger preparation. For details about these options, refer to Section 6.5.9.

For complete details, refer to the *Domain Distributed Debugging Environment Reference* manual.

6.9.2 The dbx Utility

dbx is a traditional Berkeley UNIX source language debugger. Although it is usually available only on 4.1 BSD and 4.2 BSD systems, the Domain version is available regardless of what environment you are running.

The command syntax for invoking **dbx** is:

$ dbx [*options*] [*object_file* [*coredump*]]

where *object_file* is the name of the program you want to debug.

For complete details about the **dbx** utility, refer to the *Domain/OS Programming Environment Reference*.

6.10 Program Development Tools

Domain/OS supports several programming tools that aid in program development, debugging, and source management. This section describes briefly the tools listed below. The description for each tool includes information about where to find further information. See the Preface for a complete listing of related manuals.

- Traceback (**tb**)
- DSEE (Domain Software Engineering Environment)
- Open Dialogue and Domain/Dialogue
- Domain/PAK (Domain Program Analysis Kit)

6.10.1 Traceback (tb)

If you execute a program and the system reports an error, you can use the **tb (traceback)** utility to find out what routine triggered the error. To invoke **tb**, enter the command

$

immediately after a faulty execution of the program.

For example, suppose you run a program named **test_tb** that asks for integer input. The source code for **test_tb** is as follows:

```
      program test_tb
****************************************
      integer*4 month
****************************************
      print *, 'please enter an integer'
      read *, month
      print *, 'your number is ', month
      end
```

However, when you run **test_tb**, you give it a real number instead of an integer, and the program terminates with an error. If you then invoke **tb**, the whole sequence might look like the following::

```
$ test_tb.bin
 please enter an integer
8.14
?(ftnlib) Using Unit 5 connected to "/sys/node_data/crp_mbx001" for for-
matted sequential access -
Improper character in input data (library/IO transfer)
?(sh) ".../test_tb.bin" - Improper character in input data (library/IO
transfer)
$ tb
Process        2418 (parent 2377, group 2377)
Time           90/04/15.16:01(EDT)
Program        /test_tb.bin
Status         05050003: Improper character in input data (library/IO
               transfer)
In routine     "pfm_$error_trap" line 142
Called from    "ftn_$error" line 209
Called from    "fio_$error" line 409
Called from    "fio_$collect_integer" line 1370
Called from    "fio_$xfer_i4" line 1812
Called from    "test_tb" line 6
Called from    "PM_$CALL" line 151
Called from    "pgm_$load_run" line 591
$
```

After listing identifying information about the process, time, and program, the **tb** utility reports the error status, which in this case is:

```
Status         05050003: Improper character in input data (library/IO
transfer)
```

Then **tb** shows the chain of calls leading from the routine in which the error occurred all the way back to the main program block. For example, routine **pfm_$error_trap** reported the error. **pfm_$error_trap** was called by the **ftn_$error** routine. Since all of the routines except **test_tb** are system routines, it is probable that the error occurred at line 6 of the **test_tb** routine.

See the *Domain/OS Programming Environment Reference* for details about the **tb** utility.

6.10.2 The DSEE Product

The DSEE (Domain Software Engineering Environment) product is a support environment for software development. DSEE helps engineers develop, manage, and maintain software projects; it is especially useful for large-scale projects involving several modules and developers. You can use DSEE for:

- Source code and documentation storage and control
- Audit trails
- Release and Engineering Change Order (ECO) control
- Project histories
- Dependency tracking
- System building

The DSEE product provides sophisticated enhancements to the traditional program development cycle (compiling, building libraries, binding, debugging) described in this chapter. For information on the optional DSEE product, see *Engineering in the DSEE Environment*.

6.10.3 Open Dialogue and Domain/Dialogue

Open Dialogue and Domain/Dialogue are tools for defining the user interface to an application program. Open Dialogue can be used on both Apollo and non-Apollo workstations and is layered on UNIX and the X Window System. Open Dialogue also allows you to create interfaces that are compliant with the Open Software Foundation (OSF) interface design standards presented in the *MOTIF Style Guide*. Domain/Dialogue can be used only on Apollo workstations and is layered on Domain/OS and Graphics Primitives Resource (GPR).

Both products enable you to separate the interface definition from the application code. For the user interface, this separation means that you can

- Focus more time and attention on the interface than is possible when it is intertwined with the application code.
- Develop modular interfaces that are consistent in design from application to application because they are developed with the same set of tools.
- Use an iterative approach to interface design. A program's user interface can be rapidly prototyped and modified without affecting the application code. Successive testing and refinement are relatively easy, making it possible to fine-tune the interface.
- Develop multiple user interfaces to a program, allowing users to choose the style of interaction with which they feel most comfortable.

For the application, this separation means that you can

- Write less code. Because Open Dialogue and Domain/Dialogue handle interactions with the user, the application designer does not have to provide the code for doing so.
- Achieve a modular approach to writing code that promotes phased and iterative application development independent of user interface development.

For details about Domain/Dialogue, see the *Domain/Dialogue User's Guide*. For details about Open Dialogue, see the following:

- *Open Dialogue Reference*
- *Creating User Interfaces with Open Dialogue*
- *MOTIF Style Guide*
- *Customizing Open Dialogue*

6.10.4 Domain/PAK

Domain/PAK (Domain Performance Analysis Kit) is a collection of the following three programs:

- DSPST (Display Process Status) looks at the relative use of CPU time by several processes at the system level.
- DPAT (Domain Performance Analysis Tool) is an interactive tool that looks at the performance of programs, including I/O, paging, and system calls, at the procedure level.
- HPC (Histogram Program Counter) looks at the performance of compute-bound procedures at the statement level.

Domain/PAK enables you to analyze the performance of a program. It is particularly useful for isolating bottlenecks. See *Analyzing Program Performance with Domain/PAK* for more details about Domain/PAK.

6.11 Program Development Using the Network File System (NFS)

Domain FORTRAN is fully compatible with the Network File System (NFS). You can redirect the binary output of the FORTRAN compiler to a file on a remote node that you have accessed using NFS, and you can then run the program on the remote node. In order to use this feature of Domain FORTRAN, you must have Domain NFS installed on your system.

For example, suppose you issue the following NFS **mount** command to gain access to a remote node:

```
$ [illegible]
```

This command, which you can issue in any shell, gives you access to the entire directory structure of the remote node **faraway**. You can access this directory structure as if it were a local directory named **/other_node**. Note that there is a space separating the two slashes preceding **other_node**.

To compile the program **test.f** or **test.ftn** and place it in the directory **/tmp** on the remote node, issue the **f77** command

```
$ [illegible]
```

or the **ftn** command

```
$ [illegible]
```

You can also place the source listing in a directory on the remote node by using the –l pathname option. Then you can run the program as follows:

```
$ [illegible]
```

You can also use the remote node directory as your working directory. If you do so, compiler options that use the name of the current working directory work as usual. For instance, you can use the **ftn –l** option to generate a listing file, as shown in the following sequence of commands (the example assumes you are in a UNIX shell):

```
% [illegible]
% [illegible]
% [illegible]
% [illegible]
.
.
.
%
.
.
.
test.lst
```

For more information about Domain NFS, refer to *Using NFS on the Domain Network*.

Chapter 7

Cross–Language Communication

This chapter describes how to call Domain Pascal and Domain/C subprograms from a Domain FORTRAN program and how to share data between a Domain FORTRAN program and a Domain Pascal or Domain/C program. Because Domain system routines are, for the most part, written in Domain Pascal, the information in this chapter also applies to invoking system routines from Domain FORTRAN. Briefly, this chapter covers the following topics:

- Understanding data type agreement of Domain FORTRAN, Domain Pascal, and Domain/C
- Calling Domain Pascal subprograms from a Domain FORTRAN program
- Calling Domain/C subprograms from a Domain FORTRAN program
- System service routines

7.1 Calling a Pascal Subprogram from FORTRAN

Domain FORTRAN permits you to call subprograms written in Domain Pascal source code. To accomplish this, perform the following steps:

1. Write source code in Domain FORTRAN that calls a subprogram. Compile it with the Domain FORTRAN compiler. Domain FORTRAN creates an object file.
2. Write source code in Domain Pascal. Compile it with the Domain Pascal compiler. Domain Pascal creates an object file.
3. Bind the object file(s) created by the FORTRAN compiler with the object file(s) created by the Pascal compiler. The binder creates one executable object file.
4. Execute the object file as you would execute any other object file.

This chapter describes Steps 1 and 2. For information on Steps 3 and 4, see Sections 6.6 and 6.8.

> **NOTE:** The following sections explain how to call Domain Pascal from Domain FORTRAN. If you want to learn how to call Domain FORTRAN from Domain Pascal, see the *Domain Pascal Language Reference*.

7.2 Data Type Correspondence for FORTRAN and Domain Pascal

There is really no difference between making a call to a Pascal function or procedure and making a call to a FORTRAN subprogram. However, before passing data between Domain FORTRAN and Domain Pascal, you must understand how Domain FORTRAN data types correspond to Domain Pascal data types. Table 7–1 lists these correspondences.

Table 7–1. Domain FORTRAN and Domain Pascal Data Types

Domain FORTRAN	Domain Pascal
byte	**char**
integer*2	**integer, integer16**
☆ **integer, integer*4**	**integer32**
real, real*4	**real, single**
double precision, real*8	**double**
character*1	**char**
logical*1	**boolean**
logical*2	**[word] boolean**
☆☆ **logical, logical*4**	**[long] boolean**
set emulation calls	**set**
complex, complex*8 **complex*16,** **double complex**	user–declared record
array (with restrictions)	**array**
pointer statement	**pointer**

☆If you compile your FORTRAN source code with the **–i*2** switch, the equivalent in Pascal for the **integer** variables is the same as the equivalent for **integer*2**, namely **integer** and **integer16**.

☆☆If you compile your FORTRAN source code with the **–l*1** switch, the equivalent in Pascal for the **logical** variables is the same as the equivalent for **logical*1**, namely **boolean**. If you compile your FORTRAN source code with the **–l*2** switch, the equivalent in Pascal for the **logical** variables is the same as the equivalent for **logical*2**, namely **[word]boolean**.

The **integer**, **real**, and **character** data types in both languages correspond very well to each other. For example, Domain FORTRAN's **integer*2** data type is identical to Domain Pascal's **integer16** data type, and a **real** in one language is exactly the same as a **real** in the other.

There is a difference in what the keyword **integer** means in the two languages. By default, **integer** in Domain FORTRAN is a 4-byte entity, while in Domain Pascal, **integer** is two bytes. To avoid any confusion, you should use the specific integer data types (**integer*2**, **integer*4**, **integer16**, and **integer32**) rather than the generic **integer**.

7.2.1 Logical and Boolean Correspondence

Domain FORTRAN's **logical**, **logical*1**, **logical*2**, and **logical*4** data types, on the one hand, and Pascal's **boolean** data type, on the other, serve identical purposes—namely, to hold a value of true or false. Furthermore, **logical*1** and **boolean** each take up one byte of memory. Thus Domain FORTRAN's **logical*1** type and Pascal's **boolean** type correspond exactly. However, Domain FORTRAN's **logical*2**, **logical*4**, and **logical** types take up more than one byte of memory. To make these FORTRAN data types match up with Pascal **boolean** types, you should use size attributes to create the corresponding Pascal types shown in Table 7-1. See the *Domain Pascal Language Reference* for details about size attributes.

7.2.2 Simulating the Complex Data Type

Unlike Domain FORTRAN, Domain Pascal doesn't support a predeclared **complex**, **complex*8**, **complex*16**, or **double complex** data type. However, you can easily declare Pascal record types that emulate these FORTRAN data types as follows:

```
type
   complex =          record
                      r         : single;
                      imaginary : single;
                      end;

   complex8 =         record  {same as complex}
                      r         : single;
                      imaginary : single;
                      end;

   complex16 =        record
                      r         : double;
                      imaginary : double;
                      end;

   double_complex = record                        {same as complex16}
                      r         : double;
                      imaginary : double;
                      end;
```

7.2.3 Array Correspondence

Single-dimensional arrays of the two languages correspond perfectly; for example:

In Domain FORTRAN		*In Domain Pascal*
`character*10`	`x`	`x = Array[1..10] of char`
`integer*2`	`x(50)`	`x = Array[1..50] of integer16`
`real*8`	`x(20)`	`x = Array[1..20] of double`
`logical*1`	`x(10)`	`x = Array[1..10] of boolean`

Multidimensional arrays in the two languages do not correspond very well. The tricky part is that Domain FORTRAN represents multidimensional arrays differently from Domain Pascal. In Domain FORTRAN, the leftmost element varies fastest.

For example, Domain FORTRAN represents the six elements of an array defined this way

```
integer*4 my_array(2,3)
```

in the following order:

```
1,1
2,1
1,2
2,2
1,3
2,3
```

However, the rightmost element varies fastest in Domain Pascal arrays. Therefore, Domain Pascal represents the six elements of an array defined this way

```
type
   my_array = array[1..2, 1..3] of integer32;
```

in the following order:

```
1,1
1,2
1,3
2,1
2,2
2,3
```

Obviously this can lead to confusion if you pass a multidimensional FORTRAN array as an actual argument to a Pascal dummy argument. However, there is a way to avoid this confusion. Simply declare the array dimensions of the Pascal argument in reverse order. For example, instead of declaring

```
type
   my_array = array[1..2, 1..3] of integer32;
```

declare

```
type
   my_array = array[1..3, 1..2] of integer32;
```

Following are two more examples:

Actual Argument in FORTRAN	Dummy Argument in Pascal
`real*4 x(10,5)`	`x = array[1..5, 1..10] of real;`
`real*4 x(2,3,4)`	`x = array[1..4, 1..3, 1..2] of real;`

7.3 Sharing Data Between Domain Pascal and Domain FORTRAN

There are two ways to pass data between a Domain FORTRAN program and a Domain Pascal function or procedure. You can either establish a common section of memory for sharing data or you can pass the data as an actual argument to a subprogram. The following paragraphs demonstrate the first method. Section 7.4 demonstrates the second method.

If you create a named **common** area in your Domain FORTRAN program, and then create a named section in your Domain Pascal subprogram with the same name, the binder establishes a section of memory for sharing data.

For example, suppose you want both a Domain FORTRAN program and a Domain Pascal routine to access two variables, a 2-byte integer and an 8-byte double-precision real number. In the FORTRAN program, you can declare the two variables as follows:

```
integer*2 count
real*8    stress_factor
common /my_sec/ stress_factor, count
```

If you want the value of these two variables to be accessible from the Domain Pascal subprogram, you can declare them as follows in the Pascal subprogram:

```
var (my_sec)
   stress_factor : double;
   count         : integer16;
```

Remember to preserve the same order of variable declaration in the **var** declaration part that you did in the **common** statement. For example, your run-time results are unpredictable if you write your **var** declaration part as shown here:

```
var (my_sec)                                   {Wrong!}
   count         : integer16;
   stress_factor : double;
```

7.4 Calling Domain Pascal Functions and Procedures

This section demonstrates how to call a Domain Pascal function or procedure from a Domain FORTRAN program. Calling Pascal from FORTRAN is relatively easy. The two languages' data types match up fairly well, and Section 7.2 describes ways to remedy mismatches that can occur.

There are some differences between the languages in the way they pass arguments. Domain FORTRAN always passes arguments by reference, while Domain Pascal passes them by reference unless a routine heading contains the **val_param** or **C_param** option. Make sure any Pascal routine you want to call from FORTRAN does not contain the **val_param** or **C_param** option.

7.4.1 Calling a Function

The following examples show a FORTRAN program that calls the Pascal function listed after it. Note that Domain FORTRAN's **real*4** data type corresponds perfectly to the **real** data type of Domain Pascal.

```
*  This program calls a Pascal function that calculates the hypotenuse
*  of a right triangle when a user supplies the lengths of two sides.

* NOTE: You must also obtain the Pascal program named "hypot.pas"
*       After compiling ftn_to_pas_hypot and hypot, you must bind
*       them together.

      program ftn_to_pas_hypot

*  The external statement lets FORTRAN know that this subprogram is
*  not in this program unit, but the statement itself is optional.
      external hypot
      real*4 side1, side2, result, hypot

      print 10
10    format (' Enter lengths for the two sides of the triangle: ', $)
      read *, side1, side2
      result = hypot(side1,side2)
      print *, 'The triangle''s hypotenuse is: ', result
      end
```

```
{**************************************************************************}
module hypot;
{ This is a Pascal function for calculating the hypotenuse of a right }
{ triangle.  You don't have to do anything special to make the file   }
{ callable by FORTRAN.                                                  }

{ NOTE: You must also get the FORTRAN program named "ftn_to_pas_hypot"}
{       After compiling ftn_to_pas_hypot and hypot, you must bind     }
{       them together.                                                  }

function hypot(in out leg1,leg2 : real) : real;
begin
hypot := sqrt(sqr(leg1) + sqr(leg2));
end;

begin
hypot := sqrt(sqr(leg1) + sqr(leg2));
end;
```

These programs are available online and are named **ftn_to_pas_hypot** and **hypot**. Following is a sample run of the bound program.

```
Enter lengths for the two sides of the triangle: 3 4
The triangle's hypotenuse is:  5.000000
```

7.4.2 Calling a Subroutine

A **function** in FORTRAN corresponds to a **function** in Pascal. A **subroutine** in FORTRAN corresponds to a **procedure** in Pascal. In the following examples, **hypot** changes from a function to a procedure, and the FORTRAN program changes to reflect that it is calling what it considers to be a subroutine. Note that the FORTRAN program could actually be calling a FORTRAN subroutine. There's nothing in the program that designates the language in which the called subprogram is written.

```
*  This program calls the external Pascal procedure hypot_proc
*  to calculate the hypotenuse of a right triangle when a user
*  supplies the lengths of two sides.

*  NOTE: You must also obtain the Pascal program named
*        "hypot_proc." After compiling ftn_to_pas_hypot2 and
*        hypot_proc, you must bind them together.

      program ftn_to_pas_hypot2
      external hypot_proc
      real*4 side1, side2, result

      print 10
10    format (' Enter lengths for the two sides of the triangle: ', $)
      read *, side1, side2
```

```
      call hypot_proc(side1,side2, result)
      print *, 'The triangle''s hypotenuse is: ', result
      end

{*****************************************************************}
module hypot_proc;
{ This is a Pascal procedure for calculating the hypotenuse of a}
{ right triangle.                                               }

procedure hypot(in out leg1,leg2 : real;
                out    answer    : real);
begin
answer := sqrt(sqr(leg1) + sqr(leg2));
end;
```

These programs are available online and are named **ftn_to_pas_hypot2** and **hypot_proc**.

7.4.3 Passing Character Arguments

In this section we describe how to pass character strings from Domain FORTRAN to Domain Pascal.

If you pass a character string from Domain FORTRAN to Domain Pascal, you should add an extra argument to the Pascal subprogram's list. This is because FORTRAN has an implicit string-length argument at the end of any subprogram call that contains a character string as an actual argument. Suppose your FORTRAN program includes the following:

```
character*10 first_name
        . . .
call change_name(first_name)
```

Your Pascal subprogram heading should include an "extra" dummy argument for the length of **first_name**; for example:

```
type
    name = array[1..10] of char;
          . . .
procedure change_name(in out first_name : name;
                      in            len : integer16);
```

The length argument must be of type **integer16** because FORTRAN's implicit length argument is an **integer*2**.

If you send multiple strings to Domain Pascal and you include the implicit length arguments in the Pascal argument list, the length arguments must always appear at the end of the subprogram heading. That is, it is not correct to list them in the order **string1, len1, string2, len2**.

For instance:

```
*  FORTRAN program fragment
      character*10 first_name
      character    middle_initial
      character*20 last_name
          . . .
      call process_name(first_name, middle_initial, last_name)
          .
          .
          .
{ Pascal procedure fragment }
type
    fn = array[1..10] of char;
    ln = array[1..20] of char;

{ Here's the subprogram heading }
procedure process_name  (in out       first_name : fn;
                         in out   middle_initial : char;
                         in out        last_name : ln;
                         in      len1, len2, len3 : integer16);
{ Note that the the strings first_name, middle_initial, and last_name }
{ are all listed in the subprogram heading before the lengths of the  }
{ strings. }
```


7.4.4 Passing a Mixture of Data Types

The following Domain FORTRAN program and Domain Pascal procedure demonstrate passing a variety of data types, including a character string.

```
*  This program demonstrates passing arguments of several different
*  data types to a Pascal procedure.
*  NOTE: You must also obtain the Pascal program named "mixed"
*        After compiling ftn_to_pas_mixed and mixed, you must bind
*        them together.

      program ftn_to_pas_mixed
      external mixed_types
      integer*2    i, j, my_array(2,4)
      integer*4    carbon_14_age
      logical      the_truth
      character*15 name
      complex      x
      data carbon_14_age, the_truth, name, x
     +     /250000, .true., 'Hercule Poirot', (4.53,5.2E-2)/
      do i = 1,2
          do j = 1,4
              my_array(i,j) = i*j
          enddo
      enddo
```

```
      print 10
10    format (/, 'Before calling Pascal, here are the values:', /)
      call print_vals(my_array, carbon_14_age, the_truth, name, x)
      call mixed_types(my_array, carbon_14_age, the_truth, name, x)
      print 20
20    format (/, 'After calling Pascal, here are the values:', /)
      call print_vals(my_array, carbon_14_age, the_truth, name, x)
      end
************************************************************************
*  This subroutine simply prints out the values of the variables.
      subroutine print_vals(a, b, c, d, e)
      integer*2    a(2,4), i, j
      integer*4    b
      logical      c
      character*15 d
      complex      e
      print *, 'The array contains:'
      do i = 1,2
          print *, (a(i,j), j = 1,4)
      enddo
      print *, 'Age = ', b
      print *, 'The logical variable = ', c
      print *, 'The name = ', d
      print *, 'The complex variable = ', e
      end

{**********************************************************************}
module mixed;
{  This Pascal procedure assigns new values to arguments that a FORTRAN}
{  program passed in.  It demonstrates how to pass a variety of data   }
{  types.  Notice that there is an "extra" argument in the procedure   }
{  heading; FORTRAN's implicit length argument for character strings.  }

type
    full_name = array[1..15] of char;
    four_by_two = array[1..4,1..2] of integer16;
    complex = record
              r         : real;
              imaginary : real;
              end;

procedure mixed_types (in out   a : four_by_two;
                       in out   b : integer32;
                       in out   c : boolean;
                       in out   d : full_name;
                       in out   e : complex;
                       in     len : integer16);     {"Extra" argument.}
var
    i, j : integer16;

begin
for i := 1 to 4 do
```

```
    begin
    for j := 1 to 2 do
        a[i,j] := a[i,j] * 100;
    end;

b := 500000;
c := false;
d := 'Jane Marple';
e.r := 2.333;
e.imaginary := 4.111;

end;
```


These programs are available online and are named **ftn_to_pas_mixed** and **mixed**. If you run the bound program, you get the following output.

```
Before calling Pascal, here are the values:

 The array contains:
 1 2 3 4
 2 4 6 8
 Age =  250000
 The logical variable =  T
 The name = Hercule Poirot
 The complex variable =  (4.530000,5.2000000E-02)

After calling Pascal, here are the values:

 The array contains:
 100 200 300 400
 200 400 600 800
 Age =  500000
 The logical variable =  F
 The name = Jane Marple
 The complex variable =  (2.333000,4.111000)
```

7.4.5 Passing Pointers

Domain FORTRAN's **pointer** statement gives you access to the pointers returned by programs written in other languages. When a routine written in another language returns a pointer, the pointer's value (that is, the address to which it is pointing) is assigned to the FORTRAN variable you declared with the **pointer** statement. The following example demonstrates that access. (See the listing for **pointer** in Chapter 4 for more information on the **pointer** statement and its syntax.)

Most of the work takes place in the Pascal procedure **pass_point**. That procedure creates a linked list and loads data into the list's elements. The FORTRAN program prints out the data. Notice that the way to tell FORTRAN that you want the pointer to point to the next item in the list is simply to make a straightforward assignment. In this case, it's the line

```
ptr = next
```

Following are the two program units:

```
*  This program calls a Pascal procedure, pass_point, which builds
*  a linked list and loads data into it.  After the data has been
*  loaded, Pascal sends back a pointer to the first element in the
*  list.  This program then walks through the list, printing out the
*  data in each element.
*  NOTE: You must also obtain the Pascal program named "pass_point."
*        After compiling ftn_to_pas_ptr and pass_point, you must bind
*        them together.
      program ftn_to_pas_ptr
      integer*4 ptr, next
      character*10 line
      pointer /ptr/ line, next
      call pass_point(ptr)
      print 10
10    format (/, ' The linked list contains:')
      do while (ptr .ne. 0)    {When the pointer is zero, that's    }
          print *, line        {the end of the list.                }
          ptr = next           {Reset pointer to next list element.}
      enddo
      end

{**********************************************************************}
module pass_pointers;
{ This module creates a linked list and loads user-supplied data into }
{ it.  It keeps a pointer to the first element in the list and sends  }
{ that value back to the FORTRAN program where the list's data gets   }
{ printed.  }

type
    letters = array[1..10] of char;
    link = ^list;
    list = record
           data : letters;
           p : link;
           end;

{ This procedure sends a pointer to the first element of the list back }
{ to the calling FORTRAN program.                                      }

procedure pass_point(out first : link);
var
    value          : letters;
    newdata, base  : link;
    done           : boolean;
    answer         : char;

begin                            { Procedure pass_point}
base := NIL;
new(first);
```

```
first^.data := 'BEGINNING'; {Assign value to first element of the list.}
first^.p := base;            {The first element is also the last, so set}
                             { pointer to nil.                          }
base := first;               { Base points to the beginning of the list.}
done := false;

repeat
    write ('Enter data: ');
    readln(value);
    new(newdata);
    while base^.p <> NIL do             {Walk to the end of the list.}
        base := base^.p;
    base^.p := newdata;                 {Add new data.               }
    newdata^.data := value;
    newdata^.p := NIL;
    write ('Again? (Y or N) ');
    readln(answer);
    if (answer = 'N') or (answer = 'n') then
        done := true;
until done;

end;
```

These programs are available online and are named **ftn_to_pas_ptr** and **pass_point**. Following is a sample run of the bound program:

```
Enter data: second
Again? (Y or N) y
Enter data: third
Again? (Y or N) y
Enter data: fourth
Again? (Y or N) y
Enter data: THE END
Again? (Y or N) n

 The linked list contains:
 BEGINNING
 second
 third
 fourth
 THE END
```

7.5 Calling a Domain/C Subprogram from Domain FORTRAN

In addition to allowing you to call Pascal subprograms, Domain FORTRAN permits you to call subprograms written in Domain/C source code. To accomplish this, perform the following steps:

1. Write source code in Domain FORTRAN that calls a subprogram. Compile it with the Domain FORTRAN compiler. Domain FORTRAN creates an object file.
2. Write source code in Domain/C. Compile it with the Domain/C compiler. Domain/C creates an object file.
3. Bind the object file(s) the FORTRAN compiler created with the object file(s) the C compiler created. The binder creates one executable object file.
4. Execute the object file as you would execute any other object file.

This chapter describes steps 1 and 2. For information on steps 3 and 4, see Sections 6.6 and 6.8.

> **NOTE:** The following sections explain how to call Domain/C from Domain FORTRAN. If you want to learn how to call FORTRAN from C, see the *Domain/C Language Reference*.

7.5.1 Reconciling Differences in Argument Passing

FORTRAN always passes arguments by reference, while C usually passes them by value. In order to pass arguments correctly, you must declare your dummy arguments in C to be pointers so that they can take the addresses FORTRAN passes in. The examples in the following sections demonstrate how to do this.

7.5.2 Case Sensitivity Issues

By default, when the Domain FORTRAN compiler parses a program, it makes all identifier names lowercase, regardless of the way you type the names in your source code. In contrast, when the Domain/C compiler parses a program, it is case-sensitive. In order to make identifier names match up at bind time, you should always use lowercase letters in your C subprograms. That way, they will match the lowercase identifier names in the FORTRAN program.

> **NOTE:** If you compile your FORTRAN programs with the **ftn –u** option, the compiler is case-sensitive for identifier names. Refer to Section 6.5.33 for details about the **–u** option.

7.6 Data Type Correspondence for FORTRAN and Domain/C

Before you try to pass data between Domain FORTRAN and Domain/C, you must understand how FORTRAN data types correspond to C data types. Table 7-2 lists these correspondences.

Table 7-2. Domain FORTRAN and Domain/C Data Types

Domain FORTRAN	Domain/C
byte	**char**
integer*2	**short**
☆ **integer, integer*4**	☆☆ **int, long, enum**
real, real*4	**float**
double precision, real*8	**double**
character*1	**char**
logical, logical*1, logical*2, logical*4	none
complex, complex*8, complex*16, double complex	user-defined **struct**
pointer statement	**pointer (*)**
none	**struct**
none	**union**
none	**unsigned char**
none	**unsigned short**
none	**unsigned long**

☆If you compile your FORTRAN source code with the **-i2** switch, the equivalent in C for the **integer** variables is the same as the equivalent for **integer*2**, namely **short**.

☆☆By default, the Domain/C compiler allocates four bytes of storage for all enumeration variables. However, the compiler also allows you to size **enums** differently.

As the table shows, the **integer**, **real**, and **character** data types in both languages correspond very well. For example, FORTRAN's **integer*2** data type is identical to C's **short** data type, and a **double precision** variable in FORTRAN is the same as a **double** in C. Note also that the Domain FORTRAN **byte** data type exactly corresponds to C's **char**: both are 8-bit signed integers. Although FORTRAN does not have a pointer type, you can use the Domain FORTRAN **pointer** statement, an extension, to simulate the C **pointer** data type.

There are some important differences. Domain/C has no equivalent types for FORTRAN's **logical, logical*1, logical*2, logical*4** or **complex, complex*8, complex*16, double complex** types, although you can simulate these types. Section 7.7.2 demonstrates how to simulate complex types.

There also are a few C types that have no FORTRAN equivalents. *Programming with Domain/OS Calls* describes how to simulate C's structure, union, and enumerated data types in FORTRAN. However, there is no easy way to simulate C's unsigned types in FORTRAN. Therefore, if you pass an unsigned value to a FORTRAN program unit, it is interpreted as a signed value. This only makes a difference when the high-order bit is set.

7.7 Passing Data between FORTRAN and C

The following sections describe how to pass variables with a variety of data types between Domain FORTRAN and Domain/C.

7.7.1 Passing Integers and Real Numbers

Since the FORTRAN and C **integer** and **real** data types match up so well, it is fairly easy to pass data of these types between the two languages. Make sure that all arguments agree in type and size, either by declaration or by casting.

The example below shows a FORTRAN program that solicits the values for two sides of a right triangle. It then sends those values into the C function, which computes and returns the length of the hypotenuse. The arguments for the triangle's sides are four bytes each, while the result is eight bytes.

```
*  This program calls a C function that calculates the hypotenuse
*  of a right triangle when a user supplies the lengths of two sides.
*  NOTE: You must also obtain the C program named "hypot_c."
*        After compiling ftn_to_c_hypot and hypot_c, you must bind
*        them together.

      program ftn_to_c_hypot
*  The external statement lets FORTRAN know that this subprogram is
*  not in this program unit, but the statement itself is optional.
      external hypot_c
      real*4 side1, side2
      real*8 result, hypot_c
      print 10
10    format (' Enter lengths for the two sides of the triangle: ', $)
      read *, side1, side2
      result = hypot_c(side1,side2)
      print *, 'The triangle''s hypotenuse is: ', result
      end
```

```
/* This is a C function for finding the hypotenuse of a
 * right triangle.  You must declare the arguments as pointers.
 * NOTE: You must also get the FORTRAN program "fortran_c1",
 * compile it, and bind it with the object code of this
 * program.
 */

#include <math.h>

double hypot_c(float *a, float *b)
{
     double result;
     result = sqrt((*a * *a) + (*b * *b));
     return(result);
}
```

These programs are available online and are named **ftn_to_c_hypot** and **hypot_c**. Following is a sample run of the bound program:

```
 Enter lengths for the two sides of the triangle: 3 4
 The triangle's hypotenuse is:  5.000000000000000
```

7.7.2 Passing Complex Numbers

Domain/C does not support a **complex**, **complex*8**, **complex*16**, or **double complex** data type, but these data types are easy to simulate. Simply create a structure containing two floating–point members of the correct size. In the following example, the FORTRAN program sends a **complex** argument to the C function **pass_complex**, which returns the square of the argument:

```
*   This program returns the square of a complex number.
*   It demonstrates passing a complex argument to a C function.

*   NOTE: You must also obtain the C program named "pass_complex."
*        After compiling ftn_to_c_complex and pass_complex, you
*        must bind them together.

      program ftn_to_c_complex

      complex  comx, pass_complex, result
      data comx /(2.5,3.5)/
      print *, 'C will square this complex number: ', comx
      result = pass_complex(comx)
      print *, 'The squared result is: ', result

      end
```

```
/* This function returns the square of the complex variable
 * passed to it by the FORTRAN program ftn_to_c_complex.
 *
 * NOTE: You must also get the FORTRAN program "ftn_to_c_complex",
 * compile it, and bind it with this program's object code.
 */

typedef struct                /* define the structure for the */
    {                         /* complex type                 */
    float real;
    float imag;
    } complex;

complex pass_complex(complex *comx)
{
     complex result;
     float a, b;

     a = comx->real;    /* dereference the pointer variable  */
     b = comx->imag;    /* and assign the values of the      */
                        /* struct's parts to local variables */

     /* square the complex variable the FORTRAN program sent in */
     result.imag = 2 * (a * b);
     a = a * a;
     b = b * b;
     result.real = a - b;

     return(result);
}
```

These programs are available online and are named **ftn_to_c_complex** and **pass_complex**. If you run the bound program, you get this output:

```
 C will square this complex number:  (2.500000,3.500000)
 The squared result is:  (-6.000000,17.50000)
```

7.7.3 Passing Character Arguments

Passing arguments when the two languages' data types match exactly is relatively straightforward, but passing them when they don't match requires some additional programming. Such is the case with passing character arguments. The following subsections discuss how to handle C's null-terminated string in a FORTRAN program and how to handle FORTRAN's "hidden" string length argument in a C program.

7.7.3.1 C's Null-Terminated String

One difference between the two languages is that C null-terminates strings with the null character (**0**) so that they behave like variable-length strings. When passing a string from FORTRAN to C, you should therefore take care to declare a character array that is big enough to accomodate both the string and the null character that C may append to the string. If the string occupies the entire array, appending the null character truncates the string.

If your FORTRAN program is going to use a string that has been passed back to it from C, you may want to strip off the null character, either in the C subprogram or in FORTRAN; the sample program in this section shows how to strip it off in C.

7.7.3.2 FORTRAN's "Hidden" String Length Argument

Another difference in the way C and FORTRAN pass character strings is that FORTRAN implicitly passes a "hidden" string length argument along with each character string in the argument list. By default, Domain FORTRAN passes the length argument by reference. You should therefore add an extra argument to the C subprogram's list for each string in its argument list.

Suppose your FORTRAN program includes the following:

```
CHARACTER*10 FIRST_NAME
 .
 .
 .
CALL CHANGE_NAME(FIRST_NAME)
```

Your C subprogram heading should include an "extra" dummy argument for the length of **first_name**; for example:

```
change_name(char *name, short *len)
```

Note that the length argument must appear at the end of the argument list. If you send multiple strings to Domain/C, the length arguments must always appear collected together at the end of the argument list. It is not correct to list them in the order **string1, len1, string2, len2**. (The example program below shows the correct order when passing multiple strings.)

The length argument must be of type **short** because FORTRAN's implicit length argument is an **integer*2**, and it must be a pointer because, by default, Domain FORTRAN passes the length argument by reference, not by value.

f77 passes the arguments for character strings as 4-byte integers, and it passes them by value, not by reference. If you want to use the **ftn** command to compile a program that was designed to be compiled with the **f77** command, you should use the **-uc** option, which (among other things) causes the compiler to pass the hidden length argument as a 4-byte integer, and to pass it by value rather than by reference. In this case, you would also have to rewrite the C subprogram heading to accept the string length argument as a 4-byte integer, as in the following example:

```
change_name(char *name, long len)
```

The command line for compiling the FORTRAN program with the **ftn** command would be:

```
$ ftn my_program.f -uc
```

For more information about the **-uc** option, refer to Section 6.5.34.

The following FORTRAN program sends two strings to a C subprogram which prompts a user for new values and then sends the new string values back. Notice that the C subprogram lists the two string dummy arguments first, followed by the two length arguments. Both programs assume that the FORTRAN subprogram will pass the hidden length arguments as 2-byte integers and that it will pass them by reference.

```
*  This program demonstrates passing character variables to and
*  getting them back from a C subprogram.

*  NOTE: You must also obtain the C program named "pass_char."
*        After compiling ftn_to_c_chars and pass_char, you must
*        bind them together.

      PROGRAM FTN_TO_C_CHARS

      EXTERNAL PASS_CHAR
      CHARACTER*10 FIRST_NAME
      CHARACTER*15 LAST_NAME
      DATA FIRST_NAME, LAST_NAME /'Sherlock', 'Holmes'/
      PRINT 10, FIRST_NAME, LAST_NAME
10    FORMAT (/, 'Before calling C, this is the name: ', A,1X,A)

      CALL PASS_CHAR(FIRST_NAME, LAST_NAME)
      PRINT 20, FIRST_NAME, LAST_NAME
20    FORMAT (/, 'After calling C, this is the name: ', A,1X,A)
      END
      PRINT 20, FIRST_NAME, LAST_NAME
20    FORMAT (/, 'After calling C, this is the name: ', A,1X,A)

      END

/*****************************************************************
 * This function is called by the FORTRAN program 'ftn_to_c_chars'.
 * You must compile and bind that program with the object code
 * of this program.
 */

#include <stdio.h>

void pass_char(char *first_name, char *last_name, short *len1,
               short *len2)
{
     short i, j;
     char hold_first[10], hold_last[15];
```

```
    printf ("\nEnter the first name and last name of a detective: ");
    scanf ("%s%s", hold_first, hold_last);

    /* Strip off the null character C automatically appends to any
     * string and blank out any unused places in the name strings.
     */

    for (i = 0; hold_first[i] != '\0'; i++)
        first_name[i] = hold_first[i];
    for (j = i; j < 10; j++)
        first_name[j] = ' ';

    for (i = 0; hold_last[i] != '\0'; i++)
        last_name[i] = hold_last[i];
    for (j = i; j < 15; j++)
        last_name[j] = ' ';
}
```

These programs are available online and are named **ftn_to_c_chars** and **pass_char**. Following is a sample run of the bound program:

```
Before calling C, this is the name: Sherlock   Holmes

Enter the first name and last name of a detective: Philip Marlowe

After calling C, this is the name: Philip     Marlowe
```

7.7.4 Passing Arrays

Single-dimensional arrays (except for logical arrays) of the two languages correspond fairly well. The only significant difference is that, in FORTRAN, array subscripts by default begin at one; in C, they begin at zero. However, this difference does not result in any size discrepancies. For example, the following declarations create arrays with the same number of elements:

In Domain FORTRAN | *In Domain/C*

```
CHARACTER*10  X          char x[10]
INTEGER*2     X(50)      short x[50]
REAL*8        X(20)      double x[20]
```

And the following code fragments access the identical elements in an array:

In Domain FORTRAN | *In Domain/C*

```
DO I = 1,10                   for (i = 0; i < 10; i++)
    MY_ARRAY(I) = I               my_array[i] = i;
ENDDO
```

As described earlier, FORTRAN passes arguments by reference, so when it sends an array argument to a subprogram, it is actually sending the address of the first element in the array. C gets that address when you declare (in the function's argument list) an array variable of indeterminate size. If your FORTRAN program includes the following:

```
INTEGER*4 X(10)
          .
          .
          .
CALL PASS_ARRAY(X)
```

your C subprogram heading should look like this:

```
pass_array(long x[]) /* Notice that the array is of indeterminate size */
```

The following example shows a Domain FORTRAN program that loads five user-entered scores into a single-dimensional array and then sends that array to a C function to compute the average. Notice that the argument **size** determines the size of the array; the C declaration of the array includes empty brackets only, indicating that the array is of indeterminate size.

```
*   This program, "ftn_to_c_array_single", demonstrates how to
*   pass a single-dimensional array to the C routine, single_dim.
*   NOTE: You must also obtain the C program named "single_dim",
*   compile both this program and the C program, and bind both
*   object files together.

      PROGRAM FTN_TO_C_ARRAY_SINGLE

      EXTERNAL SINGLE_DIM
      INTEGER*2    I,J,SIZE
      PARAMETER    (SIZE=5)
      INTEGER*2    GRADES(SIZE)
      REAL*4       RESULT
      DATA         RESULT /0.0/

      PRINT 10, SIZE
10    FORMAT ('Enter ', I2, ' integer test scores separated by commas.')
      READ *, (GRADES(I), I=1,SIZE)

      CALL SINGLE_DIM(RESULT, SIZE, GRADES)
      PRINT 30, RESULT
30    FORMAT(/,'C computed the average of the test scores, and it is: ',
     +       F5.2)

      END
```

```
/* The name of this program is "single_dim".  You must also get
 * the FORTRAN program "fortran_to_c_array_single", compile both
 * programs, and bind them together.
 */

void single_dim(float *result, short *size, short grades[])
{
     short i,total;
     total = 0;

     for (i = 0; i < *size; i++)  /* Add up array values */
         total += grades[i];

     *result = total / 5.0;       /* Compute average */
}
```

These programs are available online and are named **ftn_to_c_array_single** and **single_dim**. Following is a sample run of the bound program:

```
Enter  5 integer test scores separated by commas.
85,92,100,79,96

C computed the average of the test scores, and it is: 90.40
```

Multidimensional arrays in the two languages do not correspond as directly as single-dimensional arrays. The correspondence is complicated by the way each lays out multidimensional arrays in memory: FORTRAN lays out arrays in *column-major order*—that is, the leftmost dimension varies fastest; C lays out arrays in *row-major order*—that is, the rightmost dimension varies fastest. For example, FORTRAN represents the six elements of a two-dimensional array defined this way

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

in the following order:

```
1,1
2,1
1,2
2,2
1,3
2,3
```

But C represents the six elements of the same array defined this way

```
long my_array[2][3];
```

in the following order:

```
0,0
0,1
0,2
1,0
1,1
1,2
```

Chapter 7. Cross-Language Communication

To compensate for this difference, you should declare the dimensions of the C array in reverse order. Thus, to ensure that C correctly represents an array declared in a FORTRAN program this way

```
INTEGER*2 MY_ARRAY(3,6)
```

the C program should declare it this way:

```
short my_array[6][3];
```

Following are two more examples:

In Domain FORTRAN	*In Domain/C*
REAL*4 X(10,5)	float x[5][10];
REAL*4 X(2,3,4)	float x[4][3][2];

The following FORTRAN program and C function show the order in which the two languages represent identical data:

```
*   This program, "ftn_to_c_array_multi", demonstrates how to
*   pass a multi-dimensional array to the C routine, multi_dim.

*   NOTE: You must also obtain the C program named "multi_dim",
*   compile both this program and the C program, and bind both
*   object files together.

      PROGRAM FTN_TO_C_ARRAY_MULTI

      INTEGER*2   I,J
      INTEGER*4   NUMS(2,4)
      DATA ((NUMS(I,J), I=1,2), J=1,4)/1,2,3,4,5,6,7,8/

      PRINT 10
10    FORMAT (/,'This is how FORTRAN stores the array:',/)
      DO J = 1,4
          DO I = 1,2
              PRINT 20, I, J, NUMS(I,J)
20            FORMAT ('nums(', I1, ',', I1, ') = ', I2)
          ENDDO
      ENDDO
      CALL MULTI_DIM(NUMS)
      END
```

```
/* The name of this program is "multi_dim".  You must also get the
 * FORTRAN program "fortran_to_c_array_multi", compile both the
 * FORTRAN and C programs, and bind them together.
 */
void multi_dim(long b[4][2]) /* Size information included, with
                              * dimensions reversed, for this array.
                              */
{
    short i,j;
    printf ("\nThis is how C stores the array:\n\n");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 2; j++)
            printf("nums(%d,%d) = %d\n", i, j, b[i][j]);
}
```

These programs are available online and are named **ftn_to_c_array_multi** and **multi_dim**. If you run the bound program, you get the following output:

```
This is how FORTRAN stores the array:
nums(1,1) =  1
nums(2,1) =  2
nums(1,2) =  3
nums(2,2) =  4
nums(1,3) =  5
nums(2,3) =  6
nums(1,4) =  7
nums(2,4) =  8

This is how C stores the array:

nums(0,0) = 1
nums(0,1) = 2
nums(1,0) = 3
nums(1,1) = 4
nums(2,0) = 5
nums(2,1) = 6
nums(3,0) = 7
nums(3,1) = 8
```

7.7.5 Passing Subprograms

Passing subprograms between FORTRAN and C is complicated by the fact that FORTRAN adds another level of indirection when it passes a subprogram—that is, it passes the address of the address of the subprogram. C, on the other hand, passes just the address of a subprogram. To compensate for this difference, you must prepare the C program that is to receive the subprogram by declaring a pointer to a pointer in the C function's argument list. In the body of the function, you must declare a pointer to a function and assign this pointer to the dereferenced pointer-to-a-pointer. (For an explanation of why C won't allow you to declare a pointer-to-a-pointer-to-a-function as an argument, refer to the *Domain/C Language Reference*.)

For example, assume that the FORTRAN program is to pass the subprogram **ftn_sub** to the C function **c_func**. (For the sake of simplicity, we'll assume that **ftn_sub** takes no arguments.) Keep in mind also that whenever you pass a user-defined subprogram as an argument in FORTRAN, you must declare it in an **external** statement. Here are the relevant statements to make the call in the FORTRAN program:

```
      EXTERNAL FTN_SUB
      .
      .
      .
      CALL C_FUNC(FTN_SUB)
```

The C function **c_func** must be prepared both to accept a pointer-to-a-pointer and to dereference it and assign its value to a pointer-to-a-function. Here are the C statements to receive the function from FORTRAN:

```
void c_func(void **ppf)           /* declare a pointer to a pointer */
{
    void (*compare)(void);  /* declare a pointer to a function */

    compare = *ppf;               /* assign the de-referenced pointer
                                   * to the pointer to a function
                                   */

    .
    .
    .
}
```

Note that in FORTRAN, the call is straightforward. It is only in C that you have to resort to a workaround.

The following is a sample FORTRAN program that passes an integer array to a C function that sorts the array. The FORTRAN program also passes one of two subprograms, **ascend** or **descend**, to the C function. Depending on which subprogram is passed, the C function will sort the array in ascending or descending order. Note that since the array is passed by reference, C does not have to return it. Note also that FORTRAN passes the size of the array as an argument, **size**, along with the array itself and the function.

```
* This program, "pass_func_to_c", illustrates how to pass a subprogram
* to a C function.  It passes a function, an unsorted array, and the
* size of the array.  The function can be either ASCEND, which compares
* two integers for an ascending sort, or DESCEND, which compares two
* integers for a descending sort.  The C function sort_array calls one
* of the two FORTRAN functions when it sorts the array.

* NOTE: You must also obtain the C program named "sort_array", compile
* it and this program, and bind both object files together.

      PROGRAM PASS_FUNC_TO_C
```

```
* Declare ASCEND and DESCEND as external because they will be
* passed as arguments.
      EXTERNAL ASCEND, DESCEND
      INTEGER SIZE
      PARAMETER (SIZE = 10)
      INTEGER I_AR(SIZE), DIRECTION
      DATA I_AR /27, 19, 34, 65, 7, 9, 12, 2, 75, 1/

      PRINT 10, 'Enter 1 to sort in ascending order, '
      PRINT 10, '2 to sort in descending order:  '
   10 FORMAT (A, $)
      READ *, direction
* Call the C function and pass either ASCEND or DESCEND, depending
* upon the user's choice.
      IF (DIRECTION .EQ. 1) THEN
           CALL SORT_ARRAY(ASCEND, SIZE, I_AR)
      ELSE
           CALL SORT_ARRAY(DESCEND, SIZE, I_AR)
      ENDIF
* Display the numbers in sorted order
      DO 20 I = 1, SIZE
           PRINT *, I_AR(I)
   20 CONTINUE
      STOP
      END
*
* Compares two integers for ascending sort and returns
* 1 if they're out of order, 0 otherwise.
*
      INTEGER FUNCTION ASCEND(A, B)

      INTEGER A, B

      IF (A .GT. B) THEN
           ASCEND = 1
      ELSE
           ASCEND = 0
      ENDIF
      RETURN
      END
*
* Compares two integers for descending sort and returns
* 1 if they're out of order, 0 otherwise.
*
      INTEGER FUNCTION DESCEND(A, B)

      INTEGER A, B

      IF (B .GT. A) THEN
           DESCEND = 1
      ELSE
           DESCEND = 0
```

```
      ENDIF
      RETURN
      END

/*******************************************************************
 * The name of this program is "sort_array".  You must also get the
 * FORTRAN program "pass_func_to_c", compile both this program and
 * the FORTRAN program, and bind them together.
 */

#include <stdio.h>

/* FORTRAN passes a subprogram as the address of the address of the
 * subprogram.  But C expects only the address of the function.  So
 * as a workaround, you have to declare a pointer to a pointer (ppf)
 * in C's argument list, declare a pointer to a function (compare)
 * inside sort_array, then assign the de-referenced ppf to compare.
 */
void sort_array(void **ppf, int *size, int list[])
{
    int (*compare)(int *, int *);  /* declare a pointer to a function */
    int i, temp;
    enum { false, true } out_of_order;

    compare = *ppf;
    do
    {
        out_of_order = false;
        for (i = 0; i < *size-1; i++)
            if ((*compare)(&list[i], &list[i+1]))
            {
                out_of_order = true;
                temp = list[i];
                list[i] = list[i+1];
                list[i+1] = temp;
            }
    }while (out_of_order);
}
```

These programs are available online and are named **pass_func_to_c** and **sort_array**. If you run the bound program, you get the following output:

```
Enter 1 to sort in ascending order, 2 to sort in descending order:  2
 75
 65
 34
 27
 19
 12
 9
 7
 2
 1
```

7.7.6 Data Sharing Between FORTRAN and C

In FORTRAN, you declare variables as external by placing them in a **common** block. A **common** block declaration creates an overlay data section. To communicate with a C program, the C program must create an overlay section with the same name.

Domain/C supports two methods of compiling C programs—using **/com/cc** or **bin/cc**. If you compile with **/com/cc**, you can create an overlay section by defining a global variable. If you compile with **/bin/cc**, you may use either the **-nbss** option or the **__attribute((section))** modifier. The *Domain/C Language Reference* explains how and why you use the **-nbss** option and the **__attribute((section))** modifier.

For example, if your FORTRAN program includes the following code

```
INTEGER*4 X
COMMON /XVAR/ X
```

and you are using **/com/cc**, the declaration in the C module is

```
int xvar;
```

and the **/bin/cc** declaration is

```
int xvar __attribute(( section(xvar) ));
```

Note that the C declaration corresponds to the name of the **common** block, not to the name of the variable in the **common** block.

If you are compiling the FORTRAN program with **f77**, you must use the **-W0,-nuc** option to prevent the compiler from appending the underscore (_) character to global names. If you do not use this option and you do not append the underscore character to globals in the C module, you will get unresolved references when you bind the FORTRAN and C programs. C does not append the underscore character to global names. (For more information about the **-W0** and **-nuc** options, refer to Sections 6.4 and 6.5.34, respectively.)

If the FORTRAN **common** block contains more than one external variable, the C source file should define an external structure with the same name as the **common** block. The fields of the structure should correspond to the variables in the **common** block. For example, if your FORTRAN program includes the following code:

```
INTEGER*4 IFIELD
REAL RFIELD
COMMON /CNAME/IFIELD,RFIELD
```

the **/com/cc** version of the declaration is

```
struct {
    int ifield;
    float rfield;
       } cname;
```

Here is the **/bin/cc** version of the declaration:

```
struct {
    int ifield;
    float rfield;
       } cname __attribute(( section(cname) ));
```

Note that the variable is declared as **cname** and not **CNAME** in the C programs. This is because all FORTRAN global names are exported to the linker in lowercase.

The following FORTRAN program and C function show how the two communicate with each other through a **common** block.

```
* This program demonstrates how to use a common block to
* share data between FORTRAN and C.  Note that the common
* block contains more than one external variable, so that
* the C function must set up a struct with the two variables
* as members.  After compiling this program, you must obtain
* "get_nlog.c", compile it, and and bind both object files.

      PROGRAM FTN_TO_C_NLOG
      REAL*8 NUM, NLOG_OF_NUM
      COMMON /GLOBAL_VARS/ NUM, NLOG_OF_NUM
      PRINT *, 'Number   Natural Log of Number'
      PRINT *, '-------+----------------------'
      DO 10 NUM = 2.0, 10.0
           CALL GET_NLOG()
           PRINT 20, NUM, '|', NLOG_OF_NUM
   10 CONTINUE
   20 FORMAT (3X, F3.0, 2X, A, 8X, F5.2)
      STOP
      END

/* This is a C function for finding the natural log of
 * a number.  The struct global_vars is required because
 * the FORTRAN program that calls this function contains
 * more than one external variable.
 */
#include <math.h>
#include <stdio.h>

struct
{
    double num;
```

```
    double nlog_of_num;
} global_vars;

void get_nlog(void)
{
    global_vars.nlog_of_num = log(global_vars.num);
}
```

If you compile the FORTRAN program with **f77**, the command line is:

$ f77 -c -W0,-nuc ftn_to_c_nlog.f

If you compile the C function with **/bin/cc**, you can use either the following command line

$ /bin/cc -c -W0,-nbss get_nlog.c

or add the **__attribute((section))** modifier to the declaration of the struct **global_vars** as follows:

```
struct
{
    double num;
    double nlog_of_num;
} global_vars __attribute(( section(global_vars) ));
```

These programs are available online and are named **ftn_to_c_nlog** and **get_nlog**. If you run the bound program, you get the following output:

```
Number   Natural Log of Number
-------+---------------------
   2.  |        0.69
   3.  |        1.10
   4.  |        1.39
   5.  |        1.61
   6.  |        1.79
   7.  |        1.95
   8.  |        2.08
   9.  |        2.20
  10.  |        2.30
```

7.8 System Service Routines

Using Domain system calls, you can handle errors, perform I/O, and communicate with other processes. These routines are described in the *Domain/OS Call Reference* and *Programming With Domain/OS Calls*. Domain graphics calls are described in the *Domain Graphics Primitives Resource Call Reference* and *Programming with Domain GPR*.

7.8.1 Insert Files (%include)

There are a number of insert files distributed with the operating system and language software. An insert file defines constants and type definitions used by the system service routines, as well as declarations of the system service routines themselves.

Each system component has an associated insert file. For example, there are insert files for serial I/O, for touchpad manipulation, and for error reporting. All insert files are distributed in the directory /sys/**ins**. The insert files for Domain FORTRAN programs appear in the form

/sys/**ins**/*component_name*.**ins.ftn**

where *component_name* is one of the system insert files listed in the *Domain/OS Call Reference*.

To place system definitions in your program, use either the **%include** compiler directive or the **include** statement. (See the "Compiler Directives" listing in Chapter 4 for more details about **%include** and the listing for **include** in the same chapter for more details about **include**). The example below shows the files needed for a Domain FORTRAN program that uses the system I/O routines and system error-handling routines:

```
%include '/sys/ins/base.ins.ftn' { This file should ALWAYS be first }
%include '/sys/ins/streams.ins.ftn'
%include '/sys/ins/error.ins.ftn'
```

Always include the insert file /sys/**ins**/**base.ins.ftn** first, since some of the other system insert files rely on the definitions in this file.

7.8.2 Returned Status Code

Most system routines return a status code as a value of the system's **status_$t** type, which corresponds to an **integer*4** type in FORTRAN. The status code indicates whether the routine completed successfully. If it was successful, the operating system returns a value of **STATUS_$OK** (defined as zero in /sys/**ins**/**base.ins.ftn**) to the error status argument (**status_$t**). If the call was not successful, the operating system returns a number symbolizing the error. You should check the value of the status code after each system call to find out if errors occurred.

Every nonzero status code is associated with descriptive error text. To analyze the status code and retrieve the text, use the error-handling routines described in *Programming with Domain/OS Calls*.

7.8.3 Binding and Execution

The system service routines are included in preinstalled, shared libraries. Since references to identifiers in these libraries are resolved at execution time, you do not need to specify any additional files when compiling a program that calls system service routines. For information about **ld**, refer to subsection 6.6.1 and the *BSD Command Reference* or *SysV Command Reference*. For more information about **bind**, refer to subsection 6.6.2 and the *Domain/OS Programming Environment Reference*.

Chapter 8

Input and Output

Domain FORTRAN supports the following three methods of performing I/O:

- Input/Output Stream (IOS) calls
- Variable format (VFMT) calls
- Domain FORTRAN I/O statements

In general, you can perform all your I/O with the Domain FORTRAN I/O statements. However, the other two methods can be useful in certain circumstances.

This chapter provides a brief overview of all three methods, along with some background information that may help you in whatever method you choose.

8.1 Some Background on I/O in the Domain/OS Environment

This section provides a brief summary of information that may be helpful in understanding how I/O works on the Domain system. See *Programming with Domain/OS Calls* for full details about I/O. Before we describe the mechanics of I/O, we give a brief description of IOS calls and VFMT calls.

8.1.1 Input/Output Stream (IOS) Calls

IOS calls are system calls that perform I/O. You can easily make IOS calls from your Domain FORTRAN program. IOS calls can

- Create a file
- Open or close a file
- Write to or read from a file
- Change a file's attributes (a file's attributes include name, length, accessibility)
- Access magnetic tape files or serial lines

IOS calls are more primitive than the Domain FORTRAN I/O statements. Consequently, they give you more control over I/O, but they are harder to use. Therefore, for simple I/O needs you are probably better off using the FORTRAN I/O statements. If you want to do something out of the ordinary, you probably need to use IOS calls.

Programming with Domain/OS Calls details IOS.

8.1.2 VFMT Calls

VFMT (variable format) calls are special system calls that format input and output. Because FORTRAN's format statement allows you wide flexibility in formatting input and output data, you probably won't need to use **VFMT**. However, it is available for situations such as the following:

- A variable contains a hexadecimal value and you want to prompt your user with its ASCII equivalent.
- You want to tabulate results in fixed columns using scientific notation.

VFMT is a set of tools for converting data representations between formats.

VFMT performs two classes of operations: encoding and decoding. **Encoding** means taking program-defined variables and producing strings of readable text that represent the values of the variables, in a format that you specify. These encoded values are then often written to output for viewing. **Decoding** means taking readable text (typically typed by the user), interpreting it in a way that you specify, and storing the apparent data values in program-defined variables. Note that the terms "encode" and "decode" as used here have nothing to do with the **encode** and **decode** statements described in Chapter 4.

The VFMT calls allow you to format the following kinds of data:

- ISO Latin-1 characters, including ASCII characters
- 2-byte or 4-byte integers interpreted as signed or unsigned integers in octal, decimal, or hexadecimal bases
- Single-precision and double-precision reals in floating-point and scientific notations

The kinds of data that you can format with VFMT calls include the following Domain FORTRAN data types: **character**, **integer*2**, **integer*4**, **real**, **real*4**, **real*8**, and **double precision**.

Programming with Domain/OS Calls details VFMT calls.

The remainder of this section is devoted to explaining certain aspects of Domain I/O that you may find useful.

8.1.3 File Variables and Stream IDs

All of the Domain FORTRAN I/O statements except **print** take a unit or internal file ID as an argument. These identifiers are a synonym for a temporary or permanent pathname to the file. If you are using IOS calls rather than the Domain FORTRAN I/O statements, you refer to a pathname by its IOS ID rather than by its file variable. An IOS ID is a number assigned by the operating system when you open a file or device. Since FORTRAN I/O statements in your source code ultimately translate to IOS calls at run time, a file variable in your source code becomes an IOS ID at run time. The system can support up to 128 I/O streams per process.

8.1.4 Default Input/Output Streams

Every process starts out with the I/O streams shown in Table 8-1. Domain FORTRAN deals in unit and file IDs, not IOS IDs, so the table also shows the names of the file variables corresponding to these streams. You need not explicitly open **-stdin** or **-stdout**; the system opens these streams automatically. See the listing for "I/O attributes" in Chapter 4 for more details about units.

Table 8-1. The Default Streams

Stream Name	File Variable Name	Description
ios_$stdin	**-stdin**	If you specify unit 5 or list-directed formatting for a Domain FORTRAN input statement, the compiler reads from **stdin**. By default, **stdin** is the process input pad, but you can redirect this stream with the < character.☆
ios_$stdout	**-stdout**	If you specify unit 6 or 7 or list-directed formatting for a Domain FORTRAN output statement, the compiler assumes **stdout**. By default, the system associates this stream with the process transcript pad, but you can redirect this stream with the > character.*
ios_$errout **ios_$stderr**	**-errout**	By default, unit 0 is connected directly to **errout**. Domain FORTRAN sends errors to this stream. By default, this is the transcript pad, but you can redirect this stream with the > character sequence.☆
☆*Getting Started with Domain/OS* explains how to redirect I/O.		

8.1.5 Stream Markers

When you open a file, the operating system assigns a stream marker. A **stream marker** is a pointer that points to the current position inside the open file. As you read from or write to the file, the operating system moves the stream marker forward in the file. The stream marker points to the byte that the system can next access. When you open a file with the **open** statement, the stream marker initially points to the beginning of the file. Using IOS calls, you can open the file with the stream marker initially pointing to the end of file so that you can append to the file.

If you are using IOS calls, you directly control the stream marker. If you are using Domain FORTRAN I/O statements, you control the stream marker indirectly through the statements you use.

8.1.6 Domain/OS File Organization

Domain/OS supports the following four types of file organization:

- UASC context delimited ASCII record files ("UASC file" for short)
- Fixed-length record files ("**rec** file" for short)
- Variable-length record files
- No defined record structure files

Using IOS calls, you can create any of the four types of files. However, using Domain FORTRAN, you can create only the first type, UASC files. Pre-SR10 versions of Domain FORTRAN created **rec** files when you opened a file with the **form** = 'unformatted' attribute. At SR10 you can still use the current version of Domain FORTRAN to manipulate **rec** files, but you can no longer *create* **rec** files with Domain FORTRAN. When you open an existing file, Domain FORTRAN checks whether it is a **rec** or UASC file and accesses it appropriately. Whenever you create a new file, however, Domain FORTRAN creates a UASC file.

A UASC file is an ordinary text file. The system stores a text file as a 32-byte header followed by ASCII characters. The operating system makes no attempt to organize or structure the data in a text file. For example, '908' is stored in the three bytes it takes to hold the ASCII values of digit '9', digit '0', and digit '8', rather than structuring it into the value of integer 908.

By default, any file Domain FORTRAN creates has a maximum line length of 256 bytes. However, you can choose your own length with the **reclen=** I/O attribute.

Note that Domain FORTRAN programs can read any output files created by Fortran 77 compilers for use on UNIX systems.

> **NOTE:** You can use either of the two methods to manipulate your files—FORTRAN I/O statements or Domain IOS calls. However, since the files are formatted somewhat differently according to the method that you use, using both methods for the same files is quite complicated. Thus, in general, we recommend that you use one method or the other, but not both. For example, if you create a file using the FORTRAN **open** statement, then you should access that file using a FORTRAN statement such as **read**.

8.2 Domain FORTRAN Files

Domain FORTRAN supports both *external files*, which are stored on a peripheral device such as a disk, and *internal files*, which are in-memory storage areas. The following subsections describe both kinds of files.

8.2.1 External Files

Any external file created by a Domain FORTRAN program is organized as a UASC file, which is preceded by a 32-byte header (refer to subsection 8.1.6). If the file was created with the **form** = 'formatted' attribute, the data that follows the header is stored in ASCII format. If it was created with the **form** = 'unformatted' attribute, the data is stored in binary format. It is possible to open an existing file for formatted I/O that was created as an unformatted file, and vice-versa: Domain FORTRAN treats all UASC files as though they consisted of ASCII data. It is therefore your responsibility to open an existing file with the appropriate **form** = attribute and not inadvertently attempt to access binary data as though it were ASCII data.

Domain FORTRAN supports both *sequential* files and *direct-access* files. If you **open** a new file without specifying its **access** = attribute, Domain FORTRAN creates a sequential file by default. Sequential files store records in the order in which they were written, and they must be read in the same serial order. Domain FORTRAN automatically appends an end-of-file marker to any sequential file it creates. You can move this marker with the **endfile** statement (refer to Chapter 4).

Sequential files can be either formatted (the default) or unformatted. Unformatted sequential files include a 4-byte length specifier both before and after each written record, giving the length of the data in bytes. The following program

```
      OPEN(UNIT=1, FILE='UNFMTD_SEQ', FORM='UNFORMATTED',
     Z     ACCESS='SEQUENTIAL', STATUS='NEW')
      WRITE(1)  6
      CLOSE(1)
      END
```

writes a single record that looks like this:

0000 0004 0000 0006 0000 0004

The actual data is "0000 0006"; the "0000 0004" on either side of it is the length specifier, indicating that the data is four bytes long.

Direct–access files can be read or written in any order, regardless of the order in which records were written to the file. Domain FORTRAN uses the **recl** = and **rec** = attributes that you supply to access the record you wish to read or write. For example, the following program

```
      INTEGER I,J

      OPEN(UNIT=1, FILE='UNFMTD_DIR', FORM='UNFORMATTED',
     Z     ACCESS='DIRECT', STATUS='OLD', RECL=8)
      READ(1, REC=15) I, J
      PRINT *, I,J
      CLOSE(1)

      STOP
      END
```

computes an offset of 120 bytes (8 * 15) into the file **UNFMTD_DIR** in order to access the desired record. If you do not specify the **recl** = attribute, Domain FORTRAN assumes a record length of 256 bytes. If this default length happens not to coincide with the actual length of the record you are attempting to access, you risk either a run–time error or (worse) bad data. We recommend that you always specify the **recl** = attribute when working with direct–access files.

Direct–access files can be either formatted or unformatted (the default).

8.2.2 Internal Files

Internal files are "stored" in the computer's memory where they exist for the life of the program. Their main use is to enable programs to use file I/O statements to transfer data within memory in character format. Older versions of FORTRAN provided the **encode** and **decode** statements to perform this function (see Chapter 4), but these statements are nonstandard; if you are developing a new program, you should use internal files instead.

An internal file can be one of the following:

- **Character** variable
- **Character** array element
- **Character** array
- **Character** substring

An internal file's record is the **character** variable, array element, or substring. The record length is the length of the variable, element, or substring. Accessing records in an internal file is analogous to accessing them in a formatted, sequential file. Records must be defined before they can be read and are defined by the act of writing to them. You can also define the character variable, element, or substring with an assignment statement. If the number of characters written to a record is less than the record length, the remaining characters are filled with blanks.

To access internal files, you can use any of the I/O statements that you would use for sequential access except for the auxiliary I/O statements (**open**, **close**, **inquire**, and the file-positioning statements). For information on using the **read** and **write** statements with internal files, refer to the listings for these statements in Chapter 4.

The following program illustrates several uses of an internal file:

```
C This program illustrates internal files.  The internal
C file in this example is the character array INTFIL.

      PROGRAM INTERNAL_FILE_EXAMPLE

      INTEGER ARRAY_SIZE
      PARAMETER (ARRAY_SIZE = 6)
      CHARACTER*42 INTFIL
      REAL RATES(ARRAY_SIZE)
      DATA RATES/5.125, 12.375, 17.725, 22.250, 26.175, 32.625/

C Store an edit descriptor mask in INTFIL
      WRITE (INTFIL,10) ARRAY_SIZE
   10 FORMAT ('(', I2, '(', '''', '$', '''', ', F5.2, /))')

C Display RATES array formatted by the mask in INTFIL
      WRITE (6,INTFIL) RATES

C Display mask
      PRINT 20, 'The edit descriptor mask in intfil:'
   20 FORMAT (A, $)
      WRITE (6, *) INTFIL

C Redefine INTFIL
      WRITE (UNIT=INTFIL, FMT='(6(X, F6.3))') RATES
      WRITE (6, *) INTFIL

      STOP
      END
```

This program is available online and is named **internal_file_example**. Following is a sample run of the program:

```
$ 5.13
$12.38
$17.73
$22.25
$26.17
$32.63
The edit descriptor mask in intfil: ( 6('$', F5.2, /))
   5.125 12.375 17.725 22.250 26.175 32.625
```

8.3 Domain FORTRAN I/O Statements

The encyclopedia in Chapter 4 details the syntax of each of the Domain FORTRAN I/O statements. This section provides a global view of these statements.

8.3.1 Creating and Opening a New File

You can create a permanent file or a temporary file. The operating system deletes a temporary file as soon as the program that created it ends. Permanent files last beyond program execution. In fact, they last until you explicitly delete them.

8.3.2 Opening an Existing File

To open an existing file for future access, use **open** and specify either 'old' or 'unknown' as **status**. For example,

```
open (..., status= 'unknown', ...)
```

See the listing under "**Open**" in Chapter 4 for more information about opening an existing file.

> NOTE: When you use the **strid** I/O attribute with an **open** statement, FORTRAN attaches a FORTRAN unit to the existing stream that you specify with the **strid** attribute. Thus, you can attach streams opened with the **ios_$open** system call to a FORTRAN unit and perform FORTRAN I/O to the stream. See the "strid = stream_id" section of the "I/O Attributes" listing in Chapter 4 for an example.

You do not have to explicitly open the shell transcript pad; it is already open. (Refer to subsection 8.1.4 for details.)

8.3.3 Inquiring About a File

Domain FORTRAN supports the **inquire** statement, which allows you to ask about the characteristics of a particular file or unit. This can be valuable if your code accesses a file or unit that could have been opened in several different ways. Based on the information **inquire** returns, your code can take the appropriate action.

8.3.4 Reading from a File

In order to read from an open file, you must use a **read** or **decode** statement.

If you want to read something other than the next record in your file, you can use **backspace** or **rewind** to reposition the file marker.

Briefly, here's what each of these four statements does:

- **Read** transfers data from a file (which may be standard input) to internal memory. Use it to read information from the specified file into the given variables.

- [illegible] a Domain FORTRAN extension that transfers data from memory to [illegible] or arrays. This statement is analogous to a **read** to an internal [illegible] Domain FORTRAN extension.

- **Backspace** explicitly positions the file so you can reread or rewrite the record you just read or wrote. You also can use **backspace** to position the file before its end-of-file mark so that you can write more data.

- **Rewind** explicitly positions a sequential file before its first record. If the file already is at its starting point, **rewind** has no effect.

It is often useful to know when the file marker has reached the end of the file. You can use the **I/O** attribute **end=** to test for the end-of-file.

8.3.5 Writing to a File

In order to write to a file, you must use one of the three Domain FORTRAN output statements: **print, write,** or **encode**. You cannot write to a file unless you already have opened it. These are the differences among the statements:

- **Print** typically sends data to standard output (the screen), although you can redirect standard output using redirection notation. The syntax for the **print** statement is slightly simpler than the syntax for **write,** which is why **print** is often used when data needs to go to standard output.

- **Write** transfers data from internal storage to an output file. Use it to write information from the specified variables into the specified file.

- [illegible] formats data and transfers it to a specified memory area. It often is used [illegible] data to large memory areas that are declared as numeric types. [illegible] is analogous to a **write** to an internal file, and is a Domain FORTRAN extension.

8.3.6 Closing a File

When a program terminates (naturally or as a result of an error), the operating system automatically **closes** all open files. Closing means that the operating system unlocks it.

Domain FORTRAN supports a **close** statement whose purpose is to close a specified open file. Since the operating system does this automatically at the end of the program, you ordinarily don't have to call **close**. However, it is good programming practice to close all open files as soon as your program is finished using them. Open files tie up process resources and may cause your program to needlessly lock a file that another program wants to access.

Chapter 9

Diagnostic Messages

The majority of this chapter is devoted to detailing compiler errors and warnings. However, the chapter first describes the errors that system routines report.

9.1 Errors Returned from System Routines

All but one of the Domain FORTRAN I/O statements return an error status argument. The I/O statements that return an error status argument are:

- **backspace**
- **close**
- **endfile**
- **inquire**
- **open**
- **read**
- **rewind**
- **write**

The only Domain FORTRAN I/O statement that does not return an error status argument is **print**.

The argument tells you whether or not the call was successful. If it was successful, the operating system returns a value of 0 in the error status argument. If the call was not successful, the operating system returns a number symbolizing the error. Your program is responsible for handling the error. You may want to print the error and terminate execution. You may want to write your program so that it can take appropriate action when it encounters an error.

This error status argument is identical to the error status parameter that all system calls return. For complete details on using the error status parameter and for information on testing for specific errors, refer to *Programming With Domain/OS Calls*. For an overview relevant to Domain FORTRAN's I/O statements, read the following section.

9.1.1 Printing Error Messages

To print the message that an I/O statement returns, you must do the following:

1. Put the following two **%include** directives in your program just after the **program** statement (if present):

   ```
   %include `/sys/ins/base.ins.ftn´
   %include `/sys/ins/error.ins.ftn´
   ```

   Make sure you put **base.ins.ftn** before **error.ins.ftn**.

2. Declare the error status argument with the **integer*4** data type; for example,

   ```
   integer*4 err_stat
   ```

3. Use the **iostat = err_stat** I/O attribute in your I/O statement; for example,

   ```
   open(4, file=file_name, iostat=err_stat, status=´unknown´)
   ```

4. Call the **error_$print** subprogram with the error status argument as its sole argument; for example,

   ```
   call error_$print(err_stat) {Error_$print is defined}
                               {in error.ins.ftn.      }
   ```

The following program puts all the steps together:

```
      program test

%include ´/sys/ins/base.ins.ftn´
%include ´/sys/ins/error.ins.ftn´

      integer*4    open_status

      open (10, file=´my_file´, iostat=open_status, access=´sequential´)
      call error_$print(open_status)
      close (10)

      end
```

The **error_$print** procedure writes the error or warning message to **stdout**. If there is no error or warning, **error_$print** writes the following message to **stdout**:

```
status 0 (OS)
```

9.2 Compiler Errors, Warnings, and Information

When you compile a program, the compiler reports errors, warnings, and information.

An **error** indicates a problem severe enough to prevent the compiler from creating an executable object file.

A **warning** is less severe than an error; a warning does not prevent the compiler from creating an executable object file. The warning message tells you about a possible ambiguity in your program for which the compiler believes it can generate the correct code.

An **information** message is less severe than a warning; it alerts you to ways in which you could improve the quality of your code. Information messages tell you about the following types of things:

- Alignment of variable and type definitions
- Actions taken by the optimizer

In the following pages we list the common Domain FORTRAN compiler error, warning, and information messages and ways to handle them. With many messages, we include a code fragment showing a way that a particular error or warning could occur, or adding to the information in the information message. If it is a multiline fragment, the line on which the compiler would detect the error is marked with the comment

```
{Wrong!}
```

to help you find the error easily.

9.2.1 Error, Warning, and Information Message Conventions

The messages listed in the rest of this chapter follow these conventions:

- Keywords in the message text are capitalized, since that's the way they appear on your screen. In the accompanying explanatory text, they are lowercase bold, as they are elsewhere in this manual.
- Italicized words in the message text indicate values that the compiler fills in when generating the message.

For example, suppose your program contains the following:

```
program err_test
integer*4 my_num
.
.
.
real*4 my_num
```

Because the fragment includes two different declarations of variable **my_num**, it triggers Error 144, which reads:

144 ERROR *Identifier* multiple conflicting type declarations

When you compile, *identifier* gets filled in like this:

```
(00005)          real*4  my_num
**** Error #144 on Line 5:
[  real*4  @my_num] multiple conflicting type declarations
```

When filling in values for placeholders such as *identifier,* the compiler also inserts an at-sign (@) to indicate where it found the error. In this example, the @ appears next to the variable name.

9.2.2 Error and Warning Messages

Following are Domain FORTRAN's compiler error and warning messages.

1 ERROR Already equivalenced at different displacement to *identifier*

It is an error to **equivalence** variables that already appear in one **common** block. For example, the following is an error:

```
common /area/ i,j,k
equivalence (i,j)                           {Wrong!}
```

See the listing for **equivalence** in Chapter 4 for more information.

3 ERROR *Identifier1* based variable cannot be equivalenced to common variable *identifier2*

It is an error to **equivalence** variables when one appears in a **common** block and the other appears in a **pointer** statement. For example, the following is an error:

```
pointer /my_ptr/ int_num, x
common /share/ j,k,l
     . . .
equivalence (k, int_num)                    {Wrong!}
```

4 ERROR `A DATA statement for this variable must precede its first reference (or use the -SAVE option, or place it in a SAVE statement).`

Your **data** statement is in the wrong place. A **data** statement initializing a variable must precede any statements referencing that variable. You can initialize the variable using a **data** statement, or you can tell the compiler to retain its value from another invocation of the program by using the **-save** compiler option or the **save** statement.

5 ERROR *Token* `equivalencing variables in different commons`

Variables that are in separate **common** blocks may not be **equivalenced**. See the listing for **equivalence** in Chapter 4 for more information.

7 ERROR *Identifier* `illegal common block name specification`

Common block names and the variable names within the **common** block must meet the same requirements as any other Domain FORTRAN identifiers; that is, they must start with a letter and can contain only ASCII letters, digits, underscores (_), or dollar signs ($). *Identifier* doesn't meet these requirements.

8 ERROR *Identifier* `illegal name in common variable list`

You probably have dimensioning information in the **common** statement for an array that you have already dimensioned elsewhere. This error often occurs along with Error 28.

9 ERROR *Identifier* `integer*4 variable required`

The compiler is expecting an **integer*4** variable, but *identifier* is some other type. This error occurs when a **pointer** variable, or the variable to which the **assign** statement gives a value, is not an **integer*4**.

11 ERROR `Alignment error - noncharacter item at odd address:` *identifier*

Noncharacter variables in a **common** block must begin on word boundaries, but you've set them up in such a way that one or more begins on an odd byte. This error can occur if you have a **character** entity that you've defined as having an odd number of bytes.

For example:

```
integer*4     num_classes, num_students
character*15 student_name       {notice length is odd}
logical       pass_fail
common /share/ student_name, num_classes, pass_fail
          .
          .
          .
print *, student_name, num_classes, pass_fail
```

In this case, **num_classes** is the *identifier* the compiler picks out as being at an odd address. To eliminate this error, rearrange your **common** block.

12 ERROR *Identifier* dummy argument cannot appear in common list

A dummy argument cannot appear in a **common** block, but *identifier* does just that. For instance, the following causes this error:

```
subroutine dumb(x,y,z)
common /together/ x,y                          {Wrong!}
```

13 ERROR *Identifier* invalid variable in common list; already in common

Identifier already appears in the current **common** block, or in another one. An identifier may only appear one time in one **common** block. For example, the following is incorrect because the variable **again** appears in both **common** statements:

```
common /first/ i,j,again
common /second/ again,k,l                      {Wrong!}
```

14 ERROR Invalid variable; initialized in DATA statement

If a variable name already has appeared in a **data** statement, it is an error to include that variable in a **common** block.

15 ERROR *Identifier* `invalid variable in common list; illegal name`

It is an error to include something other than a variable in a **common** block. For example, if you already have used the **parameter** statement to associate a name with a constant, you cannot then include that name in a **common** statement. This means the following is illegal:

```
integer*4 i, j, k, size
parameter (size = 100)
      . . .
common /together/ i, j, k, size                    {Wrong!}
```

This error also occurs if you use a subprogram name in a **common** block.

18 ERROR `Input line over 80 characters long`

Unless you compile with the **-ff** option or use the **f77** command, your FORTRAN source code—with the exception of comment lines—cannot extend past the 80th column. See Chapter 2 for a description of FORTRAN's column conventions, and see Chapter 6 for details about the **-ff** option and the **f77** command.

19 ERROR `Improper character*(*) declaration of identifier` *identifier*

When the **character*(*)** designation appears in a subprogram, it means that the length of the **character** variable (*identifier*) will be determined separately. But if you try to assign a value to such a variable, and the variable is ***not*** a dummy argument, you get this error. That's because there's no way to get length information for the variable unless it is a dummy argument. For example, the following fragment is incorrect because it assigns a value to **text** before determining **text**'s length:

```
subroutine string()
character*(*) text
text = 'Just the facts, ma''am'                    {Wrong!}
```

20 ERROR *Token* `right parenthesis expected`

This error can occur if you omit a closing right parenthesis in a statement, or if you make a syntax error earlier in the statement that causes the compiler to think you've started another statement. For example, the following fragment causes this error because there's no period after the relational operator **.or.** :

```
if ((reply .eq. 'N') .or (reply .eq. 'n')) again=.false.
```

Chapter 9. Diagnostic Messages

21 ERROR *Identifier* dummy argument appears more than once in list

Identifier already appears in the list of dummy arguments for this subprogram. It is an error to repeat it.

22 ERROR *Token* identifier or asterisk expected

Several conditions can cause this error. You might have placed an extra comma in a subprogram heading or call, so the compiler expects to find another dummy argument or an asterisk (to indicate an alternate **return**). Instead, it has found *token*.

This error can also occur if you mistype a substring assignment. If the substring assignment is the first executable statement in the program unit, and you use a comma instead of a colon to indicate the range of the assignment, you get this error. For example:

```
character*10 first, last
{other specification statements}
      . . .
first(1,3) = last(4:6)                    {Wrong!}
```

23 ERROR *Identifier* identifier expected

The compiler is expecting an identifier, but you haven't supplied one. Possibly, you omitted the asterisk in a data type name (**integer4** or **character10** rather than **integer*4** and **character*10**), or you listed more constants than identifiers in a **parameter** statement. You also might have unbalanced parentheses; for example:

```
data truth, (matrix(1,1)/.true., 0/
```

Since there are two left parentheses but only one right, FORTRAN expects more to this identifier name. Instead, it encounters the slash (/) that says to start loading data into the identifiers listed.

24 ERROR *Token* left parenthesis expected

You omitted a left parenthesis from a statement that requires one. Many I/O statements require parentheses; see the listings for the individual I/O statements in Chapter 4 to determine which ones do. This error occurs if, for example, you try to **close** a file without enclosing the unit identifier in parentheses, or if you use the **print** statement's syntax in a **write** statement; that is:

```
write *, 'This statement is wrong'
```

instead of

```
write (*,*) 'This statement is right'
```

The error also occurs if you forget to put parentheses around a format specification in an I/O statement; for example,

```
read (*, 'I4') int_num
```

instead of

```
read (*, '(I4)') int_num
```

25 ERROR *Token* `end-of-statement expected`

A number of conditions can cause this error. You probably omitted some punctuation—or added some extra—and now the compiler assumes you're trying to start a new statement without having ended the previous one.

26 ERROR *Token* `variable expected`

This error occurs if you use a constant where FORTRAN expects a variable. For instance, the following is incorrect because once **count** is assigned a value in a **parameter** statement, it is a constant and cannot be used where a variable is needed in the **do** loop:

```
parameter (count = 100)
      .
      .
      .
do count = 1,10                          {Wrong!}
       .
       .{statement}
       .
enddo
```

27 ERROR *Token* `equal sign expected`

This error can occur if you list a valid I/O attribute in an I/O statement, but forget to put an equal sign between the attribute and its value. For example:

```
open(4, file_name='my_secret', access 'sequential')
```

28 ERROR *Identifier* `variable already dimensioned`

You already provided dimensioning information for *identifier*. It is an error to repeat that information.

29 ERROR `Too many dimensions--maximum is 7`

Domain FORTRAN arrays can contain a maximum of seven dimensions. You exceeded that limit.

30 ERROR `Assumed bound only allowed as upper bound of the last dimension`

You can only use an asterisk (*) to designate an assumed size for the upper boundary of the last dimension in an array. It is an error to use the asterisk designation for the upper boundary of dimensions other than the last, or for the lower boundary of any dimension. See Chapter 5 for more information on assumed boundaries in arrays.

31 ERROR `Constant subscript required`

When assigning a value to an array element in a **data** statement, you must use a constant or symbolic constant to designate the element. In this case, you used a variable.

32 ERROR *Token* `lower bound must be <= to upper bound`

The value of *token*'s lower boundary must be less than or equal to that of its upper boundary. For example, the following is illegal:

```
real*4 price_list(25:10)
```

33 ERROR `Non-constant dimension prohibited on non-dummy array:` *token*

It is illegal for dummy arguments to appear in certain statements within a subprogram. Section 5.1 lists those statements. You probably used a dummy array argument in one of those statements.

34 ERROR *Token* `illegal data type`

Token is not one of Domain FORTRAN's valid data types. See Section 3.1 for a list of the valid type names.

35 ERROR *Token* `left parenthesis or integer constant expected`

The compiler has found a valid data type name, followed by an asterisk, but now needs either an integer constant to designate the variable's size, or a left parenthesis. For example, both of the following trigger this error:

```
character* employee_name                            {Wrong!}
integer*   employee_num                             {Wrong!}
```

Any of the following are correct:

```
character*(20) employee_name
character*20   employee_name
integer*4      employee_num
integer        employee_num
```

36 ERROR *Token* `character length must be positive and less than 32768`

The length of a **character** variable must be positive and no more than 32768 bytes. *Token* either is a negative length, or is more than 32768. For example, both of the following are incorrect:

```
character*(-15) too_short                    {Wrong!}
character*32769 too_long                     {Wrong!}
```

37 ERROR *Token* `illegal range of letters`

You must specify a forward range of letters when you use an **implicit** statement. This error occurs when you specify a backward range. For example:

```
implicit logical (c-a)   {Invalid range of characters}
```

38 ERROR *Token* `single letter expected`

You indicated that you were going to specify a range of letters in an **implicit** statement because you used a hyphen (–), but you didn't finish the range. In that case, the compiler assumes you meant to designate a single letter and that the hyphen is incorrect.

39 ERROR *Token* `letter appears more than once in IMPLICIT statement`

You used a letter more than once in an **implicit** statement. This error can occur even if you haven't used the letter *explicitly* multiple times. For example, the following triggers the error because the letter 'b' occurs both explicitly and within the specified range:

```
implicit double precision(b, a-c)
```

43 ERROR *Token* `slash expected`

You may have omitted a slash in a **data** or **common** statement. Another possibility is that you included an extra parenthesis in one of those statements, and so made the compiler think that the first slash it enountered was part of the previous identifier name.

44 ERROR *Identifier* `illegal appearance of dummy argument`

It is illegal for dummy arguments to appear in certain statements within a subprogram. Section 5.1 lists those statements. You probably used a dummy argument (*identifier)* in one of those statements.

45 ERROR `Comma expected`

This error can occur if you include formatting information in an I/O statement, but forget to enclose the format in parentheses within quotation marks. For example, this statement causes the error because there are no quotes around the format specification:

```
decode (30,(3F6.2),data) x,y,z                    {Wrong!}
```

46 ERROR *Identifier* `variable having substring is not of type character`

You specified a substring range for *identifier*, but it is some type other than **character**. You can specify subranges for **character** variables only.

47 ERROR `Variable and its associated initial value have inconsistent data types`

You are attempting to assign a value to a variable that is incorrect given the variable's data type. For example, the following is incorrect because it attempts to assign **logical** constants to **real** variables:

```
real a, b
data a, b/.true., .false./                        {Wrong!}
```

FORTRAN does allow you to assign values to variables with differing data types in many cases. See the listing for "Assignment Statements" in Chapter 4.

48 ERROR `Integer expression expected`

It is an error if you don't supply an integer or integer expression for the dimensions of an array.

49 ERROR `Positive integer constant or symbolic name of constant expected`

When you use a repeat count in a **data** statement, the repeat count must be a positive integer constant or the name of such a constant. For example, both of the following are incorrect:

```
data oops /-3*2/                                  {Wrong!}
data oops_again /1.0*2/                           {Wrong!}
```

The first is an error because in effect it says to assign the value 2 to **oops** -3 times, which clearly is impossible. The second is an error because 1.0 is not an integer.

50 ERROR *Identifier* `invalid use of procedure name`

It is an error for a subroutine name to appear in an executable statement. This is different from the rule for function names, which *must* be assigned a value somewhere in the function body. For instance, in the following, the function on the left is correct, while the subroutine on the right is wrong:

```
{   Right!   }                    {   Wrong!   }
function real_func(x)           subroutine real_sub(y)
        .                                .
        .                                .
        .                                .
real_func = value               real_sub = value
```

You probably got this error because you forgot that *identifier* was the name of a subroutine, and you tried to assign a value to it. See Chapter 5 for more information on subroutines and functions.

51 WARNING `Subscript/substring expression out of bounds`

You specified an array subscript or a substring expression that is beyond the boundaries defined for the array or substring. This warning only occurs if you compile with **–subchk.**

52 ERROR `Colon expected`

This error occurs if you use a **data** statement to assign a value to a substring, but forget the colon in specifying the substring range. For instance, the following is wrong because there's a comma instead of a colon in the substring range:

```
character*20 detective_name
data  detective_name(1,4) /'Nick'/              {Wrong!}
```

This is the correct way to write the **data** statement:

```
data  detective_name(1:4) /'Nick'/
```

53 ERROR *Token* `inconsistent substring bounds`

This error occurs if you specify a substring range in which the lower boundary is greater than the upper boundary. For example, this is incorrect:

```
print *, name(5:3)
```

Chapter 9. Diagnostic Messages

54 WARNING or ERROR

`Referencing subroutine as function` *identifier*

You called subprogram *identifier* as a subroutine earlier in this program unit, and now you are calling it as a function. It can't be both.

For example, you might have done something like this:

```
call return_answer()
          .
          .
          .
result = return_answer()                    {Wrong!}
```

55 WARNING or ERROR

Identifier `subroutine name appeared in type statement`

Identifier appears as the name of your subroutine in a **call** statement, but *identifier* also appears in an explicit data type statement. Although you must declare a function name if it won't default to the data type you want, you cannot declare a subroutine name.

56 WARNING or ERROR

`Referencing function as subroutine` *identifier*

You called subprogram *identifier* as a function earlier in this program unit, and now you are calling it as a subroutine. It can't be both. For example, you might have done something like this:

```
result = return_answer()
          .
          .
          .
call return_answer()                        {Wrong!}
```

57 WARNING *Identifier* `constant substring out of bounds`

This warning occurs if you specify a string subrange that is beyond the boundaries you defined for *identifier*.

58 ERROR *Identifier* `pointer based variable cannot be initialized`

If a variable appears in the *based_var_list* of a **pointer** statement, it cannot then subsequently appear in a **data** statement. For example, the following is incorrect:

```
pointer /my_ptr/ car_name, year, price
data car_name /'T_bird'/                    {Wrong!}
```

See the listing for **pointer** in Chapter 4 for more information.

59 WARNING `No subscript checking for array`

You referred to a specific element in a subprogram's assumed-length array. Since the array's length is assumed, there's no way for the compiler to know whether the element you specified is valid. This warning only occurs if you compile with **-subchk**. The following simple example shows how this warning might occur:

```
subroutine send_array(assume)
integer*4 assume(1:*)                           {Wrong!}
     . . .
print *, assume(10)
```

61 ERROR *Token* `invalid specifier name`

Several conditions can cause this error. You might have specified a nonexistent I/O attribute, or misspelled the name of one that does exist. See the listing for "I/O Attributes" in Chapter 4 for information on all valid Domain FORTRAN I/O specifiers.

This error also can occur if you use an assignment statement or implied **do** loop where you should use a format description. For example, in the following, the misplaced parentheses make the compiler think that there should be a format specification after the asterisk. Instead, there's an implied **do** loop:

```
read (*, int_array(i), i=1,10)
```

Either of the following ways is the right way to do it:

```
read *, (int_array(i), i=1,10)                  {1st way}
read (*,*) (int_array(i), i=1,10)               {2nd way}
```

64 ERROR `Cannot open file`

The file you specified in an **%include** compiler directive does not exist. Check to make sure you typed the filename correctly. See the listing for "Compiler Directives" in Chapter 4 for more information.

65 ERROR `Identifier assumed size array may not appear unsubscripted in I/O list`

If an array in a subprogram has an assumed size, you cannot list that array in a **read**, **write**, or **print** statement unless you also provide a subscript for the array. Section 5.7.2 gives more information on assumed size arrays. Here's an example of this error:

```
subroutine x(a_array)
integer*4 a_array(1:*)
    .  .  .
print *, a_array                                {Wrong!}
```

66 ERROR `Variable in adjustable dimension must be dummy argument or in common: identifier`

If you use a variable name in a subprogram to provide dimensioning information for an array, that variable either must be one of the subprogram's dummy arguments, or be in a **common** block. That is, the following is incorrect:

```
subroutine pass_array(matrix)
integer*2 i, j
real*4    matrix(i,j)                           {Wrong!}
```

Following is one correct way to use variables in adjustable dimensions:

```
subroutine pass_array(matrix, i, j)
integer*2 i, j
real*4    matrix(i,j)
```

67 ERROR `Arrays not allowed in adjustable dimensions: array_name`

In a subprogram, you can't specify the name of a dummy array as one of that array's adjustable dimensions. That is, the following is incorrect:

```
subroutine x(y,z)
integer*4  z(z)                                 {Wrong!}
```

68 ERROR `Internal compiler error - adjustable dimension for array`

The error is in the compiler, not in your code. Please contact your customer support representative.

69 ERROR *Token* `integer variable or constant expected`

The compiler was expecting an integer variable or integer constant, but has found something else. Possibly you specified an array as part of the dimensioning information for a second array. For example:

```
integer*4 int(3), jint
    . . .
real*8 strength_readings(int)                    {Wrong!}
```

70 ERROR *Expression* `complex and double precision operands are incompatible types`

It is an error to perform a mathematical operation on operands when some are **complex** and others are **double precision**. *Expression* tries to do that.

For example, the following is incorrect:

```
complex          comx
double precision dbl, result
result = comx + dbl                              {Wrong!}
```

73 ERROR *Token* `invalid statement label`

A statement label must consist of digits only. *Token* includes some other characters.

76 ERROR *Token* `variable name or common block name enclosed in slashes expected`

In a **save** statement, you must supply either a variable name or a **common** block name. In this case, however, the compiler found *token*, which probably is a constant. For example, the following causes this error:

```
subroutine mysub()
save 100                                         {Wrong!}
```

77 ERROR `Comma or end-of-statement expected`

You probably misspelled a keyword or left out a comma, and now the compiler isn't sure whether you mean to continue the statement it is parsing, or end the statement and start another. The following causes this error because there's no comma between the format line number (20) and the first variable name:

```
print 20 pi, cake
```

This error also can occur if you let a statement label extend past the fifth column.

78 ERROR `Dummy argument may not appear in SAVE or DATA statement`

A dummy argument that appears in an **entry** statement, appears earlier in this subprogram in either a **save** or **data** statement. The argument cannot appear in both places. For example, the following triggers this error:

```
subroutine x(y)
save y
      .
      .
      .
entry z(y)                                     {Wrong!}
```

80 WARNING `Identifier not explicitly typed` *identifier*

You compiled with the **-type** option to check whether you had explicitly declared all your variables. You didn't declare *identifier*.

81 ERROR `If function or entry is character, all entries must be character`

If a function, or any **entry** statement in a function, is of type **character**, all the **entry** names and the function must also be of type **character**. For instance, the following is *not* correct

```
character*20 function ret_char(x,y)
      . . .
entry here(x)                                  {Wrong!}
      . . .
entry there(y)                                 {Wrong!}
```

but this is correct:

```
character*20 function ret_char(x,y)
character*20 here, there
      . . .
entry here(x)
      . . .
entry there(y)
```

82 ERROR `If function or entry is character*(*), all entries must be character*(*)`

If a function, or any **entry** statement in a function, is of type **character*(*)**, all the **entry** names and the function must also be of type **character*(*)**.

For instance, the following is *not* correct:

```
character*10 function return_char(x,y)         {Wrong!}
character*(*) here
```

```
    . . .
entry here(x)
    . . .
entry there(y)                                    {Wrong!}
```

but this is correct:

```
function return_char(x,y)
character*(*) return_char, here, there
    . . .
entry here(x)
    . . .
entry there(y)
```

83 ERROR `Pointer variable must be integer*4:` *identifier*

After you declared that *identifier* was a **pointer** variable, you explicitly typed it to be something other than an **integer*4**. A **pointer** must be an **integer*4**. For example, the following causes this error:

```
pointer /my_ptr/ tv_show, male, female, ratings
real*4 my_ptr                                    {Wrong!}
```

84 ERROR `Pointer variable cannot be dimensioned:` *identifier*

A **pointer** variable must have the **integer*4** data type; it cannot be an array of **integer*4**s (or any other data type, for that matter). However, you provided dimensioning information for *identifier* in a separate statement. For example, the following causes this error:

```
integer*4 points(10)
pointer /points/ a, b, c                         {Wrong!}
```

85 WARNING *Identifier* `redundant type declaration`

You already have explicitly declared *identifier*. You don't need to do so again.

88 ERROR Only local and common variables can be initialized

In a subprogram, you can only initialize local or **common** block variables. It is an error to initialize a subprogram name.

89 ERROR `Conditional compilation not balanced`

You used the Domain FORTRAN conditional compilation directives **%if...%then** and possibly **%else**, but you forgot to end the structure with an **%endif**. See the listing for "Compiler Directives" in Chapter 4 for more information.

90 WARNING `Conditional compilation user warning`

You triggered a warning-level problem through misuse of the conditional compiler directives. A more specific message follows this one. See the listing for "Compiler Directives" in Chapter 4 for information on using the **%warning** compiler directive.

91 ERROR `Conditional compilation user error`

You triggered an error-level problem through misuse of the conditional compiler directives. A more specific message follows this one. See the listing for "Compiler Directives" in Chapter 4 for information on using the **%error** compiler directive.

93 ERROR *Token* `character variable or array element expected`

There are certain conditions under which you must use **character** variables or array elements. For example, several I/O attributes take **character** variables as their arguments. If you try to use an argument with a different data type, you get this error. See the listing for "I/O Attributes" in Chapter 4 for more information.

94 ERROR *Token* `logical variable or array element expected`

There are certain conditions under which you must use **logical** variables or array elements. For example, the **exist**, **named**, and **opened** I/O attributes take **logical** variables as their arguments. If you try to use an argument with a different data type, you get this error. See the listing for "I/O Attributes" in Chapter 4 for more information.

95 ERROR *Token* `integer variable or array element expected`

There are certain conditions under which you must use **integer** variables or array elements. For example, the **recl** and **strid** I/O attributes take **integer** variables as their arguments. If you try to use an argument with a different data type, you get this error. See the listing for "I/O Attributes" in Chapter 4 for more information.

96 WARNING *Token* `format should be character array or expression`

You put something other than formatting information in the part of an I/O statement where Domain FORTRAN was expecting to find a format. That format must take the form of an expression—for example, '(3F6.2)'—or a **character** variable or array that holds a format.

103 ERROR *Token* `comma or identifier expected`

You should have supplied a comma or an identifier, but the compiler found *token* instead. Probably you did something like this:

```
    do 25=1,10                               {Wrong!}
        {do loop body}
25  enddo
```

If there were a comma and identifier after 25 in the **do** statement, that would be acceptable because 25 would then be a statement label. And if there were an identifier in place of the 25, that would be acceptable because each **do** loop must have an index variable. Since neither is true, this error occurs.

104 ERROR *Token* `comma or right parenthesis expected`

Several conditions can cause this error. You might have omitted a comma in an actual or implied **do** loop or left your parentheses unbalanced in a statement. The following causes the error because the asterisk in the data indicates to repeat the assignment, but there's no looping information for the array—that is, no comma after the array name to indicate how to repeat the assignment:

```
data matrix(i,j) /20*0/
```

105 ERROR `Keyword THEN expected`

You did not include the keyword **then** in a block **if** statement. Such a statement must follow this form:

if (*expression1*) **then** `{Notice keyword then}`
  stmt1
else if (*expression2*) **then** `{Here it is again}`
  stmt2
  .
  .
  .
else
  stmtN
end if

106 ERROR `Keyword TO expected`

You omitted the keyword **to** in an **assign** statement. For example, the following causes this error:

```
assign 10 error
```

The correct way is:

```
assign 10 to error
```

107 ERROR `RETURN statement used in wrong module`

A **return** statement can only appear in a subroutine or function. You have included it in some other type of program unit. See the listing for **return** in Chapter 4 for more information.

108 ERROR `Integer, real or double precision variable expected`

The compiler was expecting an **integer**, **real**, or **double precision** variable, but found a variable of another type. Probably you specified a **logical** or **character** variable as the index variable of a **do** loop.

109 ERROR *Identifier* `illegal use of dimensioned variable`

You used an array someplace where a simple variable is required. For example, you might have tried to use an array as the index of a **do** loop like the following example does:

```
integer*4 index(10)
. . .
do index=1,10                            {Wrong!}
    print *, 'This loop won''t work'
enddo
```

112 ERROR `Integer or character constant, or end-of-statement expected`

You included something other than integers or a character string in a **stop** or **pause** statement. For example, the following causes this error:

```
stop 1.0
```

See the listings for **stop** and **pause** in Chapter 4 for more information.

113 ERROR *Token* `character expression expected`

You tried to assign *token* to a **character** variable, but *token* isn't a character string or expression. For instance, this error occurs if you try to assign an integer or logical constant to a **character** variable.

115 ERROR *Label_num* `DO termination label already defined`

You specified *label_num* as the statement label that marks the termination point of this **do** loop, but *label_num* already appears elsewhere in this program unit. You cannot reuse numbers within a program unit. For example, the following fragment causes this error because label 10 appears twice:

```
10    print *, 'Here''s the first use of label 10'
      . . .
      do 10, i = 1, SIZE      {Label 10 has been}
         print *, i           {used already.    }
10    enddo
```

This fragment also causes a "duplicate statement label" error (Error 127).

117 ERROR `Continuation line not preceded by initial line`

There is a character in column 6, indicating that this is a continuation line, but there is no statement part in the preceding line(s). See subsection 2.1.10 for information on column conventions and continuation lines.

118 ERROR `Too many continuation lines`

You have exceeded the maximum number of continuation lines that FORTRAN allows.

119 ERROR `Statement appears out of sequence`

Chapter 2 explains the rules for statement order in a Domain FORTRAN program. Your program breaks one of the rules. Probably you have a specification statement after an executable statement, or you have a data type statement after a specification statement. For example, the following causes the error because the **data** statement (a specification statement) appears among the data typing statements:

```
integer*4 int_num, count
real      dec_num
data      int_num, count, dec_num /0,0,0.0/
logical   again                              {Wrong!}
```

This error also occurs if you forget to put an **end** statement at the termination of one program unit and then begin another. That's because a **subroutine**, **function**, **program**, or **block data** statement, if present, must always be the first statement in its program unit. Without an **end** to signal the termination of one unit, the compiler thinks any unit-heading statement it encounters is out of order. For example:

```
program simple
integer*4 i
do i = 1,10
    call my_sub(i)
    print *, i
enddo

subroutine my_sub(j)            {No end in the main}
integer*4 j                     {program,so this is}
j = j*2                         {an error.         }
end
```

120 WARNING `Columns 1-5 of a continuation line must be blank`

If there is a character in column 6, FORTRAN considers the entire line to be a continuation line. In such a case, the first five columns of the line must be blank. You probably did one of two things: accidentally put part of a statement label in column 6, or put what you meant to be the continuation character in one of the first five columns.

121 ERROR `Unrecognizable statement`

This error means Domain FORTRAN can't determine what kind of statement it is parsing. A number of conditions can cause this error. You might have omitted a keyword from a statement, or let a statement label extend past column 5, or misspelled a keyword. If you can't readily tell what caused this problem, consult Chapter 4 for more information on the keywords in your statement.

122 ERROR `This statement cannot be the dependent statement of a logical IF`

The dependent portion of a logical **if** statement is the action that is executed when the logical expression is true. It must be a simple executable statement. That is, it cannot be a specification statement such as **data** or **save**, nor can it be a block **if** statement or any other executable statement that actually is a structure of statements. The following, for example, is illegal because the dependent part of the **if** is a **do** statement structure:

```
if (i .lt. j) do count=1,10                        {Wrong!}
    print *, 'The Count says: ', count
    enddo
```

123 ERROR `Statement must start no earlier than column 7`

The compiler found an executable statement that begins and ends before column 6 (the continuation column). Executable statements must begin no earlier than column 7.

124 ERROR `This statement cannot be the terminal statement of a DO loop`

While a **do** loop is not required to end with an **end do** statement, some statements *cannot* terminate the loop. However, the label you specified in your **do** statement indicates that one of the illegal statements ends the loop. See the listing for **do** in Chapter 4 for more information on which statements are not permissible for ending a **do** loop.

125 ERROR `Increment value of implied DO loop cannot be zero`

As with regular **do** loops, you can specify an increment value in an implied **do** loop. However, that increment value cannot be zero. See the listing for **do** in Chapter 4 for more information.

126 WARNING `Missing END statement`

Each program unit must end with an **end** statement. That statement does not appear at the end of this program unit. Domain FORTRAN supplies the **end** if you don't, which is why this is a warning instead of an error, but you should explicitly insert an **end**.

127 ERROR *Label_num* `duplicate statement label`

You already have used the statement label *label_num* elsewhere in this program unit, and it is not permissible to use a label more than once within a unit.

128 ERROR `Unit specifier required`

Most of the FORTRAN I/O statements require that you specify the unit on which the I/O is to be done. Often the *unitid* argument is the first one listed in an I/O statement, though it need not be. However, you listed one or more arguments without specifying the unit. For example, the following causes the error:

```
open (iostat=open_stat,status='old',recl=80)    {Wrong!}
```

The following is correct, because it specifies '4' as the unit specifier:

```
open(4, iostat=open_stat, status='old', recl=80)
```

131 ERROR *Label_num* statement label was previously referenced as a format

A statement label can be attached to an executable or nonexecutable statement. The first reference to *label_num* in your program indicates that the label is attached to a **format** statement, but this current reference is to an executable statement. Consider the following:

```
print 100, artist_name, painting_name
   . . .
goto (80, 90, 100) int_var                        {Wrong!}
```

In this example, the **print** statement says that label 100 designates a **format** statement for the variables **artist_name** and **painting_name**. The computed **go to** designates that if **int_var** equals 3, control should transfer to the executable statement labeled 100. (See the listing for **go to** in Chapter 4 for more on computed **go tos**.) The two designations are incompatible—if 100 is at a **format** statement, the **go to** can't jump there, while if 100 is at an executable statement, the **print** won't find the formatting information it needs.

133 ERROR *Label_num* statement label was previously referenced as executable

A statement label can be attached to an executable or nonexecutable statement. The first reference to *label_num* in your program indicates that the label is attached to an executable statement, but this current reference is to a nonexecutable statement (probably a **format** statement). Consider the following:

```
open(4, iostat=int_var, err=50)
   . . .
print 50, num                                     {Wrong!}
```

In this example, the **open** statement says that label 50 designates an executable statement to which the program should jump if there is an error on the open. The **print** statement designates that 50 is a nonexecutable **format** statement. The two designations are incompatible—if 50 is at a **format** statement, the **open** can't jump there, while if 50 is at an executable statement, the **print** won't find the formatting information it needs.

136 ERROR Either unit or file specifier must be present

In an **inquire** statement, you must specify either the unit or file about which you are inquiring. Often the unit or the file, whichever one you choose, is the first argument listed in the **inquire**, though neither of them need to be first. However, you listed one or more arguments without specifying a unit or file.

137 ERROR `Unit and file specifiers must not both be present`

In an **inquire** statement, you must specify either the unit or the file about which you are inquiring. It is an error to specify both, but your **inquire** statement does include both. Delete one to remove this error.

139 ERROR *Token* `unsigned integer constant expected`

You probably forgot to include the field width for one of the repeatable edit descriptors (*token*) in a **format** statement. You are required to give width information. The width must be an integer. See the listing for **format** in Chapter 4 for more information.

140 ERROR `Cannot represent integer constant`

You specified a **real** or **double precision** value for an **integer** variable, and that value falls outside the range of the **integer** variable. Chapter 3 gives the ranges of the two Domain FORTRAN integer data types. For example, the following causes this error:

```
integer*4 too_big
too_big = 2147483648.0                           {Wrong!}
```

141 ERROR `Statement may not appear in a block data subprogram`

A **block data** subprogram can only be used to initialize values in a **common** block, and therefore it cannot contain any executable statements. You probably included an executable statement in the subprogram. See Section 5.5 for more information on **block data** subprograms.

142 ERROR *Identifier* `identifier has multiple or inconsistent attributes`

This error occurs if you specify that a dummy argument which corresponds to a variable actual argument is a constant. You could do that by using the dummy argument in a **parameter** statement within the subprogram. For example:

```
integer*4 vary
    . . .
call my_sub(vary)
    . . .
subroutine my_sub(dummy_vary)
integer*4 dummy_vary
parameter (dummy_vary=100)                       {Wrong!}
print *, dummy_vary
end
```

143 ERROR *Identifier* `identifier is too long`

Domain FORTRAN allows identifiers to have a maximum of 4096 characters. *Identifier* exceeds that maximum.

144 ERROR *Identifier* `multiple conflicting type declarations`

You already have declared *identifier* in this program unit *and* you've declared it to be a different data type. For example, the following triggers this error:

```
real*8 my_num
     . . .
integer*4 my_num                                 {Wrong!}
```

145 ERROR *Identifier* `name of subroutine, main program, or block data subprogram cannot appear in a type statement`

You used *identifier* as the name of a main program, subroutine, or block data subprogram *and* as the name of a variable. It is an error to use it both ways.

146 ERROR *Identifier* `variable must be typed before use in adjustable dimension`

If you use *identifier* to provide dimensioning information for an adjustable array, you cannot then explicitly declare *identifier* in a data type statement. For example, the following is *not* correct:

```
subroutine my_sub(matrix,i,j)
real       matrix(i,j)
integer*2 i,j                                    {Wrong!}
```

However, it is permissible to explicitly type *identifier* before you dimension the array. That means the following is correct:

```
subroutine my_sub(matrix,i,j)
integer*2 i,j
real       matrix(i,j)
```

147 ERROR `Common block name cannot appear in a type statement`

It is an error for a **common** block name to appear in an explicit data type statement.

148 ERROR *Identifier* `not an intrinsic function name`

Identifier, which you specified as being an intrinsic function, is not one of Domain FORTRAN's available intrinsics. See Appendix C for a list of all the intrinsic functions.

150 ERROR `Compiler internal error - operand stack overflow`

The error is in the compiler, not in your code. Please contact your customer support representative.

151 ERROR *Token* `invalid operator`

A number of conditions can cause this error. You might have mistakenly used an operator that another programming language supports but FORTRAN does not (for example, Pascal's := or C's !=). See Section 4.2 for a list of the operators available in Domain FORTRAN.

This error also can occur if you omit one of the single quotes around a string in a **print** statement, or if the string extends past the 72nd column.

Both of the following statements return an "invalid operator" error:

```
if (i != j) print *, 'Oops'                          {Wrong!}

print *, Missing the left quote on this string' {Wrong!}
```

153 ERROR *Token* `invalid comparison of operands of complex data type`

It is an error to use these relational operators on **complex** variables: **.gt.**, **.ge.**, **.lt.**, **.le.** However, *token* is one of these operators. See the listing for "Assignment Statements" in Chapter 4 for more information.

154 ERROR *Token* `invalid appearance of substring`

You tried to assign a value in a **data** statement to a substring. However, the variable for which you specified a substring isn't of type **character**. For instance, the following causes this error, because **integer** variables don't have substrings:

```
integer*4 no_strings
data no_strings(2:3)/34/                        {Wrong!}
```

155 ERROR *Token* `invalid appearance of subscript or argument list`

You used a subscript or argument someplace you weren't supposed to. You might have specified a subscript instead a substring range for a **character** string; that is, given that the variable **first_name** is defined as a **character*10**, you might have specified the following

```
first_name(1) = 'A'
```

instead of

```
first_name(1:1) = 'A'
```

156 ERROR `Identifier is not a function name`

It is an error to use a main program name as an argument in a subprogram call.

157 ERROR *Token* `wrong number of dimensions` *array_name*

You specified one number of dimensions when you declared *array_name* and another number in this statement. The numbers of dimensions must match. For example, the following causes this error because **sales_figures** is declared with two dimensions, but the assignment statement only lists one:

```
real*4 sales_figures(10,10)
    . . .
do i=1,10
    sales_figures(i) = 0.0                        {Wrong!}
enddo
```

158 ERROR *Token* `invalid subexpression`

The expression enclosed in parentheses is invalid for some reason. For example, either of the following causes this error:

```
a = (var_name,1)                                  {Wrong!}
a_num = (1,1,1)                                   {Wrong!}
```

In the first statement, the compiler thinks it is parsing an arithmetic expression because **var_name** is a variable. But in that case, there should be an operator after **var_name**. Instead, there's a comma. The second statement is incorrect because the first two constants make the compiler assume it is assigning a **complex** value to **a_num**. But then it encounters the third constant and doesn't know what to do with it.

159 ERROR *Token* `missing operand`

The compiler found a valid operator but can't find a valid operand. You might have omitted the operand completely, or let the operand name extend past column 72. Another possibility is that you used the slash (/) edit descriptor incorrectly so that the compiler thinks the slash indicates division instead of formatting.

160 ERROR *Token* `missing operator`

The compiler thinks it is parsing some sort of assignment statement, but it can't find a valid operator. Several conditions can cause this error. You might have unbalanced parentheses in a block **if** statement, or misplaced quotes in an I/O statement. Both of the following statements cause this error, the first because of a missing right parenthesis, and the second because the contraction in the string doesn't contain two single quotes:

```
if ((i .lt. j) .and. (i .gt. 0) then      {Missing right }
    {statement}                           {parenthesis.  }
endif

print *, 'This won't work'                {Missing quote.}
```

161 ERROR *Token* `invalid operand`

This error occurs if you try to assign a value to a variable whose type is incompatible with the value. For example, it's an error to assign a character constant to a **real** variable, or the **logical** constant **.true.** to an **integer** variable.

The error also can occur if you use *token* in incompatible ways. The following triggers this error when the compiler parses the second statement because **index** already has been used as a subroutine name:

```
call index(arg1, arg2)
    . . .
int_num = int_num * index                        {Wrong!}
```

162 ERROR *Token* `expression expected`

The compiler is expecting an expression at *token*, but you haven't supplied one. This error occurs if you omit an expression from an **if** statement, or **do while** statement, or other executable statement that requires an expression.

163 ERROR *Token* `not a constant expression`

The compiler is expecting a constant or an expression that resolves to a constant, but *token* isn't one. You might have specified a variable name someplace a constant is required. You also might have specified that an assignment be repeated, but left out looping information. For example, the following is incorrect because the asterisk indicates that the assignment be repeated, but there's no implied **do** loop:

```
real*4 avg_table(4,5)
data avg_table(i,j) /20*0/                       {Wrong!}
```

This is a correct way to use an implied **do** loop:

```
data ((avg_table(i,j), i=1,4), j=1,5) /20*0/
```

However, this is an even easier way to initialize the array elements:

```
data avg_table /20*0/
```

164 ERROR *Token* `expression is of incorrect data type`

The expression you supplied (*token*) produces a result that's the wrong data type for your statement. For example, the expression in a **do while** loop must evaluate to a **logical** result, so if your expression produces an arithmetic or other type result, you get this error. Similarly, an **if** statement's expression must yield a **logical** result. That means the following is incorrect:

```
if (int_num * real_num) then                {Wrong!}
     . . .
```

This error also can occur if you try to assign a constant to an incompatible variable with the **parameter** statement. For instance, you get this error if you assign the **logical** constant **.true.** to a variable that is some other data type.

165 ERROR *Identifier* `not an assignable element`

Probably, you are attempting to assign a value to an array, but haven't declared *identifier* as being an array; that is, you haven't provided dimensioning information.

This error also occurs if you try to make a substring assignment using the syntax for a regular array element assignment. With arrays, you use subscripts to indicate the location at which a value should be stored. With substrings, you must provide both the beginning *and* ending substring values, even if those values are the same. So this assignment is incorrect:

```
character*15 last_name, other_name
last_name(1) = other_name(1)                {Wrong!}
```

while this one is correct:

```
last_name(1:1) = other_name(1:1)
```

167 ERROR *Token* `expression syntax`

There's a syntax error, but the compiler can't determine what kind of error it is. The parentheses in your expression could be unbalanced, or you may be using a mathematical operator incorrectly.

168 ERROR *Identifier* `explicit typing of a symbolic constant must precede its definition`

You used *identifier* in a **parameter** statement, and now are explicitly defining its data type. You must reverse those steps; that is, if you are going to explicitly define an identifier's data type, you must do so *before* the identifier appears in a **parameter** statement.

169 ERROR *Token* `invalid use of unsubscripted array name`

You tried to assign a value to an array, but didn't specify the element to which you wanted the value assigned. For example, the following causes this error:

```
real*8 sonar_readings(100)
   . . .
sonar_readings = 53.2352                        {Wrong!}
```

171 ERROR INCLUDE `pathname must be enclosed in quotes`

You forgot to enclose the filename for the **%include** compiler directive in single quotes. You also get this error if you enclose the pathname in double quotes.

172 ERROR `ENTRY statement can appear only in subroutine or function subprogram`

You used the **entry** statement within a main program or **block data** subprogram. It is an error to do so. See the listing for **entry** in Chapter 4 for more information.

173 ERROR `ENTRY statement cannot appear within a block IF or DO loop`

You used the **entry** statement within the scope of a block **if** statement or within a **do** loop. It is an error to do so. Move **entry** outside the scope of the statement or loop.

174 ERROR *Token* `invalid complex constant`

You tried to assign an invalid value (*token*) to a **complex** variable. For instance, this error occurs if you specify a **character** string for one or both parts of the **complex** entity.

176 ERROR `Invalid use of character*(*) entity`

You cannot use a **character*(*)** variable as a concatenation operand if the compiler needs to know the size of an entity. For example, consider the following:

```
subroutine x(first_name, last_name)
character*10  first_name
character*(*) last_name
     . . .
call compute_name(first_name//last_name)        {Wrong!}
```

This program fragment is incorrect because there's no way for the compiler to know how much space to allocate for **compute_name**; the size of the argument **last_name** hasn't been determined.

177 ERROR *Identifier* `symbolic constant already defined`

Once you define a constant with the **parameter** statement, it is an error to redefine that same constant with another **parameter** statement in the current program unit. *Identifier* already has been defined in this unit.

178 ERROR *Token* `data type name expected`

One of the valid Domain FORTRAN data type names must appear directly after the keyword **implicit**. However, *token* is not a valid data type. For example, in the following, a typographical error causes the compiler to complain:

```
implicit read*8 (a-c)
```

179 ERROR *Token* `invalid usage of alternate return specifier`

FORTRAN only allows alternate **return** statements within subroutines, but not within functions. You specified alternate **returns** within a function. See the listing for **return** in Chapter 4 for more information.

181 ERROR *Token* `invalid usage of array declarator`

You listed dimensioning information for an identifier that isn't a variable. For example, you might have done the following:

```
subroutine n()
dimension n(4)                                  {Wrong!}
```

Since **n** already is the subroutine's name, you can't declare that it is an array.

182 ERROR `END specifier is not permitted in a WRITE statement`

Unlike the **read** statement, the **end=** I/O attribute is not valid for a **write** statement. See the listing for **write** in Chapter 4 for a list of that statement's valid I/O attributes.

183 WARNING `Unreachable statement`

The compiler warns you when it encounters a statement or statements that cannot be reached from any other statements in the program. An example of this condition is the following:

```
        print *, 'Starting here'
        go to 10
        print *, 'You can''t get here from anywhere'
10      print *, 'Unreachable code is wasted code'
```

184 ERROR `Too many arguments`

A Domain FORTRAN subprogram can have a maximum of 511 arguments. Your subprogram exceeds that limit.

191 ERROR *Token* `comma, colon, slash, or right parenthesis expected`

The compiler is expecting some sort of punctuation, but can't find the kind it needs. Several conditions can cause this error. You might have omitted a comma in a **format** statement, or accidentally typed a period instead of a comma, or left out a slash in a **data** statement, or forgotten to close your parentheses in any statement. For example, the following produces this error because there's no comma between **'first num ='** and **I2**:

```
        print 10, i, j
10      format ('first num =' I2, ' and second = ', I2)
```

201 ERROR *Token* `illegal format descriptor`

You used an invalid descriptor in a **format** statement. You probably did one of two things: you either used an incorrect character in a **format** statement, for example

```
format (U2)
```

instead of

```
format (I2)
```

or you omitted a quote in a string and so the compiler considers your string to be an edit descriptor. See the listing for **format** in Chapter 4 for a list of valid descriptors.

202	ERROR	`Bad character constant`

Probably you forgot to put a closing right parenthesis on a **format** statement, or you let a string in an I/O statement extend past column 72.

203	ERROR	`Period expected`

When specifying a format for real numbers, you must indicate how many digits are to appear after the decimal point. You do so by listing: 1) the edit descriptor, 2) the total number of characters in the field, 3) a period, and 4) the number of digits after the decimal point (for example, F6.2). You omitted the period in the format.

204	ERROR	`No closing parenthesis for format specifier`

You forgot the closing parenthesis when you specified the format in a **read**, **write**, or **print** statement. When you use a character expression to specify a format in those I/O statements, you must enclose the expression in parentheses enclosed in single quotes; for example, '(F6.2)'. The following statement causes this error because there's no right parenthesis before the closing quote:

```
read (*, '(3I4',) int1, int2, int3            {Wrong!}
```

205	ERROR	`P format not followed by F, E, D, or G`

The **P** edit descriptor designates a scale factor for real numbers, and is only valid with the **F**, **E**, **D**, or **G** edit descriptors. You mistakenly used it with some other descriptor.

209	ERROR	`Unlabeled FORMAT statement`

A **format** statement must have a label number. You forgot to label the statement.

219	ERROR	*Token* `too many constants`

There must be the same number of constants in a **data** statement as there are storage locations (variables and array elements) listed. You have more constants within the slashes (/ /) than you have storage locations before them. For example, you might have listed the name of an array but provided more data than the array can hold.

This error also occurs if you enter a **complex** number incorrectly. Such a number must be enclosed in parentheses; if you omit them, the compiler thinks it has two separate numbers. For example:

```
complex intricate
data    intricate /17.5,25.8/               {Wrong!}
```

221 ERROR *Identifier* `illegal appearance of entity in common`

If a variable already has been included in a **common** block, it cannot then appear in a **save** statement. That means the following causes this error:

```
common /ordinary/ i, j, k
save i                                        {Wrong!}
```

222 ERROR *Token* `invalid constant`

The constant (*token*) you provided for a variable in a **data** statement is not a valid FORTRAN constant. For example, this error occurs if your program includes the following because there's a negative sign in front of the constant **.true.**:

```
logical again
data    again /-.true./                       {Wrong!}
```

223 ERROR *Token* `constant expected`

You didn't provide a constant for a variable in a **data** statement. For instance, you might have done something like this:

```
integer*4 i, j
data i /-j/                                   {Wrong!}
```

In this case, **j** is not a constant; it is a variable.

224 ERROR *Identifier* `symbolic name of constant expected`

The compiler found an identifier listed as the value to be assigned to a variable in a **data** statement, and so assumed it must be a symbolic constant. However, it isn't. The following causes this error:

```
real*4 temp, normal_temp
data temp /normal_temp/                       {Wrong!}
```

The following is the correct way to use a symbolic constant in a **data** statement:

```
real*4 temp, normal_temp
parameter (normal_temp = 98.6)
data temp /normal_temp/
```

225 ERROR `Undefined label` *label_num*

You referred to *label_num* in an executable statement in your program, but no statement in this program unit has that label number. You probably forgot to label a statement, or mistyped a label number in your executable statement. This error also occurs if the statement in which you define *label_num* has some other error that prevents the compiler from generating code for the statement.

226 ERROR *Token* `left parenthesis or identifier expected`

In a **data** statement, it is permissible to string together groups of identifiers and constants. The syntax for doing so is the following:

data *var_list1 /constant_list1/, . . . , var_listN /constant_listN/*

After the keyword **data**, or after a comma, there must be either an identifier or a left parenthesis. However, the compiler has found *token* instead. For example, either of the following causes this error:

```
data 1/1/                                {Wrong!}
data int_num/10/, 5/10/                  {Wrong!}
```

See the listing for the **data** statement in Chapter 4 for more information.

233 ERROR *Token* `substring expression must be of type integer`

A substring expression must resolve to an integer value—for example, you can't access element 5–1/2 in a **character** string. However, *token* either is a constant of some type other than **integer**, or is an expression that does not resolve to an integer.

234 ERROR *Token* `incorrect operand data type`

The operand *token* is incompatible with the operator you specified. You probably used a relational operator to compare **logical** values. See the listing for "Assignment Statements" in Chapter 4 for more information.

235 ERROR *Token* `subscript expression must be of type integer`

A subscript expression must resolve to an integer value—for example, you can't access element 33–1/3 in an array. However, *token* either is a constant of some type other than **integer**, or is an expression that does not resolve to an integer.

236 ERROR *Token* incorrect number of arguments

You provided too many or too few arguments for a statement function or intrinsic function. For instance, the intrinsic function **sqrt** takes one argument, so if you list two, you get this error.

237 ERROR *Token* incorrect intrinsic argument data type

The intrinsic functions often require arguments with specific data types. For example, the **sqrt** intrinsic function takes a **real** argument, so if you provide an **integer** instead, you get this error. See Appendix C for a list of all intrinsic functions and the arguments they take.

240 ERROR This particular intrinsic function cannot appear as an argument

ANSI standard FORTRAN prohibits certain types of intrinsic functions from appearing as arguments in a subprogram call. You included one of those functions in your call. The types that are invalid for use as arguments are those that perform data type conversions, those that find the lexical relationship between entities, and those that find the largest and smallest of entities. See Appendix C for a list of all the intrinsic functions.

241 ERROR *Token* constant or implied DO index expected

You made an error when trying to assign values in a **data** statement to array elements. For example, the following is incorrect because the statement says to assign a value to the jth element of array **test_scores**, but then erroneously lists **k** as being the implied **do** loop index:

```
integer*4 test_scores(25), j, k
data (test_scores(j), k=1,25 /25*0/          {Wrong!}
```

242 ERROR *Token* subroutine or function passed as argument must appear in EXTERNAL statement

If a subroutine or function is one of the arguments in a subprogram call, you must declare the subroutine or function in an **external** statement. See the listing for **external** in Chapter 4 for more information.

244 ERROR Duplicate dummy argument

It is an error to repeat a dummy argument in a subprogram heading.

245 ERROR `Statement function cannot be passed as argument`

It is an error to pass a statement function as an argument in a subprogram.

247 ERROR `%INCLUDE, %LIST, %NOLIST, or %EJECT expected`

The compiler directive that you listed is not a valid one. The compiler thinks you meant to type **%include**, **%list**, **%nolist**, or **%eject**. See the listing for "Compiler Directives" in Chapter 4 for a list of Domain FORTRAN's available directives, and directions on using them.

251 ERROR `No matching DO statement`

Your program includes an **end do** statement, which indicates the termination point of a **do** or **do while** loop, but the compiler can't find the beginning of the loop. Make sure your loops match up. This error also occurs if there is an error in the **do** or **do while** statement that prevents the compiler from generating code for the statement.

252 ERROR *Token* `keyword WHILE expected`

The structure of your **do** statement makes the compiler think it is parsing a **do while**, but it can't find the **while**. You probably specified an expression that resolves to a logical value in your **do** statement, but omitted the keyword **while**. For example:

```
logical again
again = .true.
     . . .
do (again)                                  {Wrong!}
    {stmt}
enddo
```

253 ERROR `Missing END DO for DO statement on line` *num*

There are more **do while** and unlabeled **do** statements in your program than **end do** statements. You must explicitly close both types of **do** loops.

Domain FORTRAN associates an **end do** with the closest unclosed **do while** or unlabeled **do** loop. This means that, depending on your program's structure, *num* might appear misleading. In the following fragment, the **end do** closes the inner loop—regardless of what the program indentation indicates—since that's the closest unclosed loop:

```
do i = 1,10
    {stmt}
    do j = 1,5
        {stmt}
```

```
enddo                 {This closes the "do j" loop,  }
                      {despite the indentation style.}
```

If you compile this fragment, *num* indicates that the **do i** is not closed, even though at first glance you might think it is.

265 ERROR *Token* `control list specifier expected`

You were supposed to provide one of the I/O attributes for the control list in an I/O statement, but provided *token* instead. See the listing for the I/O statement in which this error occurred for a list of the valid attributes. For example, the following causes this error:

```
write (*, *, 3) my_var
```


266 ERROR `Duplicate specifier`

You repeated a specifier in an I/O statement. For example, the following triggers this error because **access='sequential'** appears twice:

```
 open (10, file=myfile, access='sequential',
+      status='unknown', access='sequential')
```

267 ERROR *Token* `integer*4 variable or array element required`

Several I/O attributes require an argument that's either an **integer*4** variable or array element. *Token* is not the correct type. See the listing for "I/O Attributes" in Chapter 4 for more information on the required argument types.

268 ERROR `REC= and END= may not appear in same control list`

The I/O attributes **rec=** and **end=** may not both appear in a **read** statement. See the listing for **read** in Chapter 4 for more information.

269 ERROR `REC= not allowed with list directed I/O`

If you designate list-directed I/O in a **read** or **write** statement, you cannot also include the **rec=** I/O attribute.

270 ERROR `REC= not allowed with internal file`

If you designate that an I/O statement is to work on an internal file, you cannot also include the **rec=** I/O attribute in that statement. For instance, the following is incorrect:

```
character*10 my_file
real*4       data
      . . .
write (my_file, *, rec=10) data                  {Wrong!}
```

271 ERROR `Format specifier required with internal file`

When you designate that a **read** or **write** is to work on an internal file, you must include format specifications for the file. (However, you cannot designate list-directed I/O; see Error 269.) This means the following is incorrect:

```
character*10 internal_file
     . . .
read (internal_file, iostat=int, end=10) nums  {Wrong!}
```

Following is one way to correct the **read** statement:

```
read (internal_file, fmt=25, iostat=int, end=10) nums
```

272 ERROR *Token* `invalid unit specifier`

A unit specifier in an I/O statement must be an integer constant or variable. *Token* isn't. For example, the following is incorrect because 3.2 isn't an integer:

```
open(3.2, iostat=open_stat, err=100)
```

273 ERROR *Token* `invalid format specifier`

The format specifier in an I/O statement must be a statement label of a **format** statement, an asterisk to indicate list-directed formatting, or another valid specifier. (The listings for **format** and **print** in Chapter 4 detail the valid specifiers.) *Token* is not one of the valid choices. For example, the following produces this error because the format specifier has been omitted, and so the compiler thinks the variable **player_name** is the specifier:

```
print player_name, rank
```

Either of the following is correct:

```
print *, player_name, rank                 {1st way}
print 30, player_name, rank                {2nd way}
```

274 ERROR `Statement functions cannot be recursive`

Only subroutines and regular functions can be recursive in Domain FORTRAN; statement functions cannot. See Section 5.3 for more information on statement functions and and Section 5.8 for more information about recursion.

275 ERROR `Identifier invalid prior use of dummy argument`

You used *identifier* as a variable earlier in this subprogram, but now are indicating that it is a dummy argument. You can't do both. For example, the following is incorrect:

```
function y()
integer*4 my_ptr
pointer /my_ptr/ list
     . . .
entry way(list)                                  {Wrong!}
```

276 ERROR `Internal compiler error - EVALUATE`

The error is in the compiler, not in your code. Please contact your customer support representative.

277 ERROR `Internal compiler error - REF_OPND`

The error is in the compiler, not in your code. Please contact your customer support representative.

278 ERROR `Internal compiler error - encoded format overflow`

The error is in the compiler, not in your code. Please contact your customer support representative.

279 ERROR `Statement function dummy argument must be a variable` *arg_name*

The dummy arguments in a statement function must be variables; they cannot be subprogram names. For instance, this is invalid:

```
subroutine my_sub(dum, dumb)
integer*4 dum, dumb
a(my_sub) = dum*dumb                             {Wrong!}
```

280 ERROR `Internal compiler error - lbl_stk overflow`

The error is in the compiler, not in your code. Please contact your customer support representative.

281 ERROR `Too many include files`

There can be a maximum of 1200 **include** files in a Domain FORTRAN program. You exceeded that maximum.

284 ERROR `No matching block IF statement`

The compiler found some part of a block **if** statement—an **else if**, **else**, or **end if**—but can't find the beginning of the statement. A block **if** statement must begin with a clause of the form:

if (expression) **then**

. . .

This error also occurs if there is an error in the beginning of the block **if** that prevents the compiler from generating code for the statement.

285 ERROR `Missing END IF statement for block IF statement on line` *num*

Each block **if** statement must end with an **end if** statement, but the one that begins on line *num* has no **end if**. You might have nested some block **if**s and forgotten to explicitly end one of them.

286 ERROR `Improper nesting of DO loop starting on line` *num*

A **do** loop can contain any number of nested **do** loops, provided the range of each inner loop does not extend beyond the range of the outer loop(s). However, the **do** loop that begins on line *num* does extend past an outer loop. See the listing for **do** in Chapter 4 for examples of properly and improperly nested loops.

287 ERROR `Invalid use of assumed-length character entity`

No statement function dummy arguments may be of type **character*(*)**.

294 ERROR `Not enough constants`

There must be the same number of constants in a **data** statement as there are storage locations (variables and array elements) listed. You have fewer constants within the slashes (/ /) than you have variable names before them. You might have omitted a few values, or left out a comma between values so the compiler couldn't tell where one constant ended and another began. You also might have set up an implied **do** loop but not provided enough values to complete all trips through the loop.

For example, the following is incorrect because the implied **do** loop says to assign 20 values to the array **table**, but the repeat count within the slashes is only 10:

```
data ((table(i,j), i=1,4), j=1,5) /10*0/
```

296 ERROR `Equivalence extends common backwards:` *common_name*

It is an error to use an **equivalence** statement to extend the **common** block *common_name* beyond its first element. See the listing for **equivalence** in Chapter 4 for more information and an example of this error.

300 ERROR *Token* `cannot represent real or double precision constant`

Token exceeds the maximum number that Domain FORTRAN can represent for the data type to which you are assigning *token*. Chapter 3 lists the range of values each numeric data type can take.

You can get this error even if you are trying to make an assignment to an **integer** variable. For example:

```
i = 12345678901234567890
```

301 ERROR *Identifier* `illegal namelist groupname specification`

Your I/O statement indicates that you want to perform I/O on the namelist *identifier*. However, you haven't declared *identifier* as a namelist. You must declare a namelist before you can perform I/O on it.

303 ERROR `REC= not allowed with namelist I/O`

It is an error to use the **rec=** I/O attribute when performing input or output on a namelist. See the listing for **read** or **write** in Chapter 4 for more information.

304 ERROR `Format specifier not allowed with namelist I/O`

It is an error to specify a format when you perform I/O on a namelist. In essence, the namelist designator takes the place of the format specifier. This means the following is an error:

```
namelist /my_names/ i,j,k
read (4, fmt=10, nml=my_names)                    {Wrong!}
```

305 ERROR `I/O list not allowed with namelist I/O`

It is an error to include an I/O list in an I/O statement when you also specify a namelist. For example, the following is incorrect:

```
namelist /names/ first, middle, last
read (*, nml=names) first, middle, last           {Wrong!}
```

See the listings for **print**, **read**, and **write** for more information on I/O lists and on using **namelists**.

307 ERROR *Identifier* `name already used`

The name you gave this namelist already appears in an earlier declaration statement. It is an error to reuse the name. For example, you might have done something like the following:

```
integer*4 list, j, k
    . . .
namelist /list/ j, k                              {Wrong!}
```

308 ERROR *Token* {*hex_value*} `illegal character in program line`

You used an illegal character in your program. *Token*'s hexadecimal value appears in *hex_value*, so you can look up the illegal character in Appendix B.

309 ERROR or WARNING
`Routine exceeds maximum stack size`

Your program asks for more dynamic storage than this machine contains. Different machine types have differing amounts of dynamic storage. Probably your program includes an array that is too big for the node to handle.

310 FATAL `Too many source lines`

No single source file can have more than 32,767 lines in it. Your file exceeds that limit. Break the file into smaller files and recompile.

311 ERROR `No valid statements specified (null source)`

There are no valid Domain FORTRAN statements in the file you specified for compilation. In order for the compiler to parse the file, there must be at least one valid FORTRAN statement.

312 ERROR `Cannot specify filename and streams ID in open statement.`

You can indicate the file to be opened using *either* **filename** or **strid**, but you cannot use *both* in the same statement.

313 INFO `Variable is unnaturally aligned:` *identifier*

To ensure that the data in a **common** block is naturally aligned, you should arrange the objects in the block in descending order by size. In general, your program runs faster if your data is naturally aligned.

314 INFO `Reference to unaligned variable`

See message 313.

316 ERROR `Pointer variable cannot also be in based variable list`

You cannot include a pointer variable in a list of variables that are referenced by a pointer. In other words, you cannot declare a pointer to a pointer.

331 WARNING `-MP option ignored for target machine`

You cannot specify the **-mp** option for MC680x0-based workstations. These workstations do not have the multiprocessor capability for which the **-mp** option is appropriate. See subsection 6.5.22 for more information about this option.

332 ERROR `Multiple specification of a parallel clause detected`

Any parallel programming directive cannot contain more than one instance of the same clause (for example, the **LOCAL** clause). Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

333 ERROR `Unrecognized PARALLEL REGION directive`

The **PARALLEL REGION** directive contains a syntax error, such as a misspelled keyword or an identifier name mistaken for a keyword. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

334 ERROR `Logical expression expected`

The **IF** clause accompanying the **PARALLEL REGION** directive does not contain a valid logical expression. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

335 ERROR `ORDERED SECTION directive found outside the scope of a PARALLEL DO directive`

ORDERED SECTION directives must be lexically contained within the scope of a **PARALLEL DO** directive. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

336 ERROR `Name already seen in local list:` *identifier*

You have specified the same *identifier* name in the list accompanying the **LOCAL** clause. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

337 ERROR `Badly formed PARALLEL DO directive`

The **PARALLEL DO** directive contains a syntax error, such as a misspelled keyword. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

338 ERROR `Nested parallel regions are not allowed`

Parallel regions cannot be lexically contained within each other. You probably forgot to insert an **END PARALLEL REGION** directive for the previous **PARALLEL REGION** directive before beginning another parallel region. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

339 ERROR `END PARALLEL REGION directive encountered outside of a parallel region`

The compiler encountered an **END PARALLEL REGION** directive without a matching **PARALLEL REGION** directive. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

340 ERROR `Missing keyword to PARALLEL REGION clause`

You specified a clause for the **PARALLEL REGION** directive but forgot to include the keyword or specified some other word in its place. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

341 ERROR `Missing END PARALLEL REGION directive`

The compiler came to the end of the program unit and did not find an **END PARALLEL REGION** directive to match the last **PARALLEL REGION** directive. Every **PARALLEL REGION** directive must have a matching **END PARALLEL REGION** directive. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

342 ERROR `The ONE PROCESSOR SECTION directive must occur within a parallel region`

The **ONE PROCESSOR SECTION** directive must be lexically contained within a parallel region. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

344 ERROR `Extraneous directive clause encountered.`

Either you have specified the same clause twice or you have specified a clause for a directive that does not take that clause. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

345 WARNING `Directive implies existence of a parallel region.`

You have used a **PARALLEL SECTION** or **PARALLEL DO** directive that is not lexically inside a parallel region. You must first specify a **PARALLEL REGION** directive before specifying either of these directives. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

347 ERROR `Section name specified in WAIT clause not found.`

The **WAIT** clause must specify a name that is also specified in a **PARALLEL SECTION** directive. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

348 ERROR `Duplicate name:` *identifier*

You have given two **PARALLEL SECTION** directives with the same *identifier* name. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

349 ERROR `Compile time constant too large for target data class:` *identifier*

The compiler detected that the value of *identifier* exceeded the range of values for its data type.

350 ERROR `Missing DO statement - after PARALLEL DO directive`

The **PARALLEL DO** directive must appear just before a **DO** statement. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

352 ERROR `Incorrect nesting of parallel directives.`

You are attempting to use parallel programming directives to nest one section of code within another, but the nest is poorly formed, perhaps because of an overlap. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

353 ERROR `Nested parallel loops not supported.`

You cannot use **PARALLEL DO** directives to create nested parallel loops. The following example will draw this error message

```
C*HP-PARALLEL* PARALLEL DO
      DO 10 I = 1, N
              .
              .
              .
C*HP-PARALLEL* PARALLEL DO
                      DO 20 J = 1, M
                      .
                      .
                      .
```

Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

354 ERROR `Lastlocal list may not include an array element`

You have included the name of a subscripted variable in the list accompanying the **LAST LOCAL** clause. The list can include only scalar variables. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

355 ERROR `Variable is both shared and local:` *identifier*

Identifier name is specified in both the **SHARED** clause and **LOCAL** clause. It can appear in only one or the other. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

356 WARNING `Static do may not be blocked`

You have used the **BLOCKED** keyword with the **STATIC** clause, but it can be specified with the **DYNAMIC** clause only. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

357 ERROR `Dummy argument may not occur in a local list:` *identifier*

The list of variables specified with the **LOCAL** clause includes *identifier* name, which is a dummy argument in a subprogram. Dummy arguments cannot be specified in the **LOCAL** clause. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

358 ERROR `Variable may not occur in a local list:` *identifier*

Identifier name cannot be included in the list of variables specified with the **LOCAL** clause. For example, pointer-based variables cannot be included in the local list. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

360 ERROR `Variable may not occur in a shared list:` *identifier*

Identifier name cannot be included in the list of variables specified with the **LOCAL** clause. For example, variables declared by the **PCOMMON** statement cannot be included in the shared list. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

361 WARNING `Variable assumed shared:` *identifier*

Any variable in a parallel region that is not explicitly declared as local—that is, is not included in the list of variables specified with the **LOCAL** clause—is assumed to be shared. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

362 ERROR `Invalid sequence variable:` *identifier*

Sequence variables declared **APOST**, **ASET**, or **AWAIT** directive must be of **INTEGER*2** data type. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

363 WARNING `Variable in shared/local list has not been declared:` *identifier*

You have not initialized *identifier* outside the parallel region before specifying it with either the **LOCAL** or **SHARED** clause *and* with the **INITIALIZE** clause. As a result, *identifier* will contain a random value. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

364 WARNING `Cannot find /usr/apollo/bin/cf_tool.`

You have incorrectly installed the HPCF product or the installation process failed.

366 ERROR `Cannot branch across parallel regions:`

You cannot branch into or out of an area of code defined by the **PARALLEL REGION** and **END PARALLEL REGION** directives. Likewise, you cannot branch from one parallel region to another. Refer to the *HP Concurrent FORTRAN User's Guide* for more information about the parallel programming directives.

929 WARNING `Assignment eliminated; value never used:` *identifier*

You assigned a value to *identifier* but then never used that value. If *identifier*'s value has side effects, such as in a function call, the value still is computed, and the optimizer eliminates only the assignment to *identifier*. However, if there are no potential side effects, the optimizer also eliminates the value's computation.

In most cases, you can simply eliminate the value assignment to *identifier* to get rid of this warning. However, you might have to rewrite some code if a function that changes values of variables in **common** returns a value that is never used.

This message could indicate that your program is not doing what you intend it to do, and it should alert you to a possible bug in your program. For example, you should make sure that you are not depending on the assignment that the optimizer is eliminating to give a variable a value to be used during the next invocation of the routine.

Refer to subsection 6.5.26 for details about the effects of optimization.

9xx ERROR An error message with a number in the 900s indicates that the problem is in the compiler, not in your code. Please contact your customer support representative.

9.3 Run–Time Error Messages

Run–time error messages are notoriously difficult to decipher, mainly because any number of programming errors can cause them. This section attempts to describe the more common run–time errors, reasons why your program may have caused them, and some general approaches for getting rid of them.

Run–time errors fall into two broad categories:

- Addressing system errors, described in Section 9.3.3
- Floating–point errors, described in Section 9.3.4

Of the two, addressing errors are the more diverse and difficult to fix. The next two subsections describe some of the causes of addressing errors and suggest some ways to locate and fix them. Floating–point errors are described in the *Domain Floating–Point Guide*.

9.3.1 Causes of Run–Time Errors

Addressing errors most commonly occur when your program attempts to access a protected area of memory. There are several ways a program can do this:

- Attempting to write to an area of memory that is read–only, such as a library or your program's object code (access violation)
- Attempting to access an area of memory that has not been allocated for its use (reference to an illegal address)
- Overwriting part of the stack frame that contains information that a subroutine needs to return from a call (stack unwind error)
- Attempting to write to a guard segment—a small area of memory on either side of the stack (guard fault)

Figure 9–1 shows the main areas of memory that a program uses and the errors caused by invalid uses of these areas.

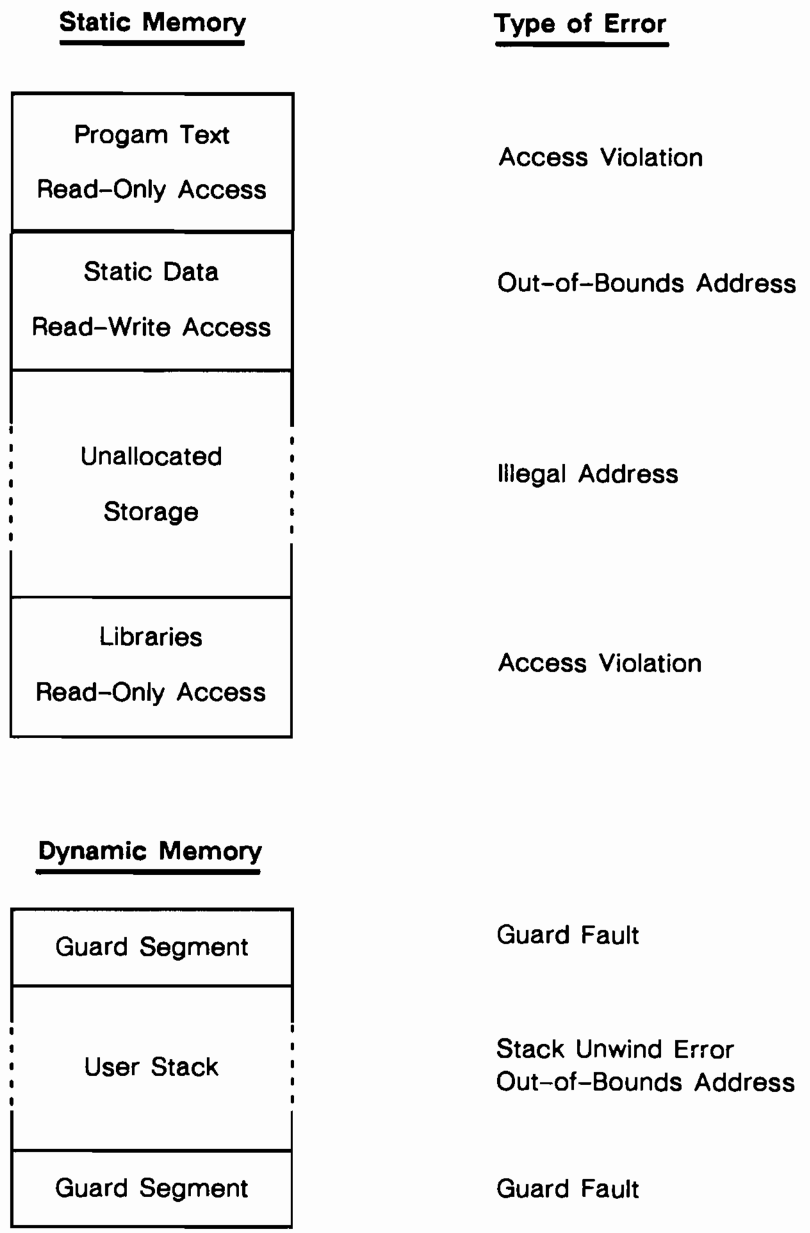

Figure 9-1. System Memory and Run-Time Errors

9.3.2 Debugging Run–Time Errors

If you get a run–time error after your program compiles with a warning message, the message may tell you what and where the problem is. However, if the program compiles with no warnings, you need to identify which lines of your program are causing the error before you can determine what the error is and how to fix it. Two Domain/OS tools can help you identify the lines causing the error:

- The traceback (**tb**) command
- The Domain Distributed Debugging Environment (Domain/DDE)

Getting a traceback is usually the best way to begin to find the cause of a run–time error. The **tb** command tells you what line of your source code caused the error. Two useful options to specify when invoking **tb** are

- **–f[ull]**, which requests information about the address that caused the error and the contents of the machine registers.
- **–l[ast]**, which requests traceback for the most recent aborted process. This option is especially useful when you are operating in a UNIX environment and invoking **tb** without options results in the message "No traceback information matched your specifications".

For example, suppose that the program you are writing compiles without errors but fails at run time with the following message:

```
Memory fault
```

The **tb –f –l** command provides a display like this:

```
$ tb -f -l
Process          25326 (parent 25274, group 25326)
Time             90/03/21.10:50(EST)
Program          /macon/my_progs/a.out
Status           00040004: reference to illegal address (OS/MST manager)
In routine       "$main" line 455
Called from      "PM_$CALL" line 176
Called from      "pgm_$load_run" line 891
Called from      "pgm_$invoke_uid_pn" line 1112
Proc2 Uid         495412A6.C000C986
Parent Process    25274 (495360D9.B000C986)
Process Group     25326 (495412A6.C000C986)
Fault Status      00040004: reference to illegal address (OS/MST manager)
Access Addr       00018000
IR                C001
Acc. Info         0121
User Fault PC     00008194
D0-D3:            00003FFC 00004000 031C75FF 00000000
D4-D7:            00000000 00000F08 00000015 00000004
```

```
A0-A3:            00017FF8 03200028 031D8C04 031D0094
A4-A7:            031C7DEC 00010000 031C7404 031C7400
Supervisor ECB    00000000
Supervisor SR     0000
Supervisor PC     00000000
```

This tells you that line 455 in your program took the error. If after examining the line you still don't understand why the error occurred, you can invoke Domain/DDE by typing **dde**. Domain/DDE enables you to step through your program as it executes and examine watch variables. For information about the debugger, refer to the *Domain Distributed Debugging Environment Reference*.

If your program fails with one of the following error messages

- access violation
- reference to illegal address
- odd address error
- stack unwind error

the problem may be an array-bounds error—that is, an error caused by the attempt to access array elements for which no memory has been allocated. Try the following approach to locating and debugging array-bounds errors:

1. Get a full traceback (**tb –f –l**) and find out where in your code things are going wrong.

2. Using the traceback information, verify that the number and type of all parameters in all called routines match.

3. Recompile with either **ftn –subchk** or **f77 –C**, and run again. This will cause a runtime fault to occur as soon as your program attempts to go out of the bounds of an array. If your program now fails with the "subchk violation" error message, get a traceback and find out where you are exceeding your array.

   Another source of array-bounds errors is using an **integer*2** as an index variable in a large array. By default the compiler uses an **integer*2** as an index variable. If you have an assumed size array that accesses beyond the 32,767th element, you may need to use the **–indexl** compiler option to force the use of an **integer*4** as the index variable. To override the default, compile with **ftn –indexl** or **f77 –W0,–indexl**.

9.3.3 Addressing Errors

This section describes the more common addressing and memory errors and offers concrete suggestions about their causes. As noted, some of the error messages—"Apollo-specific fault", "Bus error", "Memory fault", and "Segmentation fault"—describe a category of errors, for which you can get more specific information by invoking the traceback (**tb**) command. If you try to invoke traceback but get the message "No traceback available", your program has probably destroyed the user stack so that traceback is unable to trace the series of calls leading from the fault to the main program.

`access violation (OS/fault handler)`

Your program attempts to access address space that is allocated in the process but is not accessible to your program—for example, global read-only address space. For information about how to locate this kind of error, refer to Section 9.3.2.

`Apollo-specific fault (UNIX/signal)`

If you use the **tb -full** command, this error will resolve to a more specific error, such as "reference to out-of-bounds address". For information, refer to the discussion of that error later in this subsection.

`Bus error`

If you use the **tb** command, this BSD and SysV environment run-time error will resolve to a more specific error, such as "odd address error". For information, refer to the discussion of that error later in this subsection.

`disk full or not enough storage for static storage (process manager/ loader)`

This error occurs when an attempt is made at load time to allocate all static storage. It is most likely to occur if you have large arrays and a small amount of disk space. FORTRAN programs try to load all static storage at run time. If they fail, the operating system issues a "disk full" error message. Binding your program with the **-sparse_vm** option should prevent this error from occurring. Here is the command line:

/com/bind *object_file_list* -sparse_vm -b *bound_object*

`guard fault (OS/MST manager)`

Your program attempts to access one of the guard segments surrounding the program stack. (A guard segment is an area that sits on either side of sections of memory; see Figure 9-1.) This error is usually caused by exceeding the stack size. One solution is to increase the stack size with either of the following command lines:

% /bin/ld -A stacksize *hex_num object_files*

$ /com/bind *object_files* **-stacksize** *decimal_number*

For additional information about **ld** and the **stacksize** option, refer to the discussion of **ld** in the *SysV Command Reference* or the *BSD Command Reference*. The **/com/bind** command is briefly described in Chapter 6, but for a full discussion of its options, refer to the *Domain/OS Programming Environment Reference*.

Other possible causes of this kind of error include exceeding the declared size of an array (refer to Section 9.3.2), misuse of the **pointer** statement, and an infinitely recursive call.

`Memory fault`

If you use the **tb** command, this SysV environment run-time error will probably resolve to a more specific error, such as "access violation", "guard fault", "reference to illegal address", or "reference to out-of-bounds address". For information, refer to the discussion of these errors in this subsection.

`odd address error (OS/fault handler)`

Your program attempts to access an object address that is not divisible by two. This error is usually caused by assigning the address of a single-byte object (such as an element in a character array) to an integer. Check that any arguments your program may pass to a subroutine agree with the dummy arguments.

This error occurs only on Series 10000 workstations and on older, MC68010 machines.

`reference to illegal address (OS/MST manager)`

Your program attempts to access address space that isn't allocated; nothing is mapped there. Or it may be using a bad address, such as zero or a negative number.

This problem may be caused by misuse of the **pointer** statement or by a bad array reference that attempts to access a protected area of memory. Refer to Section 9.3.2.

`reference to out-of-bounds address (OS/MST manager)`

Your program references address space that is allocated but lies beyond the end of a mapped object. Walking off the end of an array is one cause of this error. Refer to Section 9.3.2.

`Segmentation fault`

If you do a traceback, this BSD environment run-time error will probably resolve to "access violation", "guard fault", or "reference to illegal address". For information, refer to the discussion of these errors in this subsection.

`supplied buffer too small (library/MBX manager)`

Your program attempts to access more memory than is allocated.

This error frequently occurs in FORTRAN programs that do not use the **recl=** I/O attribute in the **open** statement to specify an adequate record length, but instead take the default (256 bytes). To get rid of the error, make sure that your **open** statement includes a **recl=** I/O attribute and that you specify an adequate record length. For more information about the **recl=** I/O attribute, refer to the listing for "I/O attributes" in Chapter 4.

`unable to unwind stack because of invalid stack frame (OS/MST manager)`

The runtime stack has been destroyed and is unuseable.

This problem is commonly caused by attempting to access array elements for which no memory has been allocated (refer to Section 9.3.2) or by an infinitely recursive program.

9.3.4 Floating-Point Errors

This section is not an exhaustive survey of floating-point errors. It offers brief explanations of some common but puzzling errors that will cause your program to abort at run time. For a complete discussion of floating-point calculations and possible errors, refer to the *Domain Floating-Point Guide*.

`floating point operand error (OS/fault handler) {$ stcode 120025}`

One of the operands in an expression is invalid; that is, it does not represent a real number or is otherwise not an acceptable value for that particular operation. The error can result from a programmer's bug or bad data—for example, the use of a negative number as the argument to a logarithm function. For a full list of possible causes, refer to the *Domain Floating-Point Guide*.

`floating point branch/set on unordered condition (OS/fault handler) {$ stcode 12002e}`

This error results from a Quiet Not-A-Number (QNAN). The QNAN is a type of NAN, a range of floating-point values that is reserved by the IEE-754 floating-point standard to represent the results of operations that have no mathematical interpretation.

The bit pattern for the QNAN has all exponent bits set to 1s and the most significant bit of the fraction set to 1. Such a bit pattern should never occur as the result of an arithmetic operation on legitimate numbers. It can result from uninitialized floating-point variables, type transfers using the **equivalence** statement, or simply bad floating-point arithmetic.

```
floating point signalling not-a-number (OS/fault handler)
                {$ stcode 12002f}
```

This error results from a Signalling Not-A-Number (SNAN). The SNAN is a type of NAN, a range of floating-point values that is reserved by the IEE-754 floating point standard to represent the results of operations that have no mathematical interpretation.

An SNAN's most significant bit of the fraction equals 0 and at least one mantissa bit is set to 1. An SNAN is never created as the result of an operation but can result from addressing errors, bad data, poorly constructed **common** blocks, and mismatched parameters. Any arithmetic operation on an SNAN will give this error.

Appendix A

Domain FORTRAN Keywords

This appendix lists the keywords in Domain FORTRAN.

assign	do	implicit	pointer
atomic	do while	implicit none	print
backspace	double complex	include	program
block data	double precision	inquire	read
byte	else	integer	real
call	encode	integer*2	real*4
character	end	integer*4	real*8
close	end do	intrinsic	return
common	endfile	logical	rewind
complex	end if	logical*1	save
complex*8	entry	logical*2	stop
complex*16	equivalence	logical*4	subroutine
continue	external	namelist	then
data	format	open	to
decode	function	options	write
dimension	go to	parameter	
discard	if	pause	

Figure A-1. Domain FORTRAN Keywords

Appendix B

ISO Latin-1 Table

Domain FORTRAN uses the ISO DIS 8859/1 character set, commonly known as Latin-1, for character data representation. The Latin-1 set also includes all ASCII characters in their standard positions. Table B-1 shows the decimal, octal, and hexadecimal values for all ASCII characters.

You can use Latin-1 characters in comments or character strings, but you are limited to using ASCII letters A-Z and a-z (decimal positions 65-90 and 97-122, respectively), digits, underscores (_), and dollar signs ($) in identifiers. This adheres to existing FORTRAN standards.

NOTE: The characters with decimal numbers 128 through 131 are missing from the table. These values are reserved for future standardization and are not available to programmers.

Table B-1. ISO Latin-1 Codes

oct	dec	hex	character		oct	dec	hex	character
0	0	0	NUL	^@	40	32	20	space
1	1	1	SOH	^A	41	33	21	!
2	2	2	STX	^B	42	34	22	"
3	3	3	ETX	^C	43	35	23	#
4	4	4	EOT	^D	44	36	24	$
5	5	5	ENQ	^E	45	37	25	%
6	6	6	ACK	^F	46	38	26	&
7	7	7	BEL	^G	47	39	27	'
10	8	8	BS	^H	50	40	28	(
11	9	9	TAB	^I	51	41	29	)
12	10	A	LF	^J	52	42	2A	*
13	11	B	VT	^K	53	43	2B	+
14	12	C	FF	^L	54	44	2C	,
15	13	D	CR	^M	55	45	2D	-
16	14	E	SO	^N	56	46	2E	.
17	15	F	SI	^O	57	47	2F	/
20	16	10	DLE	^P	60	48	30	0
21	17	11	DC1	^Q	61	49	31	1
22	18	12	DC2	^R	62	50	32	2
23	19	13	DC3	^S	63	51	33	3
24	20	14	DC4	^T	64	52	34	4
25	21	15	NAK	^U	65	53	35	5
26	22	16	SYN	^V	66	54	36	6
27	23	17	ETB	^W	67	55	37	7
30	24	18	CAN	^X	70	56	38	8
31	25	19	EM	^Y	71	57	39	9
32	26	1A	SUB	^Z	72	58	3A	:
33	27	1B	ESC	^[	73	59	3B	;
34	28	1C	FS	^|	74	60	3C	<
35	29	1D	GS	^]	75	61	3D	=
36	30	1E	RS	^^	76	62	3E	>
37	31	1F	US	^_	77	63	3F	?

(Continued)

Table B-1. ISO Latin-1 Codes (continued)

oct	dec	hex	character	oct	dec	hex	character
100	64	40	@	140	96	60	’
101	65	41	A	141	97	61	a
102	66	42	B	142	98	62	b
103	67	43	C	143	99	63	c
104	68	44	D	144	100	64	d
105	69	45	E	145	101	65	e
106	70	46	F	146	102	66	f
107	71	47	G	147	103	67	g
110	72	48	H	150	104	68	h
111	73	49	I	151	105	69	i
112	74	4A	J	152	106	6A	j
113	75	4B	K	153	107	6B	k
114	76	4C	L	154	108	6C	l
115	77	4D	M	155	109	6D	m
116	78	4E	N	156	110	6E	n
117	79	4F	O	157	111	6F	o
120	80	50	P	160	112	70	p
121	81	51	Q	161	113	71	q
122	82	52	R	162	114	72	r
123	83	53	S	163	115	73	s
124	84	54	T	164	116	74	t
125	85	55	U	165	117	75	u
126	86	56	V	166	118	76	v
127	87	57	W	167	119	77	w
130	88	58	X	170	120	78	x
131	89	59	Y	171	121	79	y
132	90	5A	Z	172	122	7A	z
133	91	5B	[	173	123	7B	{
134	92	5C	\	174	124	7C	|
135	93	5D	]	175	125	7D	}
136	94	5E	^	176	126	7E	~
137	95	5F	_	177	127	7F	del

(Continued)

Table B-1. ISO Latin-1 Codes (continued)

oct	dec	hex	character	oct	dec	hex	character
204	132	84	IND	247	167	A7	§
205	133	85	NEL	250	168	A8	¨
206	134	86	SSA	251	169	A9	©
207	135	87	ESA	252	170	AA	ª
210	136	88	HTS	253	171	AB	«
211	137	89	HTJ	254	172	AC	¬
212	138	8A	VTS	255	173	AD	SHY
213	139	8B	PLD	256	174	AE	®
214	140	8C	PLU	257	175	AF	¯
215	141	8D	RI	260	176	B0	°
216	142	8E	SS2	261	177	B1	±
217	143	8F	SS3	262	178	B2	²
220	144	90	DCS	263	179	B3	³
221	145	91	PU1	264	180	B4	´
222	146	92	PU2	265	181	B5	µ
223	147	93	STS	266	182	B6	¶
224	148	94	CCH	267	183	B7	·
225	149	95	MW	270	184	B8	¸
226	150	96	SPA	271	185	B9	¹
227	151	97	EPA	272	186	BA	º
233	155	9B	CSI	273	187	BB	»
234	156	9C	ST	274	188	BC	¼
235	157	9D	OSC	275	189	BD	½
236	158	9E	PM	276	190	BE	¾
237	159	9F	APC	277	191	BF	¿
240	160	A0	NBSP	300	192	C0	À
241	161	A1	¡	301	193	C1	Á
242	162	A2	¢	302	194	C2	Â
243	163	A3	£	303	195	C3	Ã
244	164	A4	¤	304	196	C4	Ä
245	165	A5	¥	305	197	C5	Å
246	166	A6	¦	306	198	C6	Æ

(Continued)

Table B-1. ISO Latin-1 Codes (continued)

oct	dec	hex	character	oct	dec	hex	character
307	199	C7	Ç	347	231	E7	ç
310	200	C8	È	350	232	E8	è
311	201	C9	É	351	233	E9	é
312	202	CA	Ê	352	234	EA	ê
313	203	CB	Ë	353	235	EB	ë
314	204	CC	Ì	354	236	EC	ì
315	205	CD	Í	355	237	ED	í
316	206	CE	Î	356	238	EE	î
317	207	CF	Ï	357	239	EF	ï
320	208	D0	Đ	360	240	F0	ð
321	209	D1	Ñ	361	241	F1	ñ
322	210	D2	Ò	362	242	F2	ò
323	211	D3	Ó	363	243	F3	ó
324	212	D4	Ô	364	244	F4	ô
325	213	D5	Õ	365	245	F5	õ
326	214	D6	Ö	366	246	F6	ö
327	215	D7	×	367	247	F7	÷
330	216	D8	Ø	370	248	F8	ø
331	217	D9	Ù	371	249	F9	ù
332	218	DA	Ú	372	250	FA	ú
333	219	DB	Û	373	251	FB	û
334	220	DC	Ü	374	252	FC	ü
335	221	DD	Ý	375	253	FD	ý
336	222	DE	Þ	376	254	FE	þ
337	223	DF	ß	377	255	FF	ÿ
340	224	E0	à				
341	225	E1	á				
342	226	E2	â				
343	227	E3	ã				
344	228	E4	ä				
345	229	E5	å				
346	230	E6	æ				

Appendix C

Summary of Intrinsic Functions

This appendix contains a table that summarizes the intrinsic, or predefined, functions available in Domain FORTRAN. Each function category contains a definition, the number of arguments required, available generic and specific names, input argument types, and function result types. The table lists type conversion functions first, followed by arithmetic, trignometric, lexical, bitwise, and address functions.

When the table lists a generic function name, you can use that generic name for input arguments with any of the listed data types. You can only use the specific function names, however, with arguments having the individual corresponding data types. For example, consider the following table entry for **int**:

Description	Number of Arguments	Generic Name	Specific Name	Type of Arguments	Type of Result
Convert to integer.	1	int	–	integer	integer [1]
			int	real	integer [1]
			ifix	real	integer [1]
			idint	double	integer [1]
			–	complex	integer [1]
			–	complex*16	integer [1]

You can use the generic name **int** with arguments having any of the listed data types: **integer**, **real**, **double precision**, **complex**, or **complex*16**. But if, for example, you use the specific name **ifix**, the argument must be of type **real**.

Notice that in some cases there is no specific name attached to an individual data type. The dash (—) indicates those cases. For example, there's no specific name for **complex** arguments. In such a case, use the generic name.

NOTE: Arguments designated type **integer** can also be of type **byte**.

Table C-1. Domain FORTRAN Intrinsic Functions

Description	Number of Arguments	Generic Name	Specific Name	Type of Arguments	Type of Result
Convert to integer.	1	int	— int ifix idint	integer real real double complex complex*16	integer[1] integer[1] integer[1] integer[1] integer[1] integer[1]
Convert to real.	1	real	real float — sngl — —	integer integer real double complex complex*16	real real real real real real
Convert to double-precision.	1	dble dfloat[2]	— — — — —	integer real double complex complex*16	double double double double double
Convert to complex.	1 or 2	cmplx	— — — — —	integer real double complex complex*16	complex complex complex complex complex
Convert to double complex. (same as complex*16)	1 or 2	dcmplx	— — — — —	integer real double complex complex*16	complex*16 complex*16 complex*16 complex*16 complex*16
Convert to integer representation.	1	—	ichar	character	integer[1]
Convert to character.	1	—	char	integer	character
Convert to integer*2.	1	—	int2 int2 int2 int2 int2	integer real double complex complex*16	integer*2 integer*2 integer*2 integer*2 integer*2

[1] Result type is **integer*2** if you compile with **–i*2**; **integer*4** otherwise.
[2] A synonym.

(Continued)

Table C-1. Domain FORTRAN Intrinsic Functions (Cont.)

Description	Number of Arguments	Generic Name	Specific Name	Type of Arguments	Type of Result
Convert to integer*4.	1	—	int4 int4 int4 int4 int4	integer real double complex complex*16	integer*4 integer*4 integer*4 integer*4 integer*4
Truncate fractional part of argument. Example: aint(1.6) =1.0	1	aint	aint dint	real double	real double
Round argument to nearest whole number. Example: anint(1.6)=2.0	1	anint	anint dnint	real double	real double
Round argument to nearest integer. Example: nint(1.6)=2	1	nint	nint idnint	real double	integer [1] integer [1]
Find absolute value.	1	abs	iabs abs dabs cabs cdabs	integer real double complex complex*16	integer [2] real double real double
Find remainder of arg1 divided by arg2.	2	mod	mod amod dmod	integer real double	integer [2] real double
The result is the absolute value of arg1 with the sign of arg2.	2	sign	isign sign dsign	integer real double	integer*4 real double
Positive difference: return arg1 – arg2 if arg1>arg2 or return 0 if arg1<arg2.	2	dim	idim dim ddim	integer real double	integer*4 real double
Double-precision product: return arg1*arg2.	2	—	dprod	real	double
Return maximum of listed arguments.	2 or more	max	max0 amax1 dmax1 amax0 max1	integer real double integer real	integer [2] real double real integer [1]

[1] Result type is **integer*2** if you compile with **–i*2**; **integer*4** otherwise.
[2] Result type depends on argument types. If any argument is **integer*4**, the result is **integer*4**; otherwise, result type is **integer*2**.

(Continued)

Table C-1. Domain FORTRAN Intrinsic Functions (Cont.)

Description	Number of Arguments	Generic Name	Specific Name	Type of Arguments	Type of Result
Return minimum of arguments listed.	2 or more	min	min0 amin1 dmin1 amin0 min1	integer real double integer real	integer[2] real double real integer[1]
Find length of character argument.	1	—	len	character	integer[1]
Return an integer indicating the starting position of string arg2 in arg1. If arg2 does not occur in arg1, return 0.	2	—	index	character	integer[1]
Return the imaginary part of a complex number.	1 1	— —	aimag dimag	complex complex*16	real double
Return the real part of a complex number.	1	—	dreal	complex*16	double
Return conjugate of a complex or a double complex number; that is, if arg =(ar, ai), return (ar, −ai).	1	conjg	conjg dconjg	complex complex*16	complex complex*16
Square root: return arg$^{1\;2}$	1	sqrt	sqrt dsqrt csqrt cdsqrt	real double complex complex*16	real double complex complex*16
Return e raised to the power of arg.	1	exp	exp dexp cexp cdexp	real double complex complex*16	real double complex complex*16
Find the natural logarithm of arg; that is, log (arg).	1	log	alog dlog clog cdlog	real double complex complex*16	real double complex complex*16
Find the common logarithm of arg; that is, log10 (arg).	1	log10	alog10 dlog10	real double	real double

[1] Result type is **integer*2** if you compile with **−i*2**; **integer*4** otherwise.

[2] Result type depends on argument types. If any argument is **integer*4**, the result is **integer*4**; otherwise, result type is **integer*2**.

(Continued)

Table C-1. Domain FORTRAN Intrinsic Functions (Cont.)

Description	Number of Arguments	Generic Name	Specific Name	Type of Arguments	Type of Result
Sine: compute the sine of arg.	1	sin	sin dsin csin cdsin	real double complex complex*16	real double complex complex*16
Cosine: compute the cosine of arg.	1	cos	cos dcos ccos cdcos	real double complex complex*16	real double complex complex*16
Tangent: compute the tangent of arg.	1	tan	tan dtan	real double	real double
Arcsine: compute the arcsine of arg.	1	asin	asin dasin	real double	real double
Arccosine: compute the arccosine of arg.	1	acos	acos dacos	real double	real double
Arctangent: compute the arctangent of arg.	1	atan	atan datan	real double	real double
Divide arg1 by arg2 and compute the arctangent of the result.	2	atan2	atan2 datan2	real double	real double
Hyperbolic sine: compute the hyperpolic sine of arg.	1	sinh	sinh dsinh	real double	real double
Hyperbolic cosine: compute the hyperbolic cosine of arg.	1	cosh	cosh dcosh	real double	real double
Hyperbolic tangent: compute the hyperbolic tangent of arg.	1	tanh	tanh dtanh	real double	real double
Return .true. if arg1 is lexically greater than or equal to arg2; otherwise return .false	2	—	lge	character	logical

(Continued)

Table C-1. Domain FORTRAN Intrinsic Functions (Cont.)

Description	Number of Arguments	Generic Name	Specific Name	Type of Arguments	Type of Result
Return .true. if arg1 is lexically greater than arg2; otherwise return .false.	2	—	lgt	character	logical
Return .true. if arg1 is lexically less than or equal to arg2; otherwise return .false.	2	—	lle	character	logical
Return .true. if arg1 is lexically less than arg1; otherwise return .false.	2	—	llt	character	logical
Bitwise AND: perform a logical AND on corresponding bits.	2	—	and iand[4]	integer	integer[2]
Bitwise OR: perform an inclusive (logical) OR on corresponding bits.	2	—	or ior[4]	integer	integer[2]
Bitwise exclusive OR: perform an exclusive OR on corresponding bits.	2	—	xor ixor[4]	integer	integer[2]
Bitwise negation.	1	—	not inot[4]	integer	integer[2]
Right logical shift: shift arg1 to the right arg2 bits.	2	—	rshft rshift[4]	integer	integer[3]
Left logical shift: shift arg1 to the left arg2 bits.	2	—	lshft lshift[4]	integer	integer[3]
Return the address of arg.	1	—	iaddr[5]	any type (including arrays and subprogram names)	integer*4

[1] Result type is **integer*2** if you compile with **-i*2**; **integer*4** otherwise.
[2] Result type depends on argument types. If any argument is **integer*4**, the result is **integer*4**; otherwise, result type is **integer*2**.
[3] Result type is type of first argument.
[4] A synonym.
[5] The **iaddr** function cannot be used in a statement function statement.

Appendix D

Optimizing Floating–Point Performance on MC68040–Based Domain Workstations

This appendix describes how to obtain the best floating–point performance on the new Domain MC68040–based workstations, such as the HP Apollo 9000 Series 400 Model 425t or Model 433s.

NOTE: The performance improvements described in this appendix are estimates for typical floating–point applications based on standard hardware configurations and standard software configurations. The performance improvements on your system, if any, will probably vary from the improvements described here.

The Motorola MC68040 microprocessor chip features floating–point performance almost an order of magnitude greater than that of its predecessor, the 68030/68882. Floating–point–intensive application binaries that currently run on Domain platforms will experience an immediate and dramatic performance increase when run on the new Domain 68040–based platforms. The increase is usually in the range of two to six times over the performance of the DN4500 Personal Workstation.

Floating–point arithmetic on 68040–based Domain platforms is nearly identical to floating–point arithmetic on 68020/68881–based and 68030/68882–based platforms, with only minor differences. Also, how you compile an application affects floating–point performance on the two platforms. In the following sections, we describe these functional and performance differences and tell you how to maximize floating–point performance on the 68040–based Domain workstations.

We discuss the following topics:

- Instruction emulation
- How to determine if an application relies heavily on instruction emulation
- How instruction emulation affects performance
- What steps to take for your application
- What it means if you get different results on the 68040 and the 68020/68030

D.1 Instruction Emulation

The 68040 includes an integrated floating-point unit that directly supports only a subset of the 68881/68882 architecture. Floating-point functionality that is not directly supported in hardware is provided through system traps; these system traps invoke a kernel routine that emulates the missing functionality. The emulation routine supports some instructions in the 68881/68882 instruction set and some data types.

Because software emulation is inherently slower than direct hardware execution, emulated instructions execute more slowly than hardware instructions. To maximize floating-point performance, either compile your application so that it has no emulated instructions, or determine that it does not have enough of them to degrade the performance of the code. We describe how to do this in the following sections.

D.2 How to Determine If an Application Relies Heavily on Instruction Emulation

The only applications with emulated instructions are those that were compiled with the **-cpu 3000** option. (We now call this option **-cpu mathchip**. For information about the **-cpu** option, see subsection 6.5.8.) The emulated instructions correspond to the following arithmetic intrinsic functions:

aint and **dint**	**cos** and **dcos**
alog and **dlog**	**exp** and **dexp**
alog10 and **dlog10**	**sin** and **dsin**
atan and **datan**	**tan** and **dtan**

If an application does not use any of these intrinsics, it will run nearly optimally on both the 68040 and the 680x0/6888x when compiled with **-cpu mathchip**.

Applications that are compiled with **-cpu any** (the old default -cpu argument) do not contain any emulated instructions. Compiling with this option therefore imposes a severe performance penalty, usually a greater penalty than that caused by instruction emulation. You should use **-cpu any** only if your code must run on *all* existing Domain 680x0-based workstations.

Intrinsic functions other than those listed above are always performed by run-time libraries that use only hardware-executed floating-point instructions. For example, if your FORTRAN program calls the **sinh** intrinsic, the compiler never generates the **fsinh** instruction; instead, it generates a call to the **ftn_$dsinh** routine.

D.3 How Instruction Emulation Affects Performance

As a rule of thumb, the 68040 can emulate an instruction at least as fast as a 68882 running at the equivalent clock frequency would execute it directly. What we call a performance penalty on 68040-based systems is actually unrealized performance potential, not performance degradation. Unchanged and un-recompiled applications will almost always realize some performance increase on the 68040 above what 68030-based and 68020-based systems delivered.

We cannot predict what kinds of performance increases you can obtain by recompiling your application for the 68040 unless we know the details of your application. We can offer some general guidelines, however. The following subsections describe the performance ratio for an application on a 68040-based system when you recompile with various **-cpu** arguments.

D.3.1 Changing from -cpu 3000 (-cpu mathchip) to -cpu mathlib or -cpu mathlib_sr10

If your application makes intensive use of the intrinsic functions listed in Section D.2 and is currently compiled with **-cpu 3000**, the performance boost from recompiling with **-cpu mathlib** or **-cpu mathlib_sr10** will probably be between one and three times, with most applications improving about 1.5 times. This boost results from removing emulated instructions from your code.

D.3.2 Changing from -cpu any to -cpu mathlib or -cpu mathlib_sr10

If your application is currently compiled with **-cpu any**, regardless of the intrinsics used, the performance boost from recompiling with **-cpu mathlib** or **-cpu mathlib_sr10** will probably be approximately two times, with some applications improving up to four times. This boost is due to the superior performance of inline floating-point instructions.

D.3.3 Changing from -cpu any to -cpu mathchip (-cpu 3000)

If your application makes intensive use of the intrinsic functions listed in Section D.2 and is currently compiled with **-cpu any**, the performance boost from recompiling with **-cpu mathchip** will probably be between 0.7 times and four times, with most applications improving about 1.3 times. The use of inline floating-point instructions improves performance, but inline emulated instructions degrade performance. We derive this estimate by combining the previous figures.

> NOTE: In the worst case, performance may actually degrade when you recompile an application from **-cpu any** to **-cpu mathchip**. However, the same recompilation may significantly improve performance on 68020-based and 68030-based systems.

D.4 What Steps to Take for Your Application

You have two separate decisions to make:

- Whether to recompile
- If you recompile, which **-cpu** argument to use

D.4.1 Should You Recompile?

When you decide whether to recompile for the 68040, you should consider not only the performance to be gained on the 68040, but also the effect the recompilation will have on 68020-based and 68030-based systems. Consider the proportion of pre-68040 and 68040 systems your application runs on today, and what you expect in the future. Targeting the pre-68040 systems may yield better performance for the customer today, but recompiling for the 68040 now may well yield big dividends as the percentage of 68040-based installed Domain workstations increases.

If you compiled your application with **-cpu any**, then you are probably incurring a severe performance penalty on all 68020-based, 68030-based, and 68040-based systems. Recompile unless you have to support older Apollo architectures (for example, the DN460 workstation or the PEB).

If you compiled your application with **-cpu 3000** or one of its equivalents, recompile if you think your application performs much instruction emulation on the 68040.

D.4.2 If You Recompile, Which -cpu Argument Should You Use?

If your application needs to run optimally only on 68040-based systems, compile with **-cpu mathlib**.

If your application needs to run optimally on 68020-based and 68030-based systems, but you don't care about its performance on the 68040, compile with **-cpu mathchip**. If you previously compiled your application with **-cpu 3000** (equivalent to **-cpu mathchip**), you do not need to recompile.

If your application needs to run well on 68020-based, 68030-based, and 68040-based systems, and you think it does not perform much instruction emulation on the 68040, you may compile with either **-cpu mathchip**, **-cpu mathlib_sr10**, or **-cpu mathlib**. If you think your application performs much instruction emulation on the 68040, recompile with **-cpu mathlib_sr10** or **-cpu mathlib**. Use **-cpu mathlib_sr10** if your code must run on 68020-based and 68030-based systems with a Domain/OS release earlier than SR10.3; otherwise, use **-cpu mathlib**.

Figure D-1 shows how to decide which argument is most suited to your application.

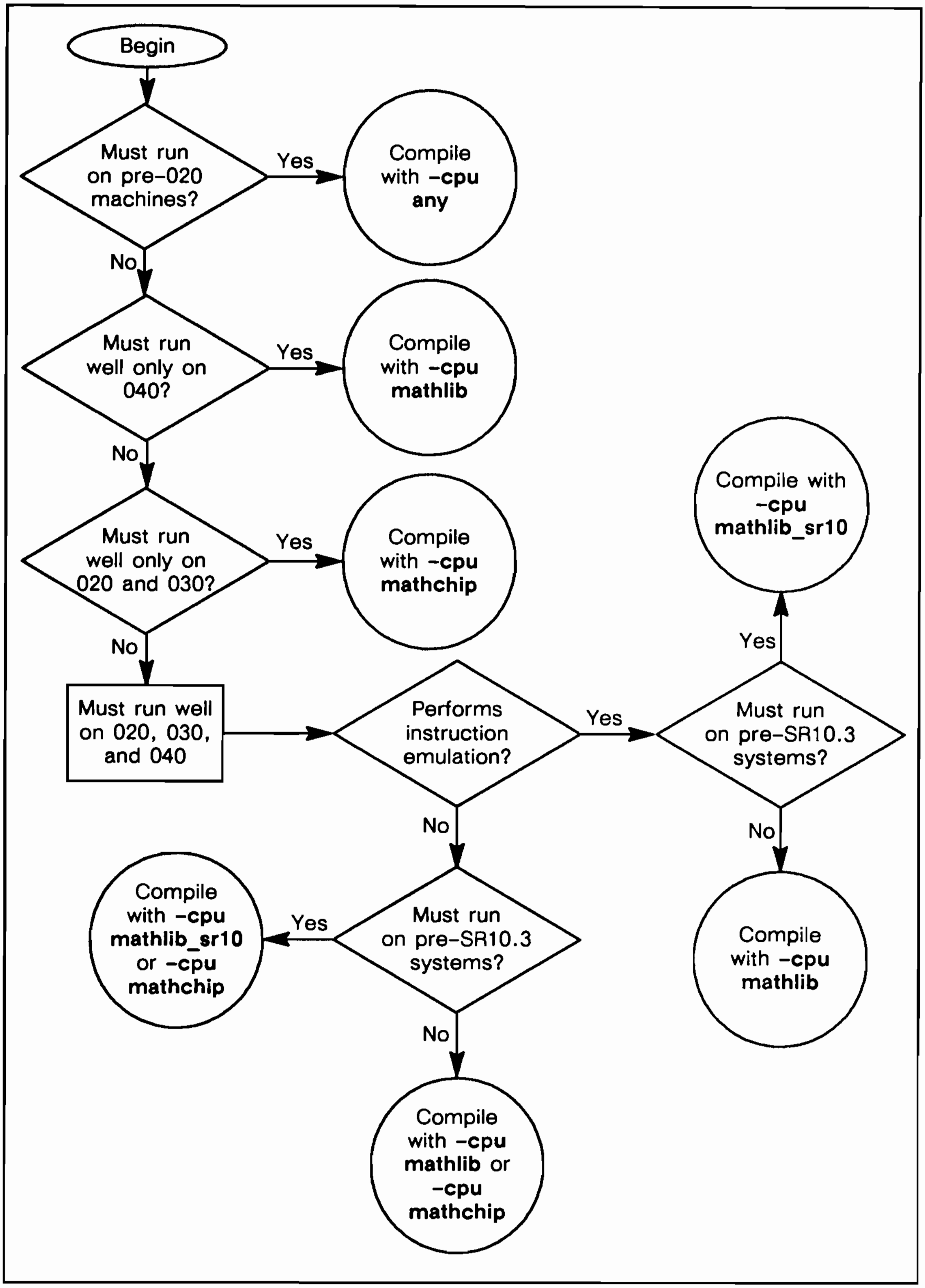

Figure D-1. Which **-cpu** *Argument Is Best for Your Application?*

D.5 If You Get Different Results on the 68040 and the 68020/68030

When you run a large floating-point application on a 68040-based Domain workstation, the results may differ slightly from those on 68020-based and 68030-based platforms. The differences are caused by the algorithms used to approximate trigonometric and transcendental math functions. Having two different sets of results does not mean that one is correct and the other incorrect, because floating-point intrinsic functions are inherently approximations. One math function can give two different results on two different platforms, yet both results can be acceptably precise approximations of the true result and are therefore both "correct."

Two different platforms can both comply with the IEEE-754 standard for floating-point arithmetic, yet applications executed on these two platforms may not behave identically. Because the IEEE standard does not cover many common math functions, each new implementation may yield slightly different behavior.

Inevitably, a few applications will yield extremely different results on the 68040 or may malfunction with various floating-point exceptions, such as overflow or divide-by-zero. Experience shows that these cases are almost always caused by applications that use some platform-specific feature; the application fails to run properly when executed on another platform that does not support that feature.

For example, the extended-precision capability of the 6888x-based platforms enables intermediate results in floating-point registers to exceed the maximum magnitude of double-precision. Thus a variable declared as double-precision can, during an application's execution, assume much larger values than on a machine that does not support extended-precision registers. Applications that use this feature will probably fail on the 68040, even though the 68040 supports extended-precision registers. The reason is that the 68040 relies on run-time libraries; therefore, values in floating-point registers are stored to memory in single-precision or double-precision format much more often than on the 6888x.

The Series 10000 workstations and the Domain Floating-Point Accelerator (FPA) do not support extended-precision registers. If your application runs correctly on either of these platforms, it will probably run correctly on the 68040.

For more information about floating-point results on different Domain platforms, see the *Domain Floating-Point Guide*.

Index

Symbols are listed at the beginning of the index.

Symbols

A

B

D

F

G

H

I

M

N

O

Q

R

T

U

V

W

X

Z

Reader's Response

Please take a few minutes to give us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain FORTRAN Language Reference*
Order No.: 000530–A01

User Profile

Your Name ______________________ Title ______________________

Company ______________________

Address ______________________

Telephone number ( ) ______________ Date ______________

When you use the HP/Apollo system, what **job(s)** do you perform?

- ☐ Programming
- ☐ Application End User
- ☐ Hardware Engineering
- ☐ System Administration
- ☐ Other (describe) ______________________

Characterize your level of **experience** in using the HP/Apollo system:

- ☐ Experienced user (2+ yrs.)
- ☐ New user (6 mos. or less)
- ☐ Moderately experienced user (6 mos.–2 yrs.)

What programming **languages** do you use with the HP/Apollo system?

____

Distribution

How do you know what manuals are **available** to support the products you're using or want to use?

____

What is a major concern for you in **ordering** books?

____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: ______________________

____

____

____

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

Apollo Systems Division
A Subsidiary of Hewlett-Packard Company
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold

Reader's Response

Please take a few minutes to give us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain FORTRAN Language Reference*
Order No.: 000530–A01

User Profile

Your Name ______________________ Title ______________________

Company ______________________

Address ______________________

Telephone number (___) ______________ Date ______________

When you use the HP/Apollo system, what **job(s)** do you perform?

- ☐ Programming
- ☐ Application End User
- ☐ Hardware Engineering
- ☐ System Administration
- ☐ Other (describe) ______________________

Characterize your level of **experience** in using the HP/Apollo system:

- ☐ Experienced user (2+ yrs.)
- ☐ New user (6 mos. or less)
- ☐ Moderately experienced user (6 mos.–2 yrs.)

What programming **languages** do you use with the HP/Apollo system?

____

Distribution

How do you know what manuals are **available** to support the products you're using or want to use?

____

What is a major concern for you in **ordering** books?

____

How would you **evaluate** this book?

	Excellent		Average		Poor
Completeness	1	2	3	4	5
Accuracy	1	2	3	4	5
Usability	1	2	3	4	5

Additional Comments: ______________________

____

____

____

No postage necessary if mailed in the U.S.

cut or fold along dotted line

fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

Apollo Systems Division
A Subsidiary of Hewlett-Packard Company
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

fold